Mutation Testing in Patterns Documentation

Release 1.0

Alexander Todorov

Aug 18, 2016

Contents

Mutation testing is a technique used to evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways, for example replacing `True` constants with `False` and re-running its test suite. When the test suite fails the *mutant* is *killed*. This tells us how good the test suite is. The goal of this paper is to describe different software and testing patterns related using practical examples.

Some of them are language specific so please see the relevant sections for information about installing and running the necessary tools and examples.

CHAPTER 1

Make sure your tools work

Mutation testing relies on dynamically modifying program modules and loading the mutated instance from memory. Depending on the language specifics there may be several ways to refer to the same module. In Python the following are equivalent

```
import sandwich.ham.ham
obj = sandwich.ham.ham.SomeClass()

from sandwich.ham import ham
obj = ham.SomeClass()
```

Note: The equivalency here is in terms of having access to the same module API.

When we mutation test the right-most `ham` module our tools may not be able to resolve to the same module if various importing styles are used. For example see python/example_00/README.

Another possible issue is with programs that load modules dynamically or change the module search path at runtime. Depending on how the mutation testing tool works these operations may interfere with it. For example see python/example_01/README.

TL,DR: explore your test tool first and manually verify the results before going further. Unless you know the tools don't trust them!

CHAPTER 2

Make sure your tests work

Mutation testing relies on the fact that your test suite will fail when a mutation is introduced. In turn any kind of failure will kill the mutant! The mutation test tool has no way of knowing whether your test suite failed because the mutant tripped one of the assertions or whether it failed due to other reasons.

For example see python/example_02/README

TL,DR: make sure your test suite is robust and doesn't randomly fail due to external factors!

CHAPTER 3

Divide and conquer

The basic mutation test algorithm is this

```
for operator in mutation-operators:
    for site in operator.sites(code):
        operator.mutate(site)
        run_tests()
```

- **mutation-operators** are the things that make small changes to your code
- **operator.sites** are the places in your code where this operator can be applied

As you can see mutation testing is a very expensive operation. For example the pykickstart project started with 5523 possible mutations and 347 tests, which took on average 100 seconds to execute. A full mutation testing execution needs more than 6 days to complete!

In practice however not all tests are related to, or even make use of all program modules. This means that mutated operators are only tested via subset of the entire test suite. This fact can be used to reduce execution time by scheduling mutation tests against each individual file/module using only the tests which are related to it. The best case scenario is when your source file names map directly to test file names.

For example something like this

```
for f in `find ./src -type f -name "*.py" | sort`; do
    TEST_NAME="tests/$f"
    runTests $f $TEST_NAME
done
```

Where **runTests** executes the mutation testing tool against a single file and executes only the test which is related to this file. For *pykickstart* this approach reduced the entire execution time to little over 6 hours!

TL,DR: Good source code and test organization will allow easy division of test runs and tremendously speed up your mutation testing execution time!

CHAPTER 4

Fail fast

Mutation testing relies on your test suite failing when it detects a faulty mutation. It doesn't matter which particular test has failed because most of the tools have no way of telling whether or not the failed test is related to the mutated code. That means it also doesn't matter if there are more than one failing tests so you can use this to your advantage.

TL,DR: Whenever your test tools and framework support the **fail fast** option make use of it to reduce test execution time even more!

CHAPTER 5

Python: Refactor if string != “”

Comparison operators may be mutated with each other which gives, depending on the langauge about 10 possible mutations.

Every time `str` is not an empty string the following 3 variants are evaluated to `True`:

- `if str != ""`
- `if str > ""`
- `if str not in ""`

The existing test cases pass and these mutations are never killed. Refactoring this to

```
if str:
    do_something()
```

is the best way to go about it. This also reduces the total number of possible mutations.

For example see python/example_03/README

TL,DR: Refactor `if str != "":` to `if str:`!

CHAPTER 6

Appendix. Mutation testing with Python

Cosmic-Ray is the mutation testing tool for Python. It is recommended that you install the latest version from git:

```
pip install https://github.com/sixty-north/cosmic-ray/zipball/master
```

Cosmic-Ray uses *Celery* to allow concurrent execution of workers (e.g. mutation test jobs). To start the worker

```
cd myproject/
celery -A cosmic_ray.tasks.worker worker
```

To execute a test job (called session) use a different terminal and

```
cd myproject/
cosmic-ray run --baseline=10 session_name.json some/module.py -- tests/some/test.py
```

Note: Test runner and additional test parameters can be specified. Refer to Cosmic-Ray's documentation for more details!

To view the mutation results execute

```
cosmic-ray report session_name.json
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search