


Software Testing, Quality Assurance & Maintenance—Lecture 10

Patrick Lam
University of Waterloo

February 6, 2026



Part I

Fuzzing Configurations

Not Just Inputs

So far: create program inputs through fuzzing
(mutation, generation).

Today: instead of inputs, consider program
configurations.

Command-line Options as Configurations

```
$ autopep8 --help
```

```
usage: autopep8 [-h] [--version] [-v] [-d] [-i]
               [--global-config filename]
               [--ignore-local-config] [-r] [-j n]
               [-p n] [-a] [--experimental]
               [--exclude globs] [--list-fixes]
               [--ignore errors] [--select errors]
               [--max-line-length n]
               [--line-range line line]
               [--hang-closing]
               [--exit-code] [files ...]
```

...

could also fuzz config files, registries, etc.

Three Takeaways

- ① configurations affect program behaviour;
- ② you can automatically construct grammars for configurations;
- ③ these grammars can be used for fuzzing.

Idea 1: Configurations are Important

```
$ autopep8 --help
...
    --experimental          enable experimental fixes
...
```

Clearly this option affects which code paths are reachable.

Some code paths are only reachable under certain configuration options.

argparse

Options in Python: getopt, optparse, argparse, ...

```
def process_numbers():
    parser = argparse.ArgumentParser
        (description='Process some integers.')
    parser.add_argument('integers', metavar='N', type=
                        int, nargs='+')
    group = parser.add_mutually_exclusive_group(
        required=True)
    group.add_argument('--sum', dest='accumulate',
        action='store_const',
        const=sum)
    group.add_argument('--min', dest='accumulate',
        action='store_const',
        const=min) # [--max omitted]

    args = parser.parse_args()
    print(args.accumulate(args.integers))
```

Processing numbers

```
$ python3 process-numbers.py --sum 2 4  
6
```

A low-angle photograph of the Auckland Grammar School building, a large, light-colored stone structure with multiple windows and a crest on the roof. The building is slightly out of focus, serving as a background for the title text.

A Grammar for Configurations

```
PROCESS_NUMBERS_EBNF_GRAMMAR: Grammar = {
    "<start>": ["<operator> <integers>"],
    "<operator>": ["--sum", "--min", "--max"],
    "<integers>": ["<integer>", "<integers> <integer>"],
    "<integer>": ["<digit>+"],
    "<digit>": crange('0', '9')
}

assert is_valid_grammar(PROCESS_NUMBERS_EBNF_GRAMMAR)
PROCESS_NUMBERS_GRAMMAR = convert_ebnf_grammar(
    PROCESS_NUMBERS_EBNF_GRAMMAR)
```

On GrammarCoverageFuzzer

We have seen `GrammarFuzzer`.

We are not talking about
`GrammarCoverageFuzzer`,
but it ensures that all alternatives are covered.

(it is a drop-in replacement for
`GrammarFuzzer`).

Fuzzing process_numbers configurations

```
>>> f = GrammarFuzzer(PROCESS_NUMBERS_GRAMMAR, min_nonterminals
                        =10)
>>> for i in range(3):
    args = f.fuzz().split()
    print(args)
    process_numbers(args)
['--max', '9', '8', '8', '162', '559606', '07043719933614']
7043719933614
['--sum', '6', '7', '4', '90', '57', '9767']
9931
['--max', '6', '1', '6900', '3637']
6900
```

Can't we automate this?

We manually proposed a grammar for
`process_numbers`.

That works, but is extra work.

Is there a better way?

Insight 2: It's Already There

The program already instructs `argparse` about which arguments it'll accept.

Idea: Construct the grammar from the program.

***Fuzzing Book* approach: dynamic analysis**

Observe program's calls to `argparse` to reconstruct the grammar.

Notes:

- ① for Python, dynamic analysis probably easier than static analysis;
- ② our implementation only works for specifically `argparse`.

Another approach: use a domain-specific language for options, generate code & grammar from that.



Part II

Mining Configurations

Key idea, again

Use Python tracing infrastructure to
track calls to `argparse`,
recording parameters,
to construct the grammar.

Exploratory code

A tracing function that observes
add_argument calls:

```
def trace_options(frame, event, arg):  
    if event != "call":  
        return  
    method_name = frame.f_code.co_name  
    if method_name != "add_argument":  
        return  
    locals = frame.f_locals  
    print(locals['args'])
```

Exploring

Let's exercise our code.

```
>>> sys.settrace(trace_options)
>>> process_numbers(["--sum", "1", "2", "3"])
('-h', '--help')
('integers',)
('--sum',)
('--min',)
('--max',)
6
>>> sys.settrace(None)
```

We can indeed see the arguments being added.

Implementation highlights

The *Fuzzing Book* exhaustively presents OptionGrammarMiner's implementation. We won't.

```
class OptionGrammarMiner:
    def __init__(self, function: Callable, log: bool = False):
        self.function = function
        self.log = log

    def mine_ebnf_grammar(self):
        # ...
```

Usage:

- 1 create an OptionGrammarMiner with the function that calls argparse,
- 2 trigger it by calling its mine_ebnf_grammar() method.

mine_ebnf_grammar highlights

`mine_ebnf_grammar()` enables tracing
& calls provided `function`.

`function` runs until `parse_args()`
called.

mine_ebnf_grammar's tracer

Tracer watches calls to `add_argument`,
`add_mutually_exclusive`, and
`add_argument_group`.

For instance, `add_argument` may call

```
def add_str_rule(self):  
    self.grammar["<str>"] = ["<char>+"]  
    self.grammar["<char>"] = xrange(  
        string.digits  
        + string.ascii_letters  
        + string.punctuation)
```

mine_ebnf_grammar in action

```
>>> miner = OptionGrammarMiner(process_numbers, log=True)
>>> process_numbers_grammar = miner.mine_ebnf_grammar()
>>> print (process_numbers_grammar)
...
{'<start>': ['<group>(<option>)*<arguments>'],
 '<option>': ['-h', '--help'],
 '<arguments>': ['( <integers>)+'],
 '<int>': ['(-)?<digit>+'],
 '<digit>': ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'],
 '<integers>': ['<int>'],
 '<group>': ['--sum', '--min', '--max']}
```

A photograph of a dense, mossy forest. The scene is filled with thick, moss-covered tree trunks and hanging moss, creating a lush, green environment. The lighting is soft and diffused, typical of a forest interior.

Part III

Fuzzing Mined Grammars

Yes, we can...

... fuzz mined options grammars.

```
>>> grammar = convert_ebnf_grammar(process_numbers_grammar)
>>> assert is_valid_grammar(grammar)
>>> f = GrammarFuzzer(grammar)
>>> for i in range(10):
>>>     print(f.fuzz())
--sum -h 19
--max -09 4
--min -685 -8
--max 73 4731240
--max --help --help -h 0 0 -34
--min --help 57
--max -6820 8
--sum 96
--min 7 -76 -61
--max --help 56
```

Another example: autopep8

```
>>> autopep8_miner = OptionGrammarMiner(autopep8)
>>> autopep8_ebnf_grammar = autopep8_miner.mine_ebnf_grammar()
>>> print (autopep8_ebnf_grammar["<option>"])
[' -h', ' --help', ' --version', ' -v', ' --verbose', ' -d', '
--diff', ' -i', ' --in-place',
' --global-config <filename>',
' --ignore-local-config', ' -r'
, ' --recursive', ' -j <n>', '
--jobs <n>', ' -p <n>', ' --
pep8-passes <n>', ' -a', ' --
aggressive', ' --experimental',
' --exclude <globs>', ' --list
-fixes', ' --ignore <errors>',
' --select <errors>', ' --max-
line-length <n>', ' --line-
range <line> <line>', ' --range
<line> <line>', ' --indent-
size <int>', ' --hang-closing',
' --exit-code']
```

autopep8 extracted grammar

Extracts correct types for lines and files:

```
>>> print (autopep8_ebnf_grammar["<line>"])\n['<int>']\n>>> print (autopep8_ebnf_grammar["<arguments>"])\n['( <files>)*']\n>>> print (autopep8_ebnf_grammar["<files>"])\n['<str>']
```

Fuzzing extracted autopep8 grammar

```
>>> autopep8_grammar = convert_ebnf_grammar(  
                                autopep8_ebnf_grammar)  
>>> assert is_valid_grammar(autopep8_grammar)  
>>> f = GrammarFuzzer(autopep8_grammar, max_nonterminals=4)  
>>> for i in range(10):  
>>>     print(f.fuzz())  
foo.py  
--range 9 9 foo.py  
--diff --help foo.py  
foo.py  
--jobs -64621 foo.py  
foo.py  
foo.py  
--indent-size -8 --list-fixes foo.py  
foo.py  
foo.py
```

GrammarCoverageFuzzer would be much better, oh well.

Could run autopep8 with these inputs.

Not shown: yet more examples

With some machinery to run arbitrary Python programs (that use `argparse`), carry out configuration fuzzing for:

- `mypy` static type checker
- `notedown` Notebook to Markdown converter.

A close-up photograph of a woven basket filled with several ripe, yellow pears. The pears have a smooth, slightly textured skin and some have short stems. The basket is made of light-colored wood or wicker. The background is softly blurred, showing more pears and the basket's structure.

Part IV

Combinatorial Testing

Option interaction

GrammarCoverageFuzzer would cover all options.

But, options also interact.
Would be prudent to test pairs of options together.

All pairs

```
>>> autopep8_miner = OptionGrammarMiner(autopep8)
>>> autopep8_ebnf_grammar = autopep8_miner.mine_ebnf_grammar()
>>> option_list = autopep8_ebnf_grammar["<option>"]
>>> pairs = list(combinations(option_list, 2))
>>> print (len(pairs))
435
>>> print (pairs[:20])
[(' -h', ' --help'), (' -h', ' --version'), (' -h', ' -v'), (' -h', ' --verbose'), (' -h', ' -d'), (' -h', ' --diff'), (' -h', ' -i'), (' -h', ' --in-place'), (' -h', ' --global-config <filename>'), (' -h', ' --ignore-local-config'), (' -h', ' -r'), (' -h', ' --recursive'), (' -h', ' -j <n>'), (' -h', ' --jobs <n>'), (' -h', ' -p <n>'), (' -h', ' --pep8-passes <n>'), (' -h', ' -a'), (' -h', ' --aggressive'), (' -h', ' --experimental'), (' -h', ' --exclude <globs>')]
```

Pairs grammar

```
>>> def pairwise(option_list):  
    return [option_1 + option_2  
            for (option_1, option_2) in combinations(  
                option_list, 2)]  
  
>>> pairwise_autopep8_grammar=extend_grammar(autopep8_grammar)  
>>> pairwise_autopep8_grammar["<option>"] = pairwise(  
    autopep8_grammar["<option>"])  
>>> assert is_valid_grammar(pairwise_autopep8_grammar)  
  
>>> pairwise_autopep8_fuzzer = GrammarFuzzer(  
    pairwise_autopep8_grammar,  
    max_nonterminals=4)  
  
>>> for i in range(10):  
    print (pairwise_autopep8_fuzzer.fuzz())
```

FYZcX s

Y u C

=kD

-h --in-place }

C ap

Counting

For `autopep8`, there are 870 pairs.

`GrammarCoverageFuzzer` would be quite useful to reach all 870.

For `mypy`, there are 140 options and 28,000 options.

But this takes less than 3 hours at 1 run per second.

Generalization to Inputs

We've seen grammar inference and fuzzing for configurations.

Can do something similar for some inputs as well.

See *Fuzzing Book* under “Mining Input Grammars”, and also the paper by Bettscheider & Zeller.