Introduction to property based testing

Tü.λ—Functional Programming Night Tübingen

2026-01-28 Stefan Walter

Slides and code: https://github.com/stfnw/talk-introduction-to-property-based-testing

Motivation

Testing: Correctness, Safety

Outline

- *pure* code
- *stateful* code
- some case studies

Introductory example

- sorting lists of integers
- `intro.fsx`

Example based tests

manually hand-code specific examples

Advantages

- easy to write and understand (concrete/specific)
- specific edge cases
- regression

Disadvantages

- small number of tests
- limited coverage: Hard to think of representative data covering all cases
- miss edge cases, non-exhaustive

Property based tests

- originally introduced from functional programming by John Hughes and Koen Claessen with the QuickCheck library for Haskell
- randomly auto-generate tests / test data, and check *properties* of the implementation
- i.e. exptected relationship between inputs and outputs

Advantages

- huge number of tests
- can cover more code paths / higher code coverage
- can uncover unexpected test cases

Disadvantages

- more abstract
- harder to write
- still not exhaustive (not formal verification); may still miss edge cases

How to specify desired properties?

without re-implementing the system-under-test / duplicating logic

Strategies for systematically specifing properties

good (re)sources:

- Scott Wlaschin The "Property Based Testing" series
- John Hughes How to specify it! A guide to writing properties of pure functions | Code Mesh LDN 19

  J. Hughes, “How to Specify It!: A Guide to Writing Properties of Pure Functions,” in Trends in Functional Programming, vol. 12053, W. J. Bowman and R. Garcia, Eds., in Lecture Notes in Computer Science, vol. 12053. , Cham: Springer International Publishing, 2020, pp. 58–83. doi: 10.1007/978-3-030-47147-7_4.

Invariants

something stays unchanged

Idempotence

doing something twice is the same as doing it once

Hard to solve, easy to verify

- e.g. NP hard problems
- example: scheduling / resource allocation, `scheduling.fsx`
  - *verification* of output of `findSchedule` is easy
  - if sum of task times $>$ sum of machine capacities: no valid schedule exists

Inverse / round-trip

- compression - decompression
- serialization - deserialization
- write - read
- addition - subtraction
- encode - decode
- render - parse
- do - undo - redo

Divide into subproblems

relate problem statement to a smaller subproblem

- divide and conquer algorithms
- structural induction

Metamorphic properties

- how does modifying the input change/affect the output? (relationship)
- combine operations

Metamorphic properties examples

- search engine: [^1]
  when a search yields result A
  a constrained search (e.g. with date filter) yields result B
  then $B \subseteq A$
- image recognition of position of specific object:
  - rotate input image, also rotates output position
  - changing color to grayscale shouldn't affect result
- filter then sort == sort then filter
- negate all numbers in list then sort == sort list then negate all numbers

1: From Automate Your Way to Better Code: Advanced Property Testing (with Oskar Wickström)

Test oracle / differential testing

second implementation that solves problem

- legacy system is oracle for replacement system
- compare multiple existing implementations of a standard

Model-based properties

special case of test oracle where you provide the model as one implementation:

- re-implement aspect of functionality you care about in a simpler way
- differentially test against that

example use cases:

- optimized complicated implementation vs slow but simpler (more obviously correct) implementation
- parallel vs single threaded implementation
- in-memory model vs multi-system application deployment

Strategies for systematically specifing properties summary

- (doesn't crash)
- Invariants
- Idempotence
- Hard to solve, easy to verify
- Inverse / round-trip
- Divide into subproblems
- Metamorphic properties
- Test oracle / differential testing
- Model-based properties

Efficiency

recommendations from overview of strategies, test against example bugs: [^1]

- model-based
- metamorphic

1: From Chapter 5.3, J. Hughes, “How to Specify It!: A Guide to Writing Properties of Pure Functions,” in Trends in Functional Programming, vol. 12053, W. J. Bowman and R. Garcia, Eds., in Lecture Notes in Computer Science, vol. 12053. , Cham: Springer International Publishing, 2020, pp. 58–83. doi: 10.1007/978-3-030-47147-7_4.

Outline

- *pure* code
- *stateful* code
- some case studies

Property testing of *purely functional* code

- no side effects, no implicit state (stateless)
- mathematical function: same input $\Rightarrow$ same output
- i.e. everything covered so far

most useful:

- in functional programming with adequate language support (immutability)
- defense, when you are developer in control of the code

Property testing of *stateful* code

- side effects and external state
- dynamically created resources

most useful:

- test of real-world / existing stateful systems
- interface / api testing
- offense, finding bugs end-to-end

Typical approach/strategy

model-based

- re-implement relevant aspect of system-under-test in a model
  has internal state and supports operations on it
- generate a random list of function calls / actions (program)
- execute each action in the program against real-world system *and* model
- assert that results for each are the same

Example: filesystem

pseudo-code interface:

- `open(path: string) -> fd`:
  Open "path", return opaque symbolic resource handle identifying the open file (file descriptor / handle).
- `readall(fd, size: int) -> [u8]`
  Read "size" bytes from an opened file, return corresponding byte buffer.
- `writeall(fd, buf: [u8]) -> ()`
  Write byte buffer, return unit (no return value).

Example list of actions against a filesystem

```
fd0 = open("/tmp/path")
writeall(fd0, "hello world")
fd1 = open("/tmp/path")
readall(fd1, 11)
```

Example execution

against real file system:

```
open("/tmp/path")                -> 3
writeall(fd0, "hello world")     -> ()
open("/tmp/path")                -> 4
readall(fd0, 11)                 -> "hello world"
```

against a correct model:

```
open("/tmp/path")                -> FD0
writeall(fd0, "hello world")     -> ()
open("/tmp/path")                -> FD1
readall(fd0, 11)                 -> "hello world"
```

Is a model actually necessary?

Yes! drives test generation.

Disadvantages

- needs model as runnable specification
- duplicate implementation

Advantages

- can test real-world systems
- grey-box testing, only interface must be known

Some case studies

- *pure:* URI parsing in .NET
- *stateful:* Reproducing CVE-2022-0847 (Dirty Pipe)
  (a security-relevant logic bug in the Linux kernel)

URI parsing in .NET

`uri-parsing.fsx`

Reproducing CVE-2022-0847 (Dirty Pipe)

- a security-relevant logic bug in the Linux kernel
- https://dirtypipe.cm4all.com/
- found by Max Kellermann

Gist

- in certain cases permissions are not checked and one can write to a file one shouldn't be able to
- can be used for privilege escalation
- ultimately caused by an *optimization*

Example trace of actions during exploit

(output from strace shows syscalls and parameters during run as unprivileged user)

```
openat(AT_FDCWD, "/etc/passwd", O_RDONLY)             = 3
pipe2([4, 5], 0)                                      = 0
fcntl(5, F_GETPIPE_SZ)                                = 65536
write(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 65536) = 65536
read(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\"..., 65536) = 65536
splice(3, [0], 5, NULL, 1, 0)                         = 1
write(5, "AAAA", 4)                                   = 4
read(3, "rAAAAx:0", 8)                                = 8
```

last read shouldn't return written data! (file opened read-only)

Reproducer

- I wrote a poc reproducer for it
- https://github.com/stfnw/reproducer-poc-CVE-2022-0847
- generates actions (syscalls) with appropriate parameters
- detects misbehavior by running actions against real-world file system and a model
- warning: lots of magic numbers / chosen parameters, not *fully* realistic,
  state space too large to traverse in acceptable time without guidance/feedback

Demo

Lessons learned

- implementing a model is useful for understanding interface better
- model gets more complicated than expected very quickly
- handling all error conditions and making sure generators only produce valid commands that don't mess up implicit program state is surprisingly hard

Mildly related

- more aspects of model-based stateful testing:
  parallel testing, concurrency issues, distributed systems (test interleavings)
- natural next step for vulnerability search: combine with modern fuzzing
  - mutation based, coverage guided
  - use properties as crash-oracle for fuzzing for high-level logic bugs
    $\Rightarrow$ *I'm interested in that; you too? Let me know!*
- (lightweight) formal methods
- formal specifications and model checking / ltl / tla+ / alloy
- deterministic simulation testing (TigerBeetle, Antithesis)

Sources and further resources

- Scott Wlaschin:
  - The "Property Based Testing" series; especially sections on choosing properties in practice
  - F# Online - Scott Wlaschin - Property Based Testing
  - The lazy programmer's guide to writing thousands of tests - Scott Wlaschin
- J. Hughes, “How to Specify It!: A Guide to Writing Properties of Pure Functions,” in Trends in Functional Programming, vol. 12053, W. J. Bowman and R. Garcia, Eds., in Lecture Notes in Computer Science, vol. 12053. , Cham: Springer International Publishing, 2020, pp. 58–83. doi: 10.1007/978-3-030-47147-7_4.
- John Hughes - How to specify it! A guide to writing properties of pure functions | Code Mesh LDN 19

Sources and further resources

- QuickCheck:
  - K. Claessen and J. Hughes, “QuickCheck: a lightweight tool for random testing of Haskell programs,” in Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, ACM, Sep. 2000, pp. 268–279. doi: 10.1145/351240.351266.
  - K. Claessen and J. Hughes, “Testing monadic code with QuickCheck,” in Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, Pittsburgh Pennsylvania: ACM, Oct. 2002, pp. 65–77. doi: 10.1145/581690.581696.
  - https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html

Sources and further resources

- F# property based testing library FsCheck
- Python property based testing library with good documentation and novel shrinking strategy: Hypothesis

  Paper on internal shrinking used by Pythons Hypothesis: D. R. MacIver and A. F. Donaldson, “Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper),” LIPIcs, Volume 166, ECOOP 2020, vol. 166, p. 13:1-13:27, 2020, doi: 10.4230/LIPICS.ECOOP.2020.13.

Sources and further resources

- The sad state of property-based testing libraries
- Podcast "Developer Voices" by Kris Jenkins (interviews/case studies):
  - Automate Your Way to Better Code: Advanced Property Testing (with Oskar Wickström)
  - From Unit Tests to Whole Universe Tests (with Will Wilson)
- Property-Based Testing in a Screencast Editor: Introduction by Oskar Wickström
- Race Conditions, Distribution, Interactions Testing the Hard Stuff and Staying Sane by John Hughes
- Property Based Testing: Concepts and Examples - Kenneth Kousen

Sources and further resources

- Code samples
  - https://github.com/kousen/pbt_jqwik
  - https://github.com/swlaschin/PropBasedTestingTalk

Questions and discussion

- When did you use property testing?
- What was a surprising or interesting property that helped you specify system behavior?
- Can you share further strategies for expressing properties?