juliacon

UnitTestDesign.jl: Combinatorial design for unit tests

Andrew Dolgert[1] and Joseph Wagner[1]

[1]Institute for Health Metrics and Evaluation, University of Washington, Seattle, WA, USA

ABSTRACT

Combinatorial interaction testing is an automated way to generate test cases for unit tests. It's designed to be a best guess at the fewest unit tests that will give good decision coverage. This article discusses when to use this technique, offers a general approach to using automated test generation for different software testing applications, and shows how to apply it with the `UnitTestDesign` package in the Julia testing ecosystem.

Keywords

combinatorial interaction testing, automated test case generation, unit testing, Julia

1. Introduction

We would like to think that our unit tests of code match how much risk we perceived in that code, that they match the components by which we judge risk: the hazard for failure, the cost of mitigating failure, and how much a result matters. However, user studies of the behavior of test authors show that our choices about testing are ruled by what is convenient within a given testing framework [24].

The first unit tests for a new library often derive from mimicking how a user might write client code. These happy-path tests are narrative because they tell stories of successful use. As we develop, we strengthen tests where we see risk in code. For low-level code, this may mean testing corner cases. For high-level code, it may mean modifying the code to be easier to test [3]. Once we've done this software engineering, we can ask what tools within our testing framework can help improve test quality.

Some automated testing tools measure the quality of existing tests. Others select which tests to run depending on test coverage or recent code modifications. Let's focus on the first antidote to the biases inherent in narrative testing, automated generation of test cases.

```
using UnitTestDesign: all_pairs

@enum Strategy naive keyfitz greville monotonic
arg1 = [1, 4, 8]
arg2 = [:a, :b, :c, :d]
arg3 = [naive, keyfitz, greville, monotonic]
arg4 = ["none", "RK4", "quadrature", "QUADPACK"]
arg5 = [0.0, 9.3, -1.2, Inf]

test_cases = all_pairs(arg1, arg2, arg3, arg4, arg5)
for test_case in test_cases:
    result = samplefunction(test_case...)
    @assert known_invariant(result, test_case)
end
```

Here, the `all_pairs` function is a combinatorial interaction test (CIT) design that uses the sample argument values to choose a set of 16 test cases out of a total possible 768 test cases. The inputs are specified as explicit lists of argument values. The name, all-pairs, means that, if we look at the test cases, and we pick any two arguments, then pick any two values those arguments can take, there will be at least one test case that includes that pair of arguments.

```
julia> test_cases
16-element Vector{Vector{Any}}:
 [1, :a, naive, "none", 0.0]
 [4, :a, keyfitz, "RK4", 9.3]
 [8, :a, greville, "quadrature", -1.2]
 [1, :a, monotonic, "QUADPACK", Inf]
 [1, :b, naive, "RK4", -1.2]
 [8, :b, keyfitz, "none", Inf]
 [4, :b, greville, "QUADPACK", 0.0]
 [1, :b, monotonic, "quadrature", 9.3]
 [4, :c, naive, "quadrature", Inf]
 [1, :c, keyfitz, "QUADPACK", -1.2]
 [8, :c, greville, "none", 9.3]
 [8, :c, monotonic, "RK4", 0.0]
 [8, :d, naive, "QUADPACK", 9.3]
 [1, :d, keyfitz, "quadrature", 0.0]
 [1, :d, greville, "RK4", Inf]
 [4, :d, monotonic, "none", -1.2]
```

For a three-way CIT, we could pick any three arguments and any three values those arguments can take, and there will be at least one test case including that triple. In this article, we describe when to use CIT and how to use CIT within the Julia language's unit testing ecosystem. We introduce the `UnitTestDesign.jl` library which has several features that make it convenient to use.

—Different levels of coverage, including all-pairs, all-triples, and higher level coverage.

—Both deterministic and stochastic generators of test cases.

—An option to prime the generation of test cases with a manually-chosen list.

—The ability to forbid combinations of arguments.

CIT is a popular tool for testing frameworks [8] and one of a few common strategies for automated test case generation. A randomized strategy chooses argument values from the space of allowed arguments, usually with some bias towards choosing possible corner cases [16, 1]. Some tools introduce more structure to choosing random arguments. The `QuickCheck`(Haskell) and `Hypothesis` (Python) packages use customized generators to create streams of test cases [18]. Concolic testing records the execution of the function under test in order to generate subsequent test cases that are likely to increase line coverage [13, 23, 4]. We will compare these

methods in Sec. 5, after we understand the scope of combinatorial interaction testing.

2. Statement of Need

Compared to other techniques for automatic test case generation, CIT generates fewer tests, choosing them in a way designed to give good decision coverage [19, 9, 15].

There are two circumstances where this is crucial. The first is that you have to test a slow function, such as a large, Monte Carlo inference. If the function under test runs for hours or days, then selective testing can conserve limited resources.

The other circumstance is when there is a large test space. We've been talking about a function under test, but this is a stand-in for any test we can parameterize. Let's say we've written an application, that we've unit tested its important supporting functions, and we want to write user-level tests that look for problems from interactions among command-line arguments and choices in parameter files. This can be equivalent to testing a function with twenty or more arguments. Twenty arguments, with four possible values each, would lead to over 10^{12} ways to call this function.

The CIT in `UnitTestDesign` takes the time to optimize the choice of test cases so that they can test the code well. The assumption is that each if–then in the code will branch depending on interacting combinations of argument values, so CIT ensures all combinations of t arguments are tried in at least one test case. Here, $t = 2$ for two-way testing, also known as all-pairs testing [8], but studies have shown that wayness up to $t = 6$ can be useful [20].

Lastly, there are excellent external CIT tools that define a domain-specific language for test specification and offer features that improve usability [7, 15]. However, having CIT within the Julia environment not only makes it easier to import as a library, but also takes advantage of Julia's strengths as a language. The simple interface for arguments and forbidden combinations (covered below) depends on dynamic typing. The speed of the underlying algorithms depends on Julia's ability to create efficient numerical code.

3. How to Use Combinatorial Interaction Testing

3.1 Outline of the method

While you may already know how to use CIT from the snippet in the introduction, it can help to think of combinatorial interaction testing as a set of steps, each of which has choices to make.

(1) Identify the system under test.
(2) Decide how a single test case, consisting of argument values, should correspond to a unit test.
(3) Address the risk associated with different arguments.
(4) Pick a test generation strategy.
(5) Choose a method for checking the results of each test.

We'll explore these steps in this section.

3.2 Identify the system under test

We have assumed, thus far, that we are unit testing a function. In that case, the function has arguments and those arguments can take on particular values. Our test cases will correspond exactly to an invocation of the function. However, this need not be the case. The same idea of interacting choices applies to other test targets:

—Integration test of modules, where a module API produces many possible ways to call it.

—User-level tests of an application, where there are choices in dialog boxes, parameter files, and command-line options.

—Systems outside of Julia, such as the problem of hardware integration.

3.3 Decide argument values

3.3.1 What arguments represent. We must decide what the arguments represent and must select particular values for those arguments.

It's rare that a function expects arguments that really take only a few possible values. We use integers, floating-point, and strings, and a function argument can be any struct. When we specify tests for CIT, and we choose a few argument values, they are representatives of equivalence classes.

An *equivalence class* is a set of input values that would discover the same faults in the code. For instance, if a given function fails for `arg3<2.7`, then it will fail for `arg3=2` as well as `arg3=2.5`, so we consider those equivalent with respect to finding faults. You can choose them by looking at the code or by looking at a specification for the function, but it's possible to miss an equivalence class and miss the faults discovered by that equivalence class.

A weakness of CIT is that your tests are only as good as the argument values you choose. Even if you choose argument values such that some combination of them would find a failure, it's possible that the subset of test cases generated by the CIT might not include that combination. There is no guarantee that using automated testing to generate hundreds of tests will find a problem.

One way to defend against our uncertainty about equivalence classes is to combine combinatorial test generation with random testing. Decide that each argument value in the CIT specification represents a generator of values from an equivalence class. Then each test case, output from CIT, is a list of generators of random arguments. This introduces bias, and control, into random test selection. It points to treating test case generation as a continuum of techniques.

There are other creative ways to apply CIT. For instance, if an application reads a CSV file, we can use CIT to ensure rows of that CSV have a wide mixture of column values. This would improve branch coverage in code that reads the CSV as a data frame. Similarly, for module-level testing, arguments could represent choices about calls to a module's functions. Here, consecutive arguments become function calls that are consecutive in time.

3.3.2 Control value choices. Sometimes combinations of function arguments don't make sense. Imagine we have a function with two arguments.

```
arg1 = ["quick", "complicated"]
arg2 = [0.0, 1.0, 10.0]
```

For this example, if the first argument is `"quick"`, imagine that the function under test requires the second argument to be `0.0`. If we generate all combinations of `arg1` and `arg2`, then some combinations won't make sense. The domain of the function isn't merely the sum of the domains of the arguments.

One way to solve this problem is to rewrite the argument list to have a single argument that reflects only the allowed combinations. This replaces `arg1` and `arg2`.

```
arg12 = [("quick", 0.0), ("complicated", 0.0),
    ("complicated", 1.0), ("complicated", 10.0)]
```

It can be more convenient to tell the CIT algorithm that certain combinations are forbidden [20, 10]. Here, the `forbid` function returns `true` when argument values aren't allowed by the function under test.

```
forbid(arg1, arg2) = arg1 == "quick" && arg2 > 0.0
test_set = full_factorial(
    arg1, arg2; disallow = forbid)
```

For testing functions with many arguments, this is simpler to use than rewriting the argument list.

Sometimes we want to begin with tests that are particular happy paths. These might be statistically common call signatures, or they could be pre-specified in design documents as calls that must be tested. Given that our goal is to minimize resource usage, we should ensure that any tuples of argument values covered in these initial calls are included in our list of covered tuples.

```
must_test = [[1, "mid", 3.7, :relax],
             [1, "mid", 4.9, :relax]]
test_cases = all_pairs(
    [1, 2, 3], ["low", "mid" ,"high"],
    [1.0, 3.7, 4.9], [:greedy, :relax, :optim];
    seeds = must_test
    )
```

They are called seed cases and are specified with an explicit option in `UnitTestDesign`.

3.4 Address risk

The combinatorial testing approach gives you a lever with which to address perceived risk in code. You decide the wayness of coverage, which is the number of arguments for which all combinations of argument values must appear. As shown in Table 1, increasing wayness increases the number of test cases. Pairwise testing has two-way coverage, but higher wayness can yield some benefit [20]. Literature on how to generate these tests uses the word coverage to mean two-way, three-way, and so on. The term test coverage usually refers to which lines of code executed (line coverage), if–then decision tests (decision coverage), or branches that are executed during a test (branch coverage).

The two uses of the word coverage are related, because combinations with higher wayness as inputs are designed to lead to more decision coverage of code during execution. There isn't enough evidence for this connection for us to choose wayness according to code's cyclomatic complexity, and we are counseled against making coverage the goal of unit testing [12]. That leaves the test author to testing as much as you can afford but shifting those tests towards exercising code that is more complex.

For CIT, that means shifting test cases towards combinations of those arguments that affect the control flow of risky code. The first way to increase testing of an argument is to increase the number of values for an argument by adding corner cases. The second way is to increase the wayness of a set of arguments.

Table 1. Count of test cases from different test generation strategies.

		Total	t-way		Excursions	
Args	Vals	Combinations	Pairs	Triples	Single	Pair
5	4	1024	16	64	16	106
5	8	32768	96	768	36	526
10	4	$>10^6$	28	143	31	436
10	8	$>10^9$	113	1223	71	2276
40	4	$>10^{24}$	45	290	121	7141
40	8	$>10^{36}$	166	2388	281	38501

For example, if the function under test used an evolutionary algorithm that had several options for the representation of individuals, their mutation, and recombination, we might want to test these choices exhaustively with full-factorial coverage, while selecting pairwise coverage for other options. This would be done by adding a `wayness` argument to the function call.

```
array_of_forty_parameters = fill(1:4, 40)
test_set = all_pairs(
    array_of_forty_parameters...;
    wayness = Dict(3 => [[3,4,5,6]])
    )
```

Here, an all-pairs coverage of these 40 arguments would lead to 45 test cases, and an all-triples coverage would lead to 290 test cases, but we increase coverage on only the third through sixth arguments to a wayness of three, and this increases the total to 64 test cases. It's unclear whether mixing wayness will always yield a small set of test cases because mixed-level covering arrays don't have well-understood bounds [6], however sample runs show it does produce fewer test cases than raising coverage on all arguments.

3.5 Pick a test generation strategy

There are a few different strategies for combinatorial test generation in `UnitTestDesign`.

The *full factorial* method generates t-way coverage for t arguments. That is, it generates all combinations of values for all arguments. This is equivalent to what you could create with an t-deep for-loop.

The *excursion* method makes a first test case using the first provided value for each argument. Then, for one-way excursions, it walks the first argument through all values, keeping the rest of the arguments the same. Then it walks the second argument through all values, keeping the rest of the arguments the same, and so on. It asks how the function under test would perform if you were to change any single argument value. A two-way excursion tries changes to any two argument values in combination. It produces considerably more test cases than an all-pairs design, as shown in Table 1.

There are two ways the `UnitTestDesign` package creates covering arrays, also known as *fractional factorial designs*. The default generator is called `IPOG` because it is similar to the IPO-G algorithm [17]. It is deterministic, so it produces the same set of test cases for the same set of inputs. The other generator is called `GND`, and it uses a random number generator to search for covering arrays. It returns test cases that obey the same covering properties as

the `IPOG` generator, but it uses a method similar to the AETG generator [5]. In both cases, the algorithms vary from the published examples, as discussed in Sec. 4.

```
rng = Random.MersenneTwister(9790375)
slow_and_short = all_triples(parameters...;
    engine = GND(rng = rng, M = 50))
```

The `M=50` argument to the `GND` generator controls the number of times it guesses each new argument of a test case before it chooses an optimal value.

3.6 Evaluate results

When a testing suite contains only a few tests of a function, the test author can usually figure out how to check those few function outputs for failures. Automatic generation of test arguments creates a problem because it isn't paired with automatic generation of checks for test failures.

One solution is to create a parallel implementation of the function under test. It could be an earlier version of the function using a naïve algorithm. It could be a function that computes the same values using a different mathematical representation, such as using numerical integration instead of using the result of symbolic integration. It could be an external implementation, in another language.

It helps to consider the check for failure, not as an assertion about the result,

```
@assert result == oracle(test_case...)
```

but as a rule that takes both result and test case into account.

```
@assert invariant(result, test_case...)
```

For instance, some mathematical calculations solve inverse problems, so that the result can be fed into a related forward problem. The test becomes whether it can recover its input values. Another test would be whether nearby inputs give continuous outputs. Lastly, a technique to assert symmetries in function arguments, when such symmetries exist in the function under test, has been shown to find faults [22]. For instance, for some function under test, f, a symmetry test might check that `f(a, b)=f(2*a, b/2)`.

4. Optimization of Test Cases with Practical Constraints

The `UnitTestDesign` package expands on the implementation of two t-way combinatorial interaction testing algorithms. One is the greedy, non-deterministic algorithm known as AETG [5]. The other is a deterministic, parameter-choosing algorithm called IPO-G [17]. While these algorithms have different structure, we were able to add features to them using similar basic moves.

In order to support a feature where the test author can require certain pre-defined test cases, we add those test cases to the list of tuples to be covered, where a tuple is a combination of argument values. We just put that into the list and let the algorithms take care of covering it.

In order to support multi-level wayness in coverage, we use multiple rounds of algorithms designed for single wayness, starting at the highest wayness. For example, given a pairwise test generation strategy that requests all-triples over a set of five arguments, we first generate test cases that cover the all-triples and then present these as initial seed values to the all-pairs calculation.

In order to avoid forbidden combinations of arguments, we check the validity of the next possible argument at every moment in the algorithm that a next argument is chosen. This sometimes means that a test case being created could be thrown out and started new from a different seed argument value. This is a situation where it might help to use the Z3 theorem prover to reduce the need for retries, as some packages have done [19].

Like many greedy algorithms, these can be sensitive to small, sometimes unspecified, choices in implementation. We found that a successful strategy for designing these algorithms is to generate an ensemble of variants and test them against sample inputs.

For example, there is a common moment in these algorithms where a putative test case, partially specified, is matched against a coverage tuple, which will have t specified argument values for t-way coverage. Given just one argument from the test case and one argument from the tuple, there are five ways they can compare.

comparison	test case	tuple
ignores	*	*
skips	a	*
misses	*	b
matches	a	$b = a$
mismatches	a	$b \neq a$

Only the last comparison is clearly a mismatch. If we compare multiple arguments of the test case and tuple, we can quantify that comparison as having zero or more "ignores," zero or more "skips," and so-on. This means there are $2^4 - 1$ ways that a non-zero test case and tuple can compare, even if they have no mismatches. The `UnitTestDesign` library has tuned its search for test cases by optimizing across these matching algorithms.

There are plenty of improvements to make to these algorithms, such as the use of advanced data structures [21] or more adaptive optimization techniques like simulated annealing [20]. There are also completely different greedy algorithms [2, 14]. Any of these would be welcome to serve some use case if they can remain convenient to use within the testing framework.

5. Comparison of Approaches

Now that we've explored combinatorial interaction testing, let's return to place it in the context of other similar testing techniques. Compared to other methods for automated test case generation, CIT makes fewer tests, with some wayness guarantees.

Random testing also generates test cases, and we know, from Arcuri et al, that these test cases increase code coverage with increasing numbers of test cases in a predictable way [1]. Random testing has the benefit that you can run it for an arbitrarily long time during acceptance testing. There is also no requirement that values be completely random within the domain of arguments. They are almost always biased towards values that carry risk, such as empty strings or other corner cases. Combinatorial interaction testing is preferable to random testing when there are few test-running resources.

There are also more mathematical test generation methods, such as orthogonal arrays or Sobol sequences [11]. These give an even distribution of values within high-dimensional spaces. They would be excellent to, for instance, seed an optimization problem with local minima, but they aren't designed to explore the space of execution paths of code in the way that CIT is.

There are also powerful testing methods that combine automated generation of test inputs with observation of test execution and outputs. For tests that run reasonably quickly, these strategies can shorten the time to find faults in the code, even if they don't shorten the length of a particular test run.

Property-based testing, for instance, offers a specification to describe the domain of each argument, or sets of arguments, so that the test author can focus on the domain of the function instead of the exact test suite. Tools of this kind can watch for failed tests, and then they can simplify the test arguments while checking for failure, in order to find the simplest test that fails. This is extraordinarily useful for fault-finding but, again, will use more testing resources than CIT in the absence of failures [18].

Concolic testing performs the remarkable feat of executing a test case and recording that execution in order to learn which argument values control choices at each decision in the code. Then the concolic algorithm chooses a different argument value and tries the test again. This is an automated way to learn corner cases in the code, which should help with the challenge of deciding equivalence classes for arguments. At the same time, concolic testing generally uses logic solvers to decide equivalence classes, and these can be idiosyncratic for mathematical code [13, 23]. The hardest part about concolic testing may be running it, conveniently, within the unit testing framework.

All of these approaches to automated test generation create more tests with less code than manual alternatives. This makes a test suite less brittle as the code-under-test is modified. It can also create test suites with a reassuring number of test cases, but none of the automated techniques above remove risk. We still need to rely on software engineering techniques, such as tracking of historical bug report rates, in order to understand the maturity of a code base.

6. Conclusion

Combinatorial interaction testing is an advanced technique because it's tailored to a difficult situation, when you can't afford not to spend the time to make your best guess at test cases. However, this kind of automation is easy to specify and run in Julia's ecosystem. The most difficult part of the transition to automated test case generation is shifting from manually-computed checks of test results to checks that work for any combination of arguments, as described in Sec. 3.6. A lot of the strategies in Sec. 3 apply to any kind of automated test case generation. They point to the broad applicability of automation for testing and to the effectiveness of tools that offer control to the test author.

7. References

[1] A Arcuri, M Z Iqbal, and L Briand. Random testing: Theoretical results and practical implications. *IEEE Trans. Software Eng.*, 38(2):258–277, March 2012. doi:10.1109/TSE.2011.121.

[2] Andrea Calvagna and Angelo Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance: combinatorial test suites by coverage inheritance. *Softw. Test. Verif. Reliab.*, 22(7):507–526, November 2012. doi:10.1002/stvr.466.

[3] Vishal Chowdhary. Practicing testability in the real world. In *2009 International Conference on Software Testing Verification and Validation*, pages 260–268. IEEE, 2009. doi:10.1109/ICST.2009.53.

[4] Valentin Churavy. Concolicfuzzer. `https://github.com/vchuravy/ConcolicFuzzer.jl`, 2019.

[5] David M Cohen, Siddhartha R Dalal, Michael L Fredman, and Gardner C Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997. doi:10.1109/32.605761.

[6] Myra B Cohen, Peter B Gibbons, Warwick B Mugridge, and Charles J Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society. doi:10.1109/ICSE.2003.1201186.

[7] Jacek Czerwonka. Pairwise testing in real world: Practical extensions to test case generators. In *24th Pacific Northwest Software Quality Conference*, volume 200, 2006.

[8] Jacek Czerwonka. Pairwise testing: Combinatorial test case generation. `https://jaccz.github.io/pairwise/`, 2021.

[9] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Softw. Test. Verif. Reliab.*, 15(3):167–199, September 2005. doi:10.1002/stvr.319.

[10] Mats Grindal, Jeff Offutt, and Jonas Mellin. Handling constraints in the input space when using combination strategies for software testing. Technical Report HS-IKI-TR-06-001, School of Humanities and Informatics, University of Skövde, 2006.

[11] Yuanzhen He and Boxin Tang. Strong orthogonal arrays and associated latin hypercubes for computer experiments. *Biometrika*, 100(1):254–260, 2013. doi:10.1093/biomet/ass065.

[12] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, May 2014. doi:10.1145/2568225.2568271.

[13] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. doi:10.1145/360248.360252.

[14] Ugur Koc and Cemal Yilmaz. Approaches for computing test-case-aware covering arrays. *Softw. Test. Verif. Reliab.*, 28:e1689, November 2018. doi:10.1002/stvr.1689.

[15] D R Kuhn, R N Kacker, and Y Lei. Practical combinatorial testing. Technical Report 800-142, National Institute of Standards and Technology, U.S. Department of Commerce, 2010. doi:10.6028/NIST.SP.800-142.

[16] Leonidas Lampropoulos, Benjamin C Pierce, Li-Yao Xia, Diane Gallois-Wong, Cătălin Hriţcu, and John Hughes. Luck: A probabilistic language for testing. In Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva, editors, *Foundations of Probabilistic Programming*, pages 449–487. 2020. doi:10.1017/9781108770750.014.

[17] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. IPOG/IPOG-D: efficient test generation for

multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18(3):125–148, September 2008. doi:10.1002/stvr.381.

[18] Andreas Löscher and Konstantinos Sagonas. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 70–80, 2018. doi:10.1109/ICST.2018.00017.

[19] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, January 2011. doi:10.1145/1883612.1883618.

[20] J Petke, M B Cohen, M Harman, and S Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Trans. Software Eng.*, 41(9):901–924, September 2015. doi:10.1109/TSE.2015.2421279.

[21] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 254–264, 2011. doi:10.1145/2001420.2001451.

[22] S Segura, G Fraser, A B Sanchez, and A Ruiz-Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, September 2016. doi:10.1109/TSE.2016.2532875.

[23] Xinyu Wang, Sun Zhenbang, Chen Jun, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In *2018 ACM/IEEE 40th International Conference on Software Engineering*, pages 291–302, New York, NY, USA, 2018. doi:10.1145/3180155.3180177.

[24] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark, and Kristina Lundqvist. Impediments for software test automation: A systematic literature review. *Softw. Test. Verif. Reliab.*, 27(8):e1639, December 2017. doi:10.1002/stvr.1639.