Reviewing Automatic Test Generation Tools (Auger, Deal, Klara, Pynguin)

Evelyn Kuo
https://github.com/enkokoro/program_analysis_project

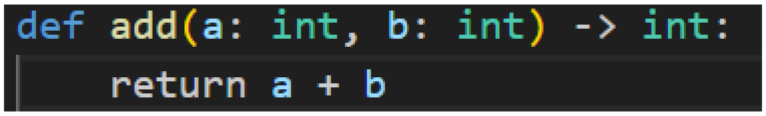
Overview

In this project, I reviewed 4 automatic test generation tools which included Auger (Laffra), Deal (“Deal”), Klara (“Klara”) and Pynguin (“pynguin”) to better understand the existing open-source program analysis tools. All 4 tools were tools for Python as the language is very popular and is interesting from a program analysis standpoint as it is a dynamically typed language, though Python does have typing capabilities. When reviewing the tools, I wanted to get a good understanding of how the tools may be useful, namely their strengths and potential weaknesses. In particular, I wanted to evaluate the applicability of the tool in terms of ease of use with respect to how much extra work is needed to start producing automatic tests and with respect to how easy it is to tell what test case is failing in order to debug, as well as how good the tool is at catching bugs. I wanted to evaluate automatic test generation tools as opposed to other program analysis tools because of importance in testing in ensuring code quality. While formal verification and proofs are necessary to prove program correctness, often in industry and in practice, it is costly to verify programs and is usually only done for critical software, like airplane control systems. Instead, most software engineers employ the use of testing to give reasonable evidence that the software performs as expected. While there are methodologies like test driven development in which test cases are written ahead of the actual code and are used to guide development, it is very common for writing test cases to be a task that is procrastinated as it is considered cumbersome. Hence, automated test generation seems to be an attractive solution, but how good are automated test generation software in actuality?

Process

For evaluation purposes, I only focused on 4 automated testing tools: Auger, Deal, Klara, and Pynguin. To start off with, I created several python file examples in the examples folder (https://github.com/enkokoro/program_analysis_project/tree/master/examples) following the folder structure below.

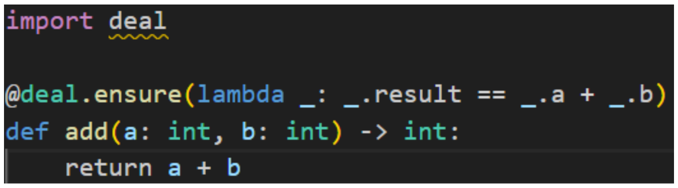
```
project/
|   README.md
└───examples/
|   └───module/
|       |   module.py: base version of file that you want to generate test cases for
|       |   module_types.py: type annotated version
|       |   module_deal.py: deal annotated version
|       └───tests/
|           └── (tool)/: subfolders corresponding to each tool
|                 |      test_module.py: generated or created test files from the tool it belongs to
```
In addition to the base python file, as many of the tools share the need for type annotation for better performance, I created a version of the file which is type annotated.



```python
def add(a: int, b: int) -> int:
    return a + b
```

Type annotated version of add function

Furthermore, Deal was a tool which allowed for more extensive annotations of contracts (preconditions and postconditions), so a deal specific file was also included in the folder.



```python
import deal

@deal.ensure(lambda _: _.result == _.a + _.b)
def add(a: int, b: int) -> int:
    return a + b
```

Deal annotated version of add function

I then created a script which, given a module name, automatically generates a test file for each of the tools. Auger and Deal were the only tools which required additional file setups to create the test file, which will be discussed in more detail in their respective sections. Pynguin and Klara both had command line tools for generating a test file. All tools generated a test file which could be run using pytest. For more detailed, step-by-step instructions on creating test files for a new module, see the Github README. Then, for each tool, I evaluated them by number of extra characters to generate tests and quality of test cases generated. I collected examples from the tool pages to evaluate the quality of test case generation. All tool pages except Klara only had examples where their tool worked. Klara, on the other hand, had a page of limitations of the tool, from which I used as an example. I wasn't able to find a suitable test benchmark for evaluation of how good the tool could catch bugs, so I created a simple benchmark based on bugs for a function which adds 2 numbers and evaluated the tools on the created bugs.

Tool Descriptions

Auger

Github: https://github.com/laffra/auger (Laffra)
Blog: http://chrislaffra.blogspot.com/2016/12/auger-automatic-unit-test-generation.html (Laffra)

Auger is a test generation tool which utilizes the function calls that are already present in your code. Normally, when a programmer writes code, the main function will encapsulate all the calls and "do" what the programmer wants. Auger takes advantage of all the calls the programmer writes and essentially tracks the inputs and outputs of function calls to be used as test cases for the code. Hence, this tool in theory requires minimal extra setup (40 characters counting the tab and "module" as the name of the python module in question) of changing the main call of

```
if __name__ == '__main__':
    main()
```

to

```
import auger
if __name__ == '__main__':
    with auger.magic([module]):
        main()
```

This type of unit test generation is useful for regression testing, however fares poorly against random functions and has the drawback of "if the code contains a bug, Auger will simply record the buggy behavior" (Laffra). In the project, since the python example files only had a function, I created a dummy auger_setup.py file, which called the desired test function with programmer inputs. In order to create the auger test file, you just need to run the python file with the main function annotated with the auger.magic. For more information, see the README.

Deal

Github: https://github.com/life4/deal ("Design by contract for Python. Write bug-free code. Add a few decorators, get static analysis and tests for free.")
Documentation: https://deal.readthedocs.io/basic/intro.html ("Deal")

Deal is a test generation tool which relies on contract annotations. Notable contracts include preconditions which are requirements on the inputs to a function as well as postconditions which are requirements on the outputs of a function. Deal has other functionality aside from automatic test generation, but I did not focus on exploring those functionality. To function well in test generation, python modules need to be annotated with preconditions and postconditions. Examples for what deal contracts look like can be found on the deal website ("Deal") and in the github under files with the suffix _deal.py. After annotating a file, the following is sufficient to have Deal generate and run tests.

```
@deal.cases(module.function)
def test_function(case: deal.TestCase) -> None:
    case()
```

Note that Deal doesn't explicitly list out test cases, but instead runs them in pytest, though in the event of test case failure, pytest will output the arguments and result which are useful for debugging. In tests, given strong contracts, Deal does well in detecting the add bugs, however being able to write strong contracts in practice which are necessary to do well in catching bugs may be hard. Contracts are more applicable in the case of one-way functions where computing the result of the function is difficult, but verifying the result is easy. A common example of such is factoring, where it is hard to generate factors for a number, but easy to verify if a group of numbers are factors to a number (just multiply and compare). However, usually it may be hard to come up with a function, because it may reduce to needing an oracle for the function you are trying to test in the first place. Essentially Deal trades effort in contract annotations for strong tests. For more information on running Deal tests, see the README.

Klara

Github: https://github.com/usagitoneko97/klara (Ho)
Documentation: https://klara-py.readthedocs.io/en/latest/ ("Klara")
Limitations Page: https://klara-py.readthedocs.io/en/latest/limitation.html ("Klara Limitation")
Klara is an automatic test generation tool which generates test by aiming to have full coverage of all return values for the function in question. It uses Z3 to generate constraints for each branch of return and find the necessary parameters to enter that branch. It is also possible to configure Klara with custom coverage metrics, one of which may be coverage of the actual function, however this analysis focused on its default test generation. Klara requires type annotation and is a CLI tool. A few of Klara's limitations are outlined in their limitations page. It is similar to Auger in that its main strength would be for regression testing because it takes the outputs of the function as ground truth. A benefit to Klara is that it uses ASTs and Z3 to generate all test examples, so it doesn't actually run the file in question when generating test cases which can be good from a security perspective, however the benefit disappears when you need to run the actual test file. For more information on running Klara tests, see the README.

Pynguin

Github: https://github.com/se2p/pynguin (Lukasczyk)
Documentation: https://pynguin.readthedocs.io/en/latest/user/quickstart.html ("pynguin")
Paper: https://dl.acm.org/doi/10.1145/3510454.3516829 (Lukasczyk and Gordon)
Pynguin is an automatic test generation tool which generates test by aiming to have full code coverage using concolic testing. Furthermore, it generates mutants to the original source code and creates test cases which aim to differentiate the mutants from the original. Similar to Auger and Klara, the tool "generates assertions that interpret the returned values on the original module as the ground truth," so it is best suited for regression testing as opposed to testing to find bugs. Like Klara, Pynguin is a CLI tool. The tool page claims it does best when given type annotations, however the tool can work, but to a lesser degree on non-annotated python files. However, when I used the tool, I found that regardless of if I gave type-annotated python files, the tool would generate test files which neglected the type of the input parameters. For example, Pynguin would generate test cases where the input to a function would be boolean values or sometimes other objects like sets when the function was type annotated to accept

integers. I have not figured out if this is a configuration issue on my side as I tested on the same triangle example as the documentation, and the documentation has inputs which align with the type annotations, however the test cases on my local machine appear to ignore the type annotations when generating Pynguin test cases. For more information on running Pynguin tests, see the README.

Interesting Examples

The following are a subset of the examples the 4 tools were evaluated on which include the add tests where I made several buggy implementations and checked to see how the tools behaved on the buggy implementations; choice where the function uses a random call; and sneaky_div which was an example that Klara listed as a limitation. For more results for the other examples, see the examples folder on Github where for a desired module, looking into the tests folder and then the desired tool will yield an output.txt file which is the pytest output.

Add

In total there are 4 directories in the examples folder corresponding to add: add, add_bug1, add_bug2, add_bug3. The function in question is computing addition between two integers, and have 3 buggy versions. Bug1 is where + is substituted for -, bug2 adds an additional case which returns the wrong answer for when the two inputs are equal which tests for if the tools have proper coverage of if/else branches, and bug3 has a special failure case for when the second input is 0 and tests for if the tools can detect and cover special cases of divide by 0. The following table contains the comparison of how many characters are needed to use each of the tools.

Number of Extra Characters Needed Per Tool

Auger	Deal	Klara	Pynguin
40 (auger import and auger.magic)	69 (type and deal annotation)	13 (type annotation)	13 (type annotation)

The following table contains the results on if the tool is able to detect the bug if given the bug as the original source code (blanks indicate that no bug was detected). If we look at it from a bug finding perspective, only Deal seems viable in catching bugs, however it is also only for the case where the function is simple enough to have a good post condition.

Bug Detection Per Tool

	bug1	bug2	bug3
Auger			

Deal	{'a': 0, 'b': 1, 'result': -1}	{'a': 1, 'b': 1, 'result': 0}	a = 0, b = 0 ZeroDivisionError
Klara			
Pynguin			

If we look from a regression testing standpoint, we can compare whether the bugs will be caught using the test generated from the bug-free version. The Auger test will be able to catch the bug1, because the input happens to be one where the outputs between the bug-less version and bug1 differ, however for bug2 and bug3, the outputs happen to be the same. The Deal test will be able to catch all of the bugs as the test file between the buggy and non-buggy versions look the same. The Klara test happens to be one where the outputs match for bug1 and bug2 match, but for bug3 it catches the bug. The Pynguin test also passes all of the bugs.

Choice

Choice was an interesting example as the function involves taking a random choice from a list of items. As expected, Auger thought the code was buggy because it records the output of the last occurrence as the answer. Deal performed fine, though it is to be noted that this example was taken from the Deal documentation page. Klara produced no test cases, I believe due to it's lack of support for imports. Lastly, Pynguin's test case runs ended up with type errors due to not conforming to the type requirements of the function. This example demonstrated the weakness of test generation tools which take the code output as ground truth when the code is non-deterministic.

Number of Extra Characters Needed Per Tool

Auger	Deal	Klara	Pynguin
63 (auger import and auger.magic)	187 (type and deal annotation)	15 (type annotation)	15 (type annotation)

Results Summary

Overall, in terms of ease of use, Pynguin and Klara required the least amount of extra annotation of the original python module, only requiring type annotations of function inputs and outputs. Auger also required relatively little amounts of extra annotation, only requiring wrapping the main function call in an auger.magic environment to track the function calls' arguments and outputs. Lastly, Deal required the most amount of extra annotation, with needing contract annotations for best results. In terms of quality of testing, Deal was the clear winner, though it often came down to if the contracts written were sufficiently strong enough. Auger, Klara, and Pynguin had the issue of taking the code as ground truth, though Deal is similar in that it takes the contracts as ground truth. Auger furthermore is less powerful in that it only records the function calls that the programmer performs. Klara and Pynguin on the other hand aimed to create test cases to improve code coverage.

Conclusion

In this project, I investigated 4 different automatic test generation tools which include Auger, Deal, Klara and Pynguin. In addition to comparing their test generation qualitatively by reading up on the documentation, I also compared the tools using several examples taken from their documentation pages. I compared the number of extra characters needed to use the tools as a proxy for ease of use and learning/integration curve. I also created several small buggy examples to check the strength of the test cases generated by the tools. In conclusion, these tools show that there is no free lunch in automatic test generation, as it was the case that you had to choose between better quality testing or the cost of extra annotation and cannot choose both. While this evaluation did not show promising results for me in terms of using these tools for the future, it was a good learning experience to see how far testing has come along and to understand the strengths and weaknesses of the tools.

References

"Deal." *Deal*, 30 July 2018, https://deal.readthedocs.io/basic/intro.html. Accessed 8 May 2023.

"Design by contract for Python. Write bug-free code. Add a few decorators, get static analysis and tests for free." *GitHub*, https://github.com/life4/deal. Accessed 8 May 2023.

Ho, Guo Xian. "Automatic test case generation for python and static analysis library." *GitHub*, 2021, https://github.com/usagitoneko97/klara. Accessed 8 May 2023.

"Klara." *klara 0.6.3 documentation*, 2021, https://klara-py.readthedocs.io/en/latest/. Accessed 8 May 2023.

"Klara Limitation." *klara 0.6.3 documentation*, 30 July 2018, https://klara-py.readthedocs.io/en/latest/limitation.html. Accessed 8 May 2023.

Laffra, Chris. "Auger - Automatic Unit Test Generation for Python." *The Flying Dutchman*, 19 December 2016, http://chrislaffra.blogspot.com/2016/12/auger-automatic-unit-test-generation.html. Accessed 8 May 2023.

Laffra, Chris. "Automated Unittest Generation for Python." *GitHub*, 2019, https://github.com/laffra/auger. Accessed 8 May 2023.

Lukasczyk, Stephan. "The PYthoN General UnIt Test geNerator is a test-generation tool for Python." *GitHub*, 2023, https://github.com/se2p/pynguin. Accessed 8 May 2023.

Lukasczyk, Stephan, and Fraser Gordon. "Pynguin: Automated Unit Test Generation for Python." *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172. *10.1145/3510454.3516829*, https://doi.org/10.1145/3510454.3516829.

"pynguin." *Pynguin—PYthoN General UnIt test geNerator*, 2023, https://pynguin.readthedocs.io/en/latest/user/quickstart.html. Accessed 8 May 2023.

Results
Project Description

- General gist
    - For each tool have working example and nonworking example to show limitation
    - benchmark
- Tools descriptions
    - Pynguin
        - https://link.springer.com/article/10.1007/s10664-022-10248-w
        - Tool paper https://dl.acm.org/doi/10.1145/3510454.3516829
        - Goal of tool is to generate tests which have 100% coverage
        - “Generates assertions that interpret the returned values on the original module as the ground truth”
        - Mutation testing, generates mutants of original source code and yields assertions which can distinguish mutants
        - Useful for regression testing?
        - Hypothesis
            - Pros: if know code is already correct, detect mutant changes, cover all paths
            - Cons: if existing code is buggy, executes code - concolic
    - Klara
        - https://klara-py.readthedocs.io/en/latest/
        - “ cover all return values of the input file’s functions”
        - Uses z3 to generate constraints satisfying branch for return
        - “generate test case for all possible return values of the function, instead of generate test case for all control path of the function.”
        - Hypothesis
            - Pro: doesn’t execute code works on ast level
            - Con: non z3 stuff
                - https://klara-py.readthedocs.io/en/latest/limitation.html
                - `def foo(v1: int, v2: float):`
                - `    if v1 > 10000:`
                - `        s = v1 / 0  # unused statement`
                - `    if v1 > v2:`
                - `        s = v1`
                - `    else:`
                - `        s = v2`
                - `    return s`
    - Deal
        - Contract based
            - Runtime. Call the functions, do usual tests, just play around with the application, deploy it to staging, and Deal will check contracts in runtime. Of course, you can disable contracts on production.
            - Tests. Deal is easily integrates with PyTest or any other testing framework. It does property-based testing for functions with

        contracts. Also, deal has test CLI command to find and run all pure functions in the project.
        - Linter. The most amazing part of Deal. It statically checks constant values in the code, does values inference, contracts partial execution, propagates exceptions and side-effects. Deal has lint CLI command for it and flake8 integration.
        - Experimental: Formal verification. The most powerful but limited idea in the whole project. Deal can turn your code into mathematical expressions and verify its correctness.
            - https://deal.readthedocs.io/basic/verification.html
            - Cool
- Auger
    - http://chrislaffra.blogspot.com/2016/12/auger-automatic-unit-test-generation.html
    - Similar to pynguin
    - Hypothesis
        - Pro
        - Con
            - “If the code contains a bug, Auger will simply record the buggy behavior.”

Results

- Coverage but its a bit unfair
- Mutations
- Collection of bugs and see what will be caught