# Practical Property-based testing in scientific software

Mousumi Chattaraj, Luiz Irber, and Li Li

University of California, Davis
{mchattaraj,lcirberjr,llili}@ucdavis.edu

## ABSTRACT

Developing correct software is difficult and expensive. On top of this, research groups need to balance the scientific needs and software correctness due to lack of resources or incentives, among many other parameters. In this article we present our findings while exploring a property-based testing framework in the context of scientific applications, with special focus on biological sequencing data analysis. Property-based testing provides a good tradeoff between development effort and software correctness, and can be a viable alternative to unit testing and fuzzing in scientific contexts.

## 1. INTRODUCTION

Correct software is expensive to develop. Scientific software suffers even more due to lack of software development expertise among scientists and maintainability problems: people leave labs, usually without creating enough documentation or processes for continued development. We would like to explore ways to increase our confidence in the correctness of scientific software or, more realistically, find cheap ways of show when it is incorrect.

The khmer project is a mixed Python/C++ codebase implementing a few core data structures (sketches) and many methods and analysis on top of it. The source code is available on GitHub under a 3-clause BSD license, and every pull request is tested on a Continuous Integration server (Jenkins). It has high coverage (90% line coverage) derived from manually written ( 600) unit tests, but many tests were written just to trigger coverage, with no particular focus on correctness.

According to Claessen and Hughes [2], passing numerous tests in trivial cases can create a false sense of security, and many of the unit tests currently in khmer fits the description. One technique available for better testing is property-based testing, where instead of testing for specific cases the test is generalized (in terms of properties), and a library generates inputs to try to falsify these properties. If every property is verified the software cannot be considered as correct and fully tested, but it does provide stronger evidence it is not wrong in complex cases.

## 2. MOTIVATING EXAMPLE

Hashing is an essential operation in khmer, since all the core data structures depend on good hash functions to perform correctly. Benefiting from a public development process coordinated through GitHub, an exploratory branch was created with an accompanying issue and discussion: https://github.com/dib-lab/khmer/issues/990

This original exploration used a property from the hash function, reversibility: any string used as input generates a hash value (an integer), which can be used to generate the original string again. The implementation using Hypothesis is the following:

Listing 1: Motivating example: testing a reversible hash function

```
@given(text("ACGT"))
def test_forward_hash_no_rc(kmer):
    ksize = len(kmer)
    assume(ksize > 0)
    assume(ksize < 32)

    rh = reverse_hash(
        forward_hash_no_rc(kmer, ksize),
        ksize)
    assert rh == kmer
```

This example contains some important aspects:

1. the input *kmer* is generated automatically from a specification (the *given* decorator), meaning each *kmer* is a string with alphabet "ACGT".

2. the *assume* function asserts that the generated *kmer* is not an empty string, and also not longer than 32 characters. This is used by Hypothesis to guide the input generation.

3. the conditions are tested using normal Python syntax (assert, variables, function calls)

The input specification was added later, because the code was not prepared to deal with empty strings, strings longer than 32 characters or with a different alphabet. These are all bugs that need to be fixed, and they are good to show the approach. This example is very similar to typical unit testing in Python, while the Hypothesis extensions to the concept allow both property-based testing and fuzzing usage.

## 3. TECHNICAL APPROACHES

We extended the available infrastructure for running tests in khmer to also run our new tests. During development we also set up Travis CI for continuous integration and Coveralls for coverage reports (both for convenience, because khmer already have a Jenkins CI server up and running, but it is not as easy to track coverage in it). Every change committed to the repository is tested, generating

coverage data (in *gcov* format) and uploaded to Coveralls for analysis. Since Coveralls doesn't support function coverage, we also set up LCOV and uploaded the generated report to another server. Links for the reports are available in the project repository README: `https://github.com/luizirber/ecs260-property/`

## 3.1 Incremental exploration

Our plan is to understand what are the best options to generate meaningful Hypothesis tests to find incorrect behavior in khmer. We'll do progressive coverage and exploration of the khmer codebase, starting from simpler pieces and building up knowledge to be able to tackle complex regions. This allows us to start from valid input (to figure out what properties make sense to be tested), and then increase the complexity of the code by generating invalid inputs too. Most tests will deal with biological data in the form *sequences*, strings on a restricted alphabet "ACGT", or *records* composed of a *name* (initially restricted to printable ASCII characters), *sequence* and optionally a *quality score* for each character in the *sequence*. We can increase the complexity (and the validity) of inputs by allowing extra characters in any of the fields of a record. In the extreme case, after the code is sufficiently resilient, we can also start generating unustructured random data, turning the whole approach into a fuzzing test. Fuzzing Qi et al. [10] is a technique introduced in 1990 which found failures in UNIX utilities by generating random inputs, but it doesn't help with correctness verification, only to find crashing bugs.

We split the plan into four areas:

### *3.1.1 Hashing*

Hashing is the implementation of functions mapping strings to an integer space. Depending on the implementation this operation is reversible.

### *3.1.2 HyperLogLog*

HyperLogLog is an algorithm that finds the number of unique elements in a multiset. It is a probabilistic cardinality estimator, consuming less memory than exact solutions. The algorithm uses a hash function in the original multiset to produce uniformly-distributed random numbers that have the same cardinality Flajolet et al. [6]. Usually a *set* can be used as an oracle in tests involving HyperLogLog, since the cardinality estimation is an operation equivalent to the size of the set.

### *3.1.3 Bloom Filter*

A Bloom Filter is a data structure designed to check the presence of an element in a set rapidly and memory-efficiently. A bit vector is the base structure of a bloom filter, where each empty cell represents a bit and the number below is its index. Mill [9] In order to add an element to the table, we will hash it a few times and then set the bits in the bit vector at the index of those hashes to 1. False positive matches are possible, but false negatives are not. A *set* is the ideal oracle for a Bloom Filter.

### *3.1.4 Count-Min Sketch*

Count-Min Sketch is a probabilistic data structure that projects events to frequencies with the implementation of hash functions. The output is a frequency table Cormode [3]. The goal is to consume a stream of events one at a time, and count the frequency of different types of events in it without storing the whole stream Cormode and Muthukrishnan [4]. In the Python standard library there is a class, *collection.Counter*, that can be used as an oracle for frequency counting.

## 3.2 Unit tests adaptation

Many unit tests for khmer are based on fixed values, and checking the results for this specific value. Hypothesis has a decorator, *@example*, that can be used to run specific inputs. We can use this to to prototype new tests: by copying the unit test and setting the example, we have a working test and then can generalize for some property.

# 4. EVALUATION METHODOLOGY

It is hard to find a good metric for success in this project: number of bugs found wouldn't necessarily represent interesting or important bugs, but at the same time we can't guarantee we'll find any non-trivial bug. There is also the complexity of reasoning about the data structures to find testable properties and how to set up the tests. Nonetheless, we would like to track how many method calls we can cover, starting from the core data structures and moving to specialized methods (the ones used directly by data analysis scripts). We're not aiming at complete line or branch coverage, but it is likely that we can get good coverage with the Hypothesis tests, even if inconsistent between runs because new test cases will be generated.

Coveralls is a web service to help track the coverage of our code, and since all the code is hosted on GitHub it is easy to integrate it to the workflow. We are then using Travis CI to get a report on the coverage level. Travis can be enabled by the click of a button from GitHub repository. We are also using LCOV to report function coverage, because Coveralls doesn't support it. LCOV is a graphical front-end for GCC's coverage testing tool gcov. It collects data from gcov for more than one source file and creates web pages containing the source code with coverage details. It also adds pages for easy navigation in the file structure. LCOV supports line coverage, function coverage and branch coverage measurement.

Finally, we want to document and generate guidelines for testing similar software, as well as other biological data analysis projects or scientific in general. This document can be used as a quickstart and introduction to these testing techniques in specialized fields.

# 5. EXPERIMENTAL RESULTS

We wrote 16 property tests, with at least two tests covering each of the areas proposed for the incremental exploration. These tests were enough to generated a method coverage of 37.9% in the C++ code (the *lib* directory), as shown in table 1. The *khmer* directory contains the glue code for calling C++ code from Python, and while important it wasn't part of our initial focus.

| Directory | Line Coverage | Functions |
|---|---|---|
| khmer | 14.0% (217/1554) | 19.5% (29/149) |
| lib | 19.1%(580/3042) | 37.9% (96/253) |
| Total | 17.3% (797/4596) | 31.1% (125/402) |

Table 1: Line and function coverage information from LCOV.

## 5.1 Interesting bugs found

### 5.1.1 The HyperLogLog cardinality estimation is out of the error bounds

When running the test for HyperLogLog cardinality estimation, a total error rate of 1% or less is expected. We set up the test to also consider when less than one hundred unique elements are tested, because over or underestimating by 1 or 2 is still acceptable (we want to estimate large cardinalities). However, even considering the hashing function is not perfectly uniform (and so smalls variations on the error rate are expected), Hypothesis found falsifying examples for both low cardinality cases (10 items in the set, 7 in the HyperLogLog) and large cardinalities (where it was closer to 3%.

Listing 2: HyperLogLog cardinality estimation test

```
@given(st.lists(st_kmer,
                min_size=10,
                unique_by=lex_rc))
def test_hll_cardinality(kmers):
    oracle = set()
    hll = khmer.HLLCounter(0.01, KSIZE)

    for kmer in kmers:
        oracle.update([kmer])
        hll.consume_string(kmer)

    if len(oracle) < 100:
        assert abs(len(oracle) - len(hll)) <= 2
    else:
        error = (abs(len(hll) - len(oracle))
                 / len(oracle))
        assert error <= 0.02
```

The falsifying example can now be used to understand why the estimation is not behaving as expected, but it will demand more analysis to figure this.

Listing 3: HyperLogLog cardinality estimation test output, with falsifying example

```
======================================================================
FAIL: Testing HyperLogLog cardinality estimation.
----------------------------------------------------------------------
Traceback (most recent call last):
  (...)
  File "test_hypothesis.py", line 230, in test_hll_cardinality
    assert abs(len(oracle) - len(hll)) <= 2, (len(oracle), len(hll))
AssertionError: (10, 7)
---------- >> begin captured stdout << ----------
Falsifying example: test_hll_cardinality(kmers=['CTTCGTCCTTGTC',
'CCTTGCTTTTTGT', 'GCCGCGCTGTGCT', 'GCGTTTTCGTTTC', 'AAAAAAAAAAAAA',
'AAACAAAAAAAAA', 'AAAAAAAAACAAA', 'AAAAAAAAAAACA', 'ACAAAAAAAAAAA',
'CTCTCTCCCTTCC'])
---------- >> end captured stdout << ----------
```

### 5.1.2 Introducing invalid data, finding inconsistent behavior

All tests are still using only valid data as input, except one: *test_countgraph_consume_fasta*. We used this test to start implementing invalid input generation, and it was an appropriate choice because *Countgraph* (the library name for Count-Min sketches) implements both a *consume_fasta* method (for reading multiple records from a file) and *consume*, which consumes the sequence from only one record. We allow invalid records to be generated by accepting any Unicode character, either for names or sequences. Because generating long lists of records without errors would be very difficult if we chose each element between valid and invalid, we opted for generating either a list with only valid records (same behavior from initial implementation), or generating lists with invalid records (which might have valid records too, but rarely). All these options can be controlled by using search strategies defined by default on Hypothesis: *one_of* and *lists*. *st_record* and *st_invalid_records* are the strategies we defined using basic default strategies to generate both valid and invalid records.

Listing 4: Countgraph sequence consumption test

```
@given(st.one_of(st.lists(st_record),
                 st.lists(st_invalid_record)))
def test_countgraph_consume_fasta(records):
    cg_fasta = khmer.Countgraph(KSIZE, TABLE_SIZE, N_TABLES)
    cg_string = khmer.Countgraph(KSIZE, TABLE_SIZE, N_TABLES)

    fasta = fasta_build(records)
    with NamedTemporaryFile() as temp:
        temp.write(fasta.encode('utf-8'))
        temp.flush()

        try:
            cg_fasta.consume_fasta(temp.name)
        except OSError:
            assert fasta == ''

    for record in records:
        cg_string.consume(record['sequence'])

    assert cg_fasta.n_unique_kmers() == cg_string.n_unique_kmers()
    assert cg_fasta.n_occupied() == cg_string.n_occupied()
    assert cg_fasta.hashsizes() == cg_string.hashsizes()
```

Surprisingly, we didn't even have to finish implementing the property checks to find a bug: during prototyping we hit an error and we found inconsistencies in the behavior of both functions. *consume_fasta* accepted an invalid record without errors, but when *consume* was executed on the same record a validation error was reported.

Listing 5: Inconsistent behavior between consume_fasta and consume

```
======================================================================
ERROR: tests.test_hypothesis.test_countgraph_consume_fasta
----------------------------------------------------------------------
Traceback (most recent call last):
  (...)
  File "tests/test_hypothesis.py", line 199, in test_countgraph_consume_fasta
    cg_string.consume(record['sequence'])
ValueError: string length must >= the hashtable k-mer size
---------- >> begin captured stdout << ----------
Falsifying example: test_countgraph_consume_fasta(records=[
                {'name': '0', 'sequence': '0'}])
---------- >> end captured stdout << ----------
```

The question is: Which behavior is correct? *consume_fasta* ignored invalid records, but this should probably be controlled with a parameter for the method, instead of silently ignoring records. *consume* raised an exception, but depending on the analysis being done we might want to just ignore errors instead of crashing the program.

## 5.2 Generalizing existing unit tests

Because we already have many unit tests available for khmer, we wanted to figure out how to leverage this knowledge to make it easier to write Hypothesis test. As an example, we show here how we converted the unit test for the *Nodegraph.filter_if_present* method into a Hypothesis test.

The *filter_if_present* unit test takes two previously generated files, *filter-test-A.fa* and *filter-test-B.fa'*, and reads the first one as a mask using *consume_fasta*. *Nodegraph.filter_if_present* then takes the second file as input, and filter out any sequence containing *k-mers* from the first file, writing the output to a third file. Finally, this final file is read as records and two specific values are checked: if only one record remained, and if the record name is '3'.

Listing 6: filter_if_present unit test

```
def test_filter_if_present():
    nodegraph = khmer._Nodegraph(32, [3, 5])
```

```
    maskfile = utils.get_test_data('filter-test-A.fa')
    inputfile = utils.get_test_data('filter-test-B.fa')
    outfile = utils.get_temp_filename('filter')

    nodegraph.consume_fasta(maskfile)
    nodegraph.filter_if_present(inputfile, outfile)

    records = list(screed.open(outfile))
    assert len(records) == 1
    assert records[0]['name'] == '3'
```

While useful to check if it works in this case, the test can be generalized as follows:

1. generate two lists of records: *mask* and *records*
2. write both lists to files: the *mask_fasta* file contains all records from the *mask* list, while the *input_fasta* file contains all sequences (from both *mask* and *records*).
3. instead of checking for specific values in the filtered records, write an assertion verifying if all sequences present in *mask* are absent in the filtered records.
4. finally, use the *@example* decorator to list the records from the original files from the unit test, guaranteeing that we are still checking the initial case every time.

While more verbose due to file manipulation, the following code implement all items from the list and can be executed for any generated input:

Listing 7: filter_if_present Hypothesis test

```
@given(st.lists(st_record), st.lists(st_record))
@example([{"name": "a", "sequence": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}],
         [{"name": "1", "sequence": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"},
          {"name": "2", "sequence": "GAGATCAGAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"},
          {"name": "3", "sequence": "AGAGATACACAAGATAGAGAGACCCAGGAGGGGG"}])
def test_nodegraph_filter_if_present(mask, records):
    ng = khmer.Nodegraph(KSIZE, TABLE_SIZE, N_TABLES)

    mask_fasta = fasta_build(mask)
    with NamedTemporaryFile() as maskfile:
        maskfile.write(mask_fasta.encode('utf-8'))
        maskfile.flush()

        try:
            ng.consume_fasta(maskfile.name)
        except OSError:
            assert mask_fasta == ''

    input_fasta = fasta_build(records)
    with NamedTemporaryFile() as outfile:
        with NamedTemporaryFile() as inputfile:
            inputfile.write(mask_fasta.encode('utf-8'))
            inputfile.write(input_fasta.encode('utf-8'))
            inputfile.flush()

            try:
                ng.filter_if_present(inputfile.name, outfile.name)
            except OSError:
                assert input_fasta == ''

        with screed.open(outfile.name) as filtered:
            filtered_records = [r for r in filtered]

    assert all(m['sequence'] not in r['sequence']
               for m in mask
               for r in filtered_records)
```

# 6. RELATED WORK

## 6.1 QuickCheck

Property-based testing has its roots on the Haskell programming language, in a module called QuickCheck (Claessen and Hughes [2]). Instead of defining test cases for specific values (like corner cases), QuickCheck proposes that tests should be verifiable by checking properties and invariants that describe what is expected for any input, and then generating inputs to try to falsify the test. Haskell is a functional language that avoids side-effects ("pure"), making it simpler to reason about properties as mathematical aspects, and apply concepts across many domains. This approach doesn't work so well in other languages, and for Python there is a library called Hypothesis implementing a mix of unit testing, property-based testing, and fuzzing.

## 6.2 QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

QuickCheck(Claessen and Hughes [2]) assists in testing program properties (Which are Haskell functions) for Haskell programmer. The properties can be tested automatically on random input or custom set data. Testing is a popular and expensive approach to check software quality. Reducing the cost of testing has been a major motivation to automate the process. Two relatively old ideas, oracles and random testing are combined and work well for Haskell. The first step is defining properties, with features like quantifiers, conditionals and data monitors. Next step is defining generators, which are type-based default random test data generators. The last step is providing an embedded language to specify custom test data generators. It has also been proven that the functional nature allowed for local and fine-grained properties, and this tool is lightweight and easy to use.

## 6.3 Hypothesis

Hypothesis (MacIver [8]) is a Python library for creating simple and powerful unit tests to find edge cases. It asks user to write cases that assert something should be true for every case instead of just the ones that could be thought of. For a normal unit test, the first step is to setup some data, and then performing some operations on the data, and at last asserting something about the result. However Hypothesis asks user to write test cases for all data matching some specification, and then performs some operation on the data, and makes the assertion in the end. This is often called property based testing. Hypothesis works by generating random data that match the specification for the data and checks if the guarantee still holds in the case. If not it will take the example and cut its size to simplify it until a much smaller example that still causes the problem is found. Hypothesis then saves that example for later use.

## 6.4 khmer

khmer (Crusoe et al. [5]) is a free software library that works with fixed length DNA words (also known as $k$-mers) and implements a probabilistic k-mer counting data structure. Currently khmer is implemented in $C++$ and Python. The data structures and graph traversal code are implemented in $C++$, and then wrapped for Python in hand-written C code. khmer is primarily developed on Linux for Python 2.7 and 64-bit processors, and several core developers use Mac OS X. $k$-mers are a common abstraction in DNA sequence analysis that enable alignment-free sequence analysis and comparison. However the dramatic increase in sequence data generation has led to the development of data structures ad algorithms to discover possible improvements to $k$-mer-based approaches. The khmer project provides several efficient data structures and algorithms to analyze short-read nucleotide sequencing data while emphasizing online analysis, low memory data structures and streaming algorithms.

## 6.5 The economics of software correctness

According to MacIver [7], it is almost impossible for someone to write a significant piece of correct software. It is rea-

sonable to write bug-free small libraries, but the chances to write a whole perfect program are almost zero. The problem is not that people do not know how to write correct software, it is that correct software is too expensive. Being too expensive means that nobody would be willing to pay a price for the software, or a competing product could be released much earlier. The result of this is shipping buggy software to users. The problem is: bugs found by users are more expensive than bugs found before a user sees them, since in the former case the result could be lost users, lost time and etc. Also, fixing a production bug needs more analysis since there can not be extended down time for production system. Hypothesis is a good example of reducing the effort of finding bugs.

### 6.6 Automatic Predicate Abstraction of C Programs

Model checking has been proven very efficient in testing hardware and protocol domains. Ball et al. [1] The increase of interest in the implementation of model checking in software has led to the development of C2BP, a tool that combines predicate abstraction, model checking, symbolic reasoning, and iterative refinement to test a program. C2PB is able to perform predicate abstraction of C programs automatically. It produces boolean output to the BEBOP model checker to compute the set of reachable states. C2BP is part of the SLAM tool kit, which goal is to check a programâĂŹs temporal safety properties automatically.

## 7. CONCLUSION

Property-based testing is an interesting alternative to unit testing, and the Hypothesis library approach, where property-based tests are written like unit testing and explore Python syntax to do input generation, is a good fit for research software, which usually lacks formal specification. We explored how to apply this idea to *khmer*, a library for biological sequence data analysis, and found bugs, inconsistent behavior and uncovered test cases with just a few tests. Finally, we created some input generation strategies that can be used by other projects in the same area, and hopefully our efforts can also be useful for other scientific areas.

## 8. REFERENCES


[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. *Acm sigplan notices*, 36(5):203–213, 2001. URL `http://web.cs.ucla.edu/~todd/research/pldi01.pdf`.

[2] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011. URL `http://dl.acm.org/citation.cfm?id=1988046`.

[3] G. Cormode. Count-min sketch. *Encyclopedia of Algorithms*, pages 1–6, 2008. URL `http://dimacs.rutgers.edu/~graham/pubs/papers/cmencyc.pdf`.

[4] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, Apr. 2005. URL `http://dl.acm.org/citation.cfm?id=1073718`.

[5] M. R. Crusoe, H. F. Alameldin, S. Awad, E. Boucher, A. Caldwell, R. Cartwright, A. Charbonneau, B. Constantinides, G. Edvenson, S. Fay, J. Fenton, T. Fenzl, J. Fish, L. Garcia-Gutierrez, P. Garland, J. Gluck, I. GonzÃąlez, S. Guermond, J. Guo, A. Gupta, J. R. Herr, A. Howe, A. Hyer, A. HÃd'rpfer, L. Irber, R. Kidd, D. Lin, J. Lippi, T. Mansour, P. McA'Nulty, E. McDonald, J. Mizzi, K. D. Murray, J. R. Nahum, K. Nanlohy, A. J. Nederbragt, H. Ortiz-Zuazaga, J. Ory, J. Pell, C. Pepe-Ranney, Z. N. Russ, E. Schwarz, C. Scott, J. Seaman, S. Sievert, J. Simpson, C. T. Skennerton, J. Spencer, R. Srinivasan, D. Standage, J. A. Stapleton, S. R. Steinman, J. Stein, B. Taylor, W. Trimble, H. L. Wiencko, M. Wright, B. Wyss, Q. Zhang, e. zyme, and C. T. Brown. The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Research*, Sept. 2015. doi: 10.12688/f1000research.6924.1. URL `http://f1000research.com/articles/4-900/v1`.

[6] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *2007 Conference on Analysis of Algorithms*, pages 127–146, 2007. URL `http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf`.

[7] D. R. MacIver. The economics of software correctness, May 2015. URL `http://www.drmaciver.com/2015/10/the-economics-of-software-correctness/`.

[8] D. R. MacIver. Welcome to Hypothesis! âĂŤ Hypothesis 1.14.0 documentation, Nov. 2015. URL `https://hypothesis.readthedocs.org/en/latest/`.

[9] B. Mill. Bloom filters by example, July 2015. URL `http://billmill.org/bloomfilter-tutorial/`.

[10] L. Qi, J. Wen, H. Huang, X. Wang, and Z. Wu. The research on the fuzzing. In *Proceedings of 2013 Chinese Intelligent Automation Conference*, pages 85–91. Springer, 2013.