Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 10 — February 6, 2026

Patrick Lam *version 1*

We've talked about fuzzing using mutation and using grammars. So far, we've been fuzzing program inputs. But we can also fuzz, for instance, configurations, and that's what we'll talk about today, following `https://www.fuzzingbook.org/html/ConfigurationFuzzer.html`.

There are three main takeaways from this lecture. (1) things influencing program execution isn't just inputs, it's also information from other sources, like configurations; (2) you can observe dynamically program behaviour to automatically construct grammars for configurations; (3) these grammars can be used for grammar fuzzing.

To expand on that: in the previous lecture, we showed that if we have a grammar, we can produce inputs based on that grammar. But, where do grammars come from? It is sometimes possible to mine grammars for inputs, but I don't think we'll go there this term. It is easier to mine configurations than inputs, and we'll do that.

Configuration Options

Command-line programs take configuration options. `autopep8` is a code reformatter and takes these options:

```
$ 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 ...]
```

The first idea in this lecture is that *because some code paths are only reachable under certain configuration options, it is important to also cover the program's configuration options when testing it.* We are going to consider command-line configuration options, though configuration information might also come from configuration files. Configuration languages can be notoriously complicated, e.g. `sendmail`, a UNIX mail transfer agent, has book-length descriptions of its configuration.

argparse in Python

There are many command-line parsing libraries. Python has at least `getopt`, `optparse`, and `argparse`. We'll talk about `argparse`, which is higher-level than the others that I mention.

In `code/L10/process-numbers.py` you can find this code:

```
1 def process_numbers():
```

```
    parser = argparse.ArgumentParser(description='Process some integers.')
    parser.add_argument('integers', metavar='N', type=int, nargs='+',
                        help='an integer for the accumulator')
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument('--sum', dest='accumulate', action='store_const',
                       const=sum,
                       help='sum the integers')
    group.add_argument('--min', dest='accumulate', action='store_const',
                       const=min,
                       help='compute the minimum')
    group.add_argument('--max', dest='accumulate', action='store_const',
                       const=max,
                       help='compute the maximum')

    if args is not None:
        args = parser.parse_args(args)
    else:
        args = parser.parse_args()
    print(args.accumulate(args.integers))
```

This takes a accumulator command, which can be `--sum`, `--min`, or `--max`. It also takes a list of integers, and applies the accumulator to the integers. So, for instance:

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

A Grammar for Configurations

Before we go and infer grammars, let's write a grammar for this program.

```
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)
```

We can print `PROCESS_NUMBERS_GRAMMAR` and get what we would expect.

I've chosen to not talk about the `GrammarCoverageFuzzer` from the *Fuzzing Book*, which ensures (on subsequent runs) that all productions are covered. We can use the `GrammarFuzzer` from last time, though.

```
f = GrammarFuzzer(PROCESS_NUMBERS_GRAMMAR, min_nonterminals=10)
for i in range(3):
    args = f.fuzz().split()
    print(args)
    process_numbers(args)
```

produces the following output:

```
['--max', '9', '8', '8', '162', '559606', '07043719933614']
7043719933614
['--sum', '6', '7', '4', '90', '57', '9767']
9931
['--max', '6', '1', '6900', '3637']
6900
```

Sure, this works, but can't we just extract the grammar from the program? The program already instructs `argparse` about the arguments it will accept, after all.

The *Fuzzing Book* uses a dynamic analysis approach: it observes the program's calls to `argparse` to reconstruct the grammar. Two comments about that.

- For Python, dynamic analysis is probably easier than static analysis, since Python has pretty complete built-in introspection abilities. Sometimes, static analysis can be easier than dynamic analysis, especially when actually running the full software is difficult.
- The approach here is tied to the use of `argparse`, and different argument parsing libraries would require different grammar mining techniques.

Another approach that I've seen is to autogenerate the calls to the argument parsing library from a declarative definition of the program's arguments. This complicates the build process, but it simplifies building a grammar for the arguments.

Mining Configuration Options

As I wrote above, the second idea is that *we can track calls to* `argparse` *and record the parameters to construct the grammar*. Recording continues until the call to `argparse.parse_args()` and construction happens after that.

We used Python's tracing infrastructure to collect coverage information before. We now use it to track calls to `add_argument` in `argparse`:

```
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'])
```

and we try it out:

```
>>> sys.settrace(trace_options)
>>> process_numbers(["--sum", "1", "2", "3"])
('-h', '--help')
('integers',)
('--sum',)
('--min',)
('--max',)
```

```
6
>>> sys.settrace(None)
```

We can observe the calls to `argparse.add_argument`, as well as the output to our call (6). These are the calls that are in our `process_numbers` implementation and consistent with the API documentation for `add_argument()`.

There is a lot of infrastructure from the *Fuzzing Book* which I won't include in the notes; once again, I'll try to include only the most important points. I've put all the code in `code/L10/option_grammar_miner.py`.

We define a `OptionGrammarMiner` class which takes a function to mine. This is the function that transitively constructs the `ArgumentParser` object; in our example, we would mine `process_numbers()`.

The key method on the miner is `mine_ebnf_grammar()`, which returns a mined grammar containing some number of options and some number of arguments:

```
<start> ::= <option>* <arguments>
<option> ::= <empty>
<arguments> ::= <empty>
```

The mining function enables tracing and calls the provided `function`, running it until a `ParseInterrupt` is thrown.

The tracer function throws a `ParseInterrupt` when `function` calls `parse_args()`, indicating that there are no more arguments to be added.

The tracer function also acts on calls to `argparse`'s `add_argument`, `add_mutually_exclusive_group`, and `add_argument_group` methods. There's a lot of implementation detail, which I don't think is productive to explain here. But I'll show one of the ways that the code adds something to the grammar, in the event that it observes a argument marked to be a string type (as specified in the call to `add_argument`).

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

We can run the code that I haven't shown you and, indeed, extract a grammar:

```
>>> 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']}
```

And, we can apply the third observation—that *this grammar can be used for fuzzing*:

```
>>> 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
```

Example: autopep8

The *Fuzzing Book* shows that we can apply our machinery on real Python applications, including `autopep8`, which re-styles Python code.

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

We can observe that our miner extracts the correct types for lines and files. And, of course, we can fuzz with this grammar. We set the arguments to a single file `foo.py`.

```
>>> 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
```

Using a `GrammarCoverageFuzzer` would be better, but we didn't explain that, and there is also a lot of implementation for that class.

You can actually run `autopep8` with the provided inputs as well, but we won't show that.

The *Fuzzing Book* also shows configuration fuzzing for the `mypy` static type checker and the `notedown` Notebook to Markdown converter.

Combinatorial Testing

Had we used a `GrammarCoverageFuzzer`, we'd visit each option at least once. But options also interact, and it would be prudent to test pairs of options together. There should be some memories of MATH 239 here.

We can import `combinations` from the Python `itertools` and then get all pairs of options:

```
>>> 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>')]
```

The *Fuzzing Book* asserts that pairs is usually enough to cover all interferences between options; three-option interactions are rare.

We can create a new grammar with pairs of options.

```
>>> 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
 -j -5 --list-fixes
 q ?
 --global-config w --ignore-local-config 0 L
```

There are 870 pairs to cover. The `GrammarCoverageFuzzer` would be quite useful here. Even if there are 140 options, as for `mypy`, then we would have 28,000 pairs of options, which is still less than three hours of testing if a run takes 1 second.

Generalization to Inputs

It is possible to use similar techniques to automatically derive input grammars based on input handling code. We don't do that, but you can find a description in the Fuzzing Book: `https://www.fuzzingbook.org/html/GrammarMiner.html`. There is also [BZ25], where Bettscheider and Zeller describe how they infer a grammar from recursive descent parser implementation code.

References

[BZ25] Leon Bettscheider and Andreas Zeller. Inferring input grammars from code with symbolic parsing. *ACM Trans. Softw. Eng. Methodol.*, November 2025. Just Accepted.