SLE Assignment 4

Test Data Generation

Nicolas Beck

Testing objectives

test parsing (1),

constraint checking (2),

simulation (3)

and Code Generation (4) ...

… of my fsml implementation

What sort of errors are you trying to catch?

Implementation errors in the modules for (1),(2),(3) and (4)

Testing objectives

What sort of generated test-data set helps in this context?

- randomly generated valid *.fsml + input files
- randomly generated invalid *.fsml + input files

How do you implement your test strategy?

- generate test data (see next slides)
- use data-driven testing approach (unittest Testsuite)

Test data generation

- Use nltk library (Natural Language Toolkit) to generate all possible combinations of CFG up to a certain depth

```
FSM -> ISTATE STATES
ISTATE -> 'initial' 'state' '#initState#' '{' TRANS '}'
STATES -> STATE STATES |
STATE -> 'state' '#stateDecl#' '{' TRANS '}'
TRANS -> TRANSITION TRANS |
TRANSITION -> '#input#' '/' '#action#' '->' '#newState#' ';'
```

Test data generation Step 1

- possible combination

```
initial state #initState# {
#input# / #action# -> #newState#
#input# / #action# -> #newState#
}

state #stateDecl# {
#input# / #action# -> #newState#
#input# / #action# -> #newState#
}

initial state #StateDecl# {
}
```

- can't be used by template library in this form

Test data generation Step 2

- template for use with jinja library

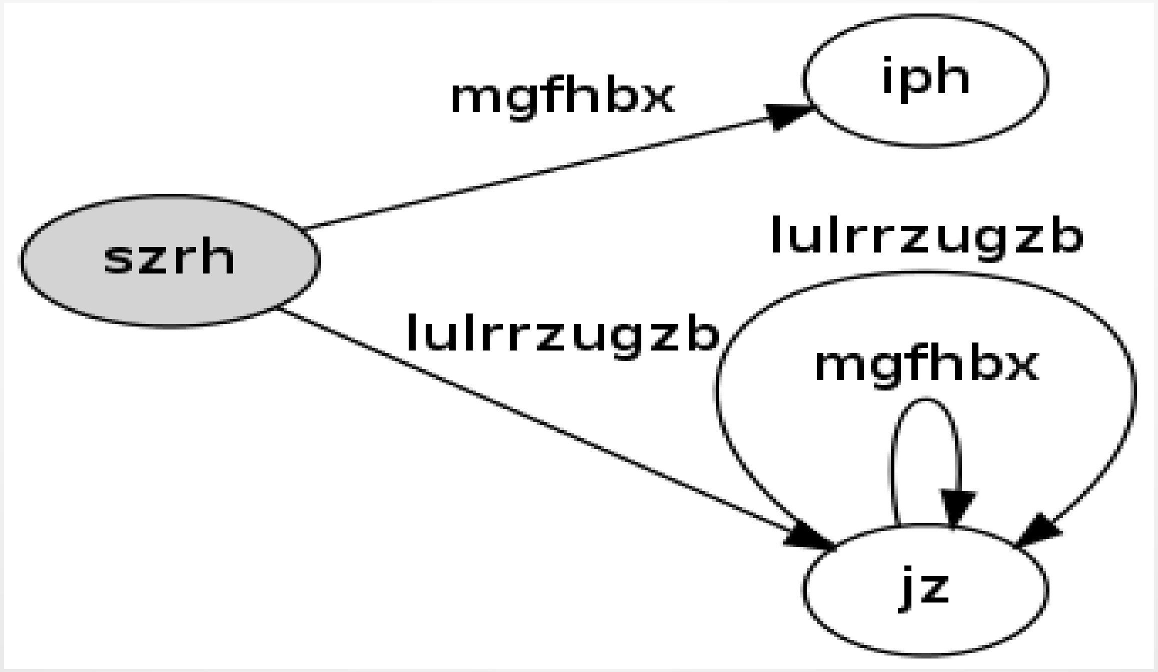
```
initial state {{ states[0] }} {
{{ states[0].input[0] }} / {{ states[0].action[0] }} -> {{ states[0].newstate[0] }}
{{ states[0].input[1] }} / {{ states[0].action[1] }} -> {{ states[0].newstate[1] }}
}

state {{ states[1] }} {
{{ states[1].input[0] }} / {{ states[1].action[0] }} -> {{ states[1].newstate[0] }}
{{ states[1].input[1] }} / {{ states[1].action[1] }} -> {{ states[1].newstate[1] }}
}

state {{ states[2] }}  {
}
```

- → next step: generate random graph with edges :

  2 , 2 , 0

Test data generation Step 3

- generate random (valid) graph with edges : 2 , 2 , 0
- own algorithm, result looks like :



Test data generation Step 4

- Fit the generated graph into the template

```
initial state szrh {
lulrrzugzb /  „“ -> jz
mgfhbx / „“ -> iph
}

state jz {
lulrrzugzb /  „“ -> jz
mgfhbx / „“ -> jz
}

state iph  {
}
```

Input generation

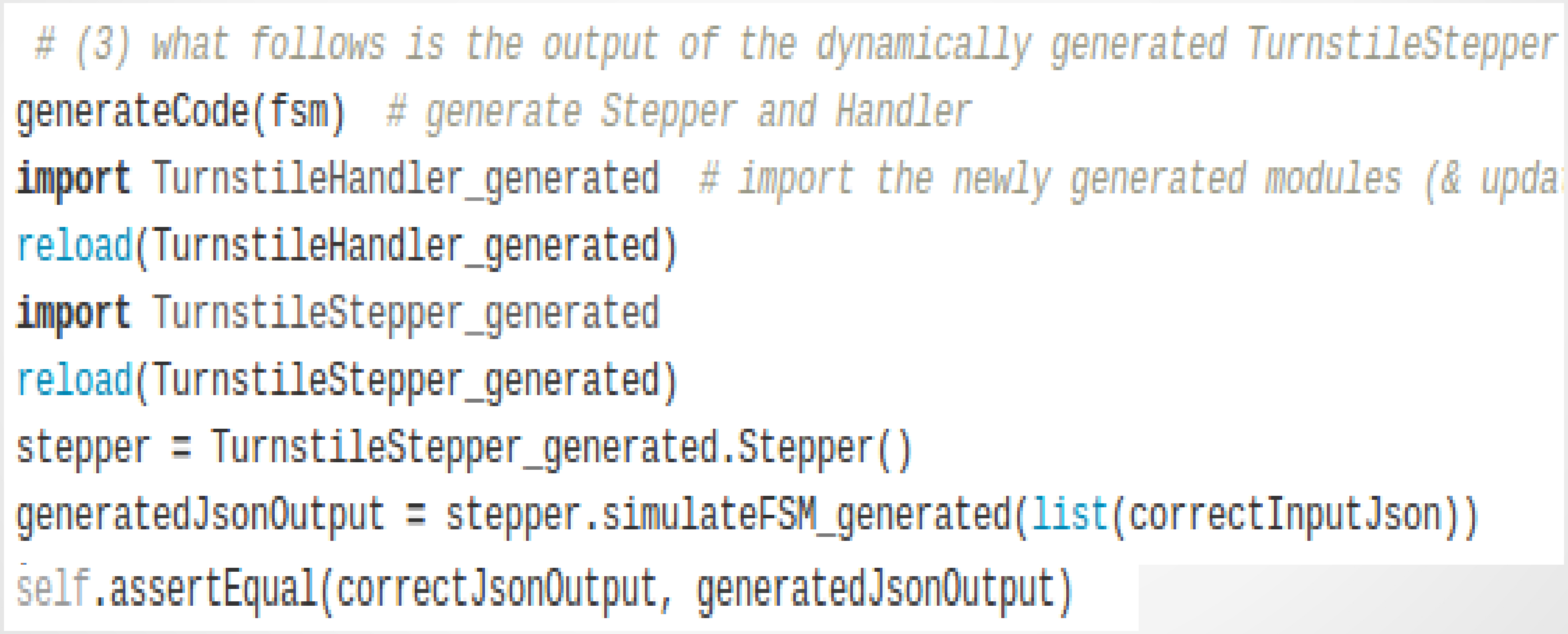
- The generated graph is also used to generate valid Input files (by randomly walking through the graph and dumping the output)
- This gold standard output is then compared to

  1) the implemented simulation of the fsml spec

  2) the simulation of the generated Code

  → all 3 have to be equal
- Python allows to import the generated modules dynamically

Input generation



```
 # (3) what follows is the output of the dynamically generated TurnstileStepper
generateCode(fsm)  # generate Stepper and Handler
import TurnstileHandler_generated  # import the newly generated modules (& upda
reload(TurnstileHandler_generated)
import TurnstileStepper_generated
reload(TurnstileStepper_generated)
stepper = TurnstileStepper_generated.Stepper()
generatedJsonOutput = stepper.simulateFSM_generated(list(correctInputJson))
self.assertEqual(correctJsonOutput, generatedJsonOutput)
```

Negative test cases

- For each possible error : syntax error, infeasible input, illegal input, duplicate ids, single initial, reachability, resolution, determinism
  invalid test data is generated (up to a certain depth)

  1) valid fsm is constructed like shown on prev slides

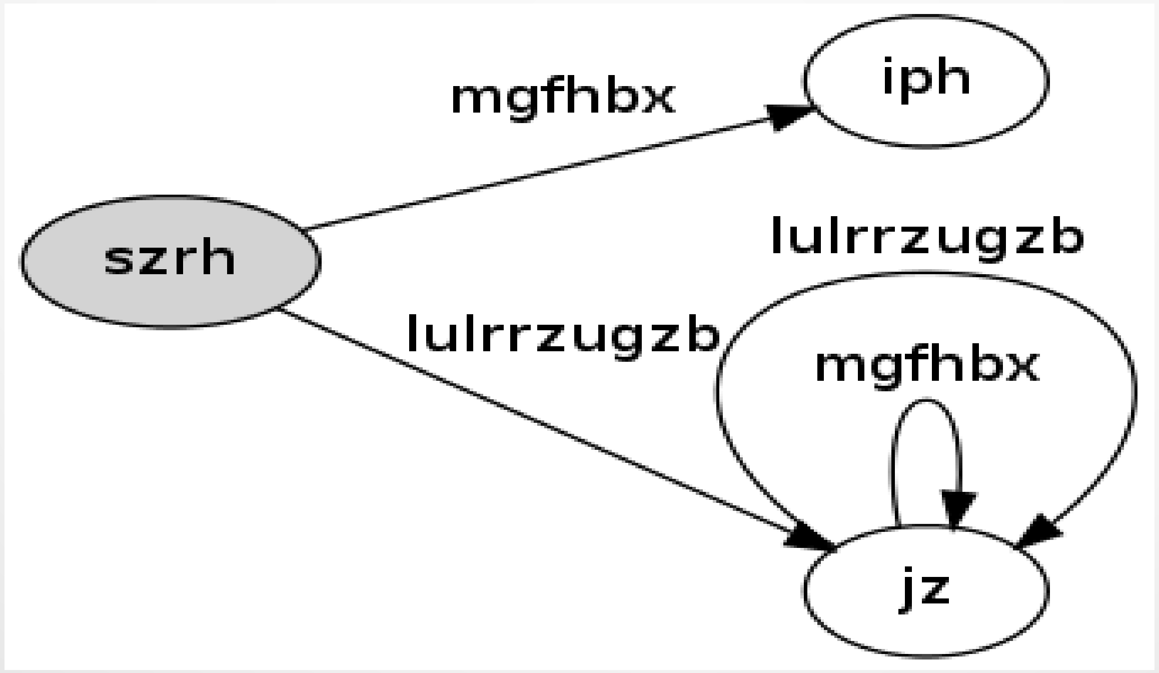
  2) add error at random position

  3) fit the invalid fsm into the (invalid?) template

single initial & parser errors are constructed by changing the template

Negative test cases

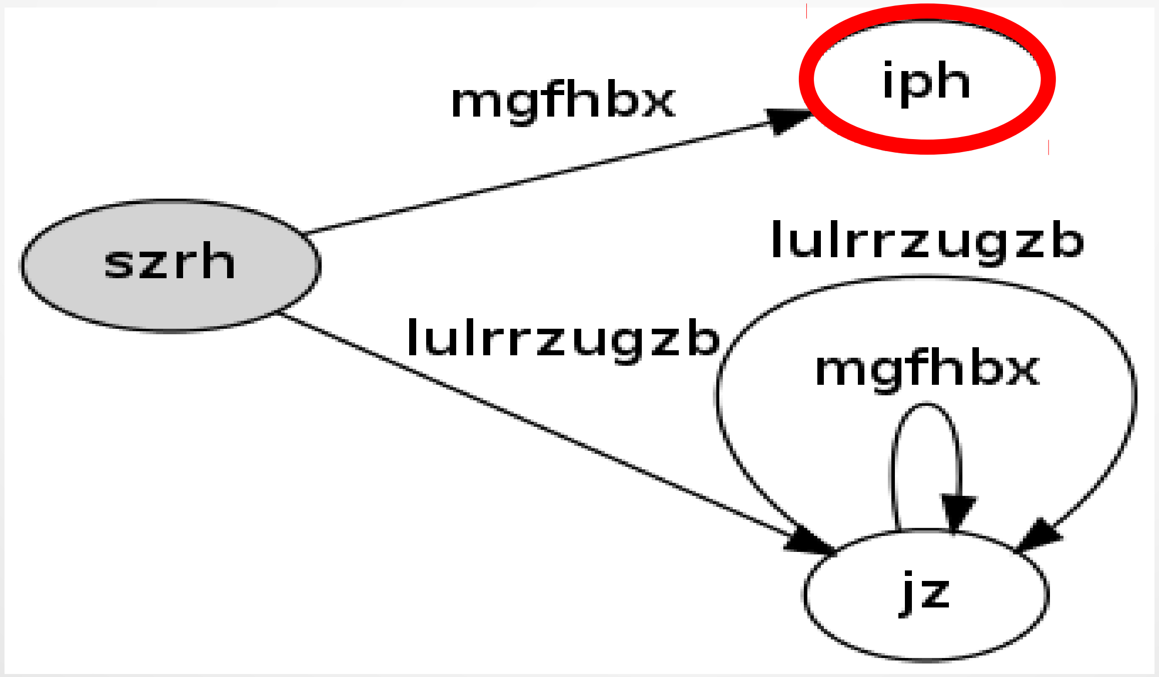
- Example : reachability error



Negative test cases

- Example : reachability error

  select random state

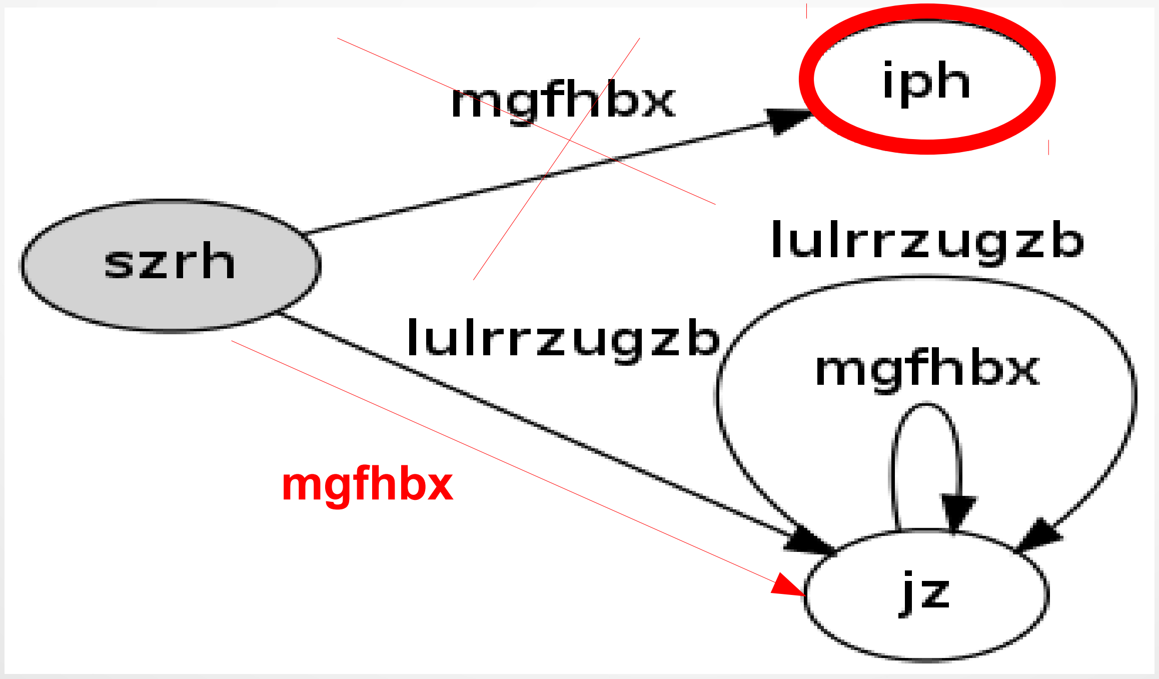


Negative test cases

- Example : reachability error

  select random state

  redirect all transitions to this state

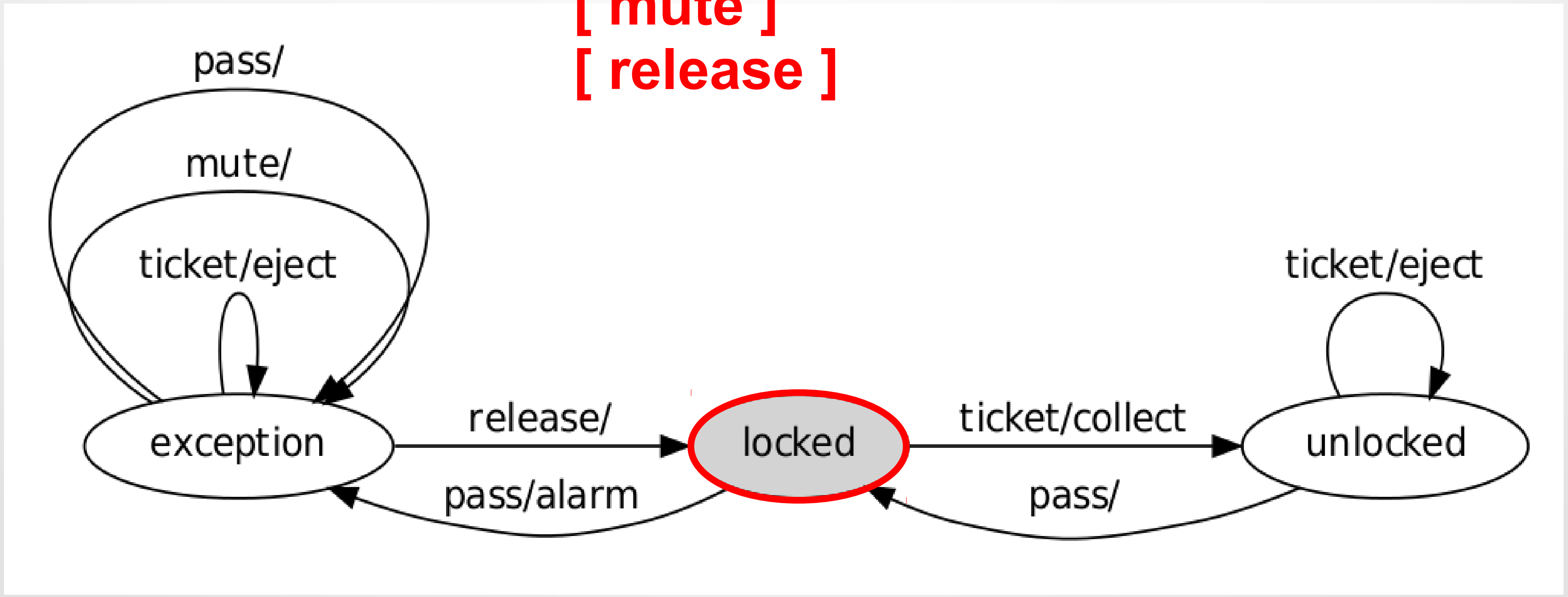


Negative test cases

- Example : infeasible input

start at initial node

[mute]
[release]



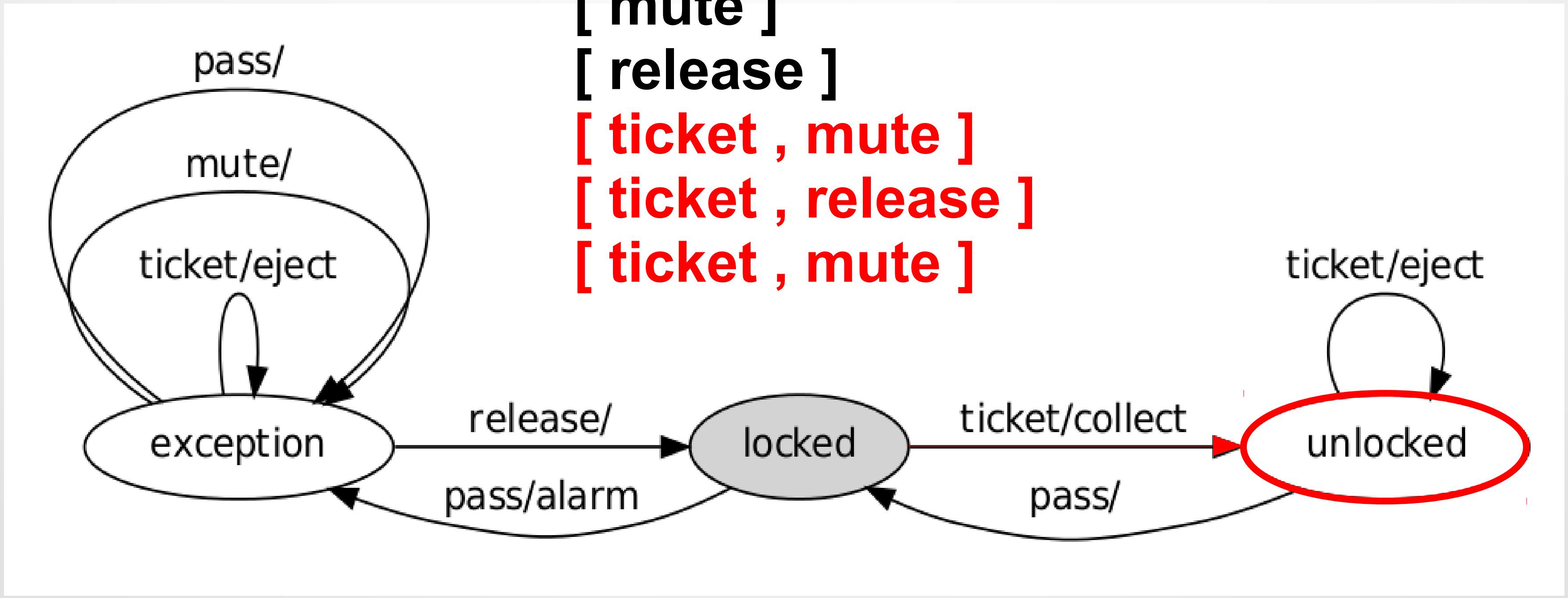
Negative test cases

- Example : infeasible input

  start at initial node

  walk through graph and append input to inputfile

[mute]
[release]
[ticket , mute]
[ticket , release]
[ticket , mute]



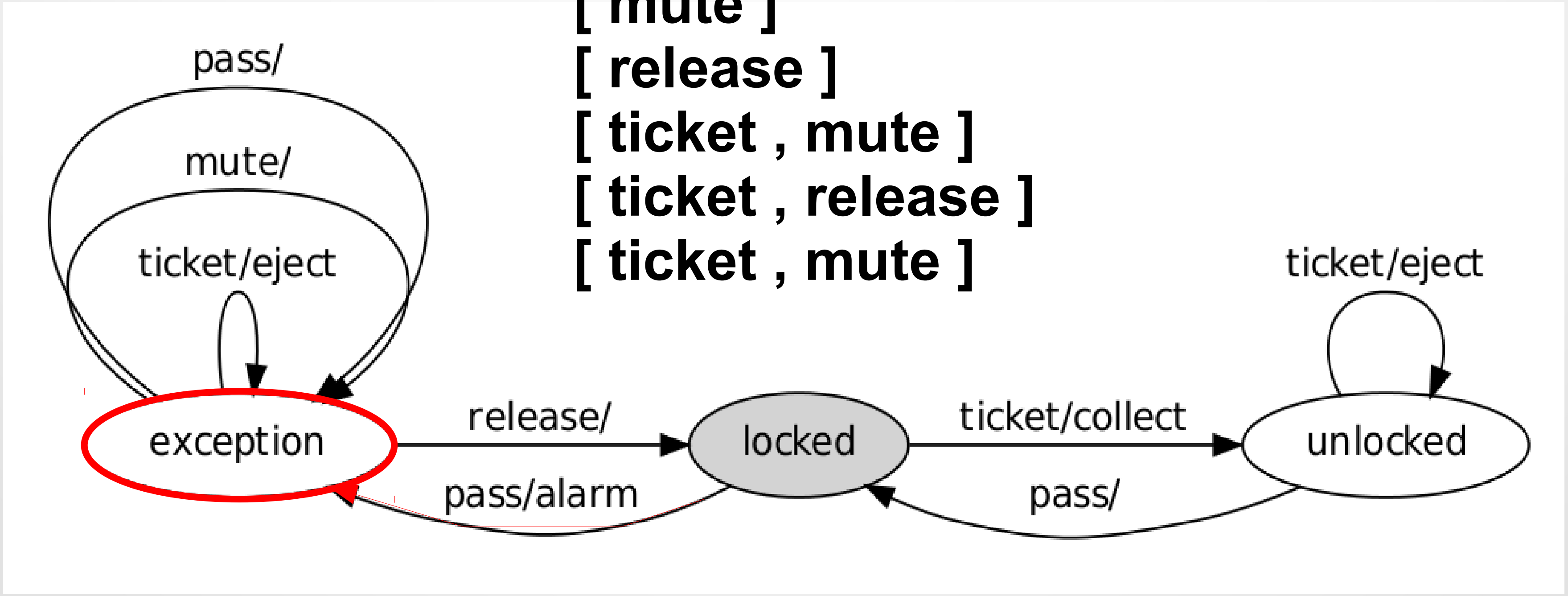
Negative test cases

- Example : infeasible input

  start at initial node

  walk through graph and append input to inputfile

[mute]
[release]
[ticket , mute]
[ticket , release]
[ticket , mute]



Negative test cases

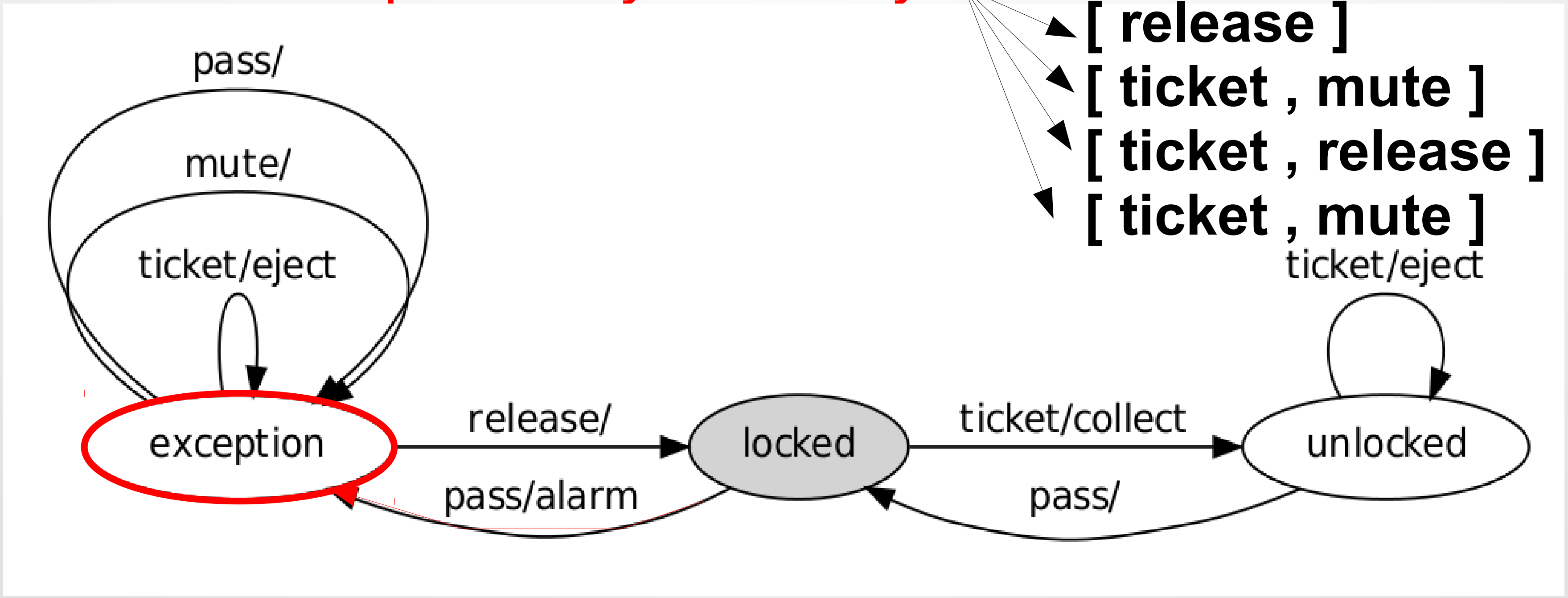
- Example : infeasible input

  start at initial node

  walk through graph and append input to inputfile

  select one possibility randomly

[mute]
[release]
[ticket , mute]
[ticket , release]
[ticket , mute]



Negative test cases

- Results for depth = 7

  ~ 300 test data files for positive and negative test cases
  → 3000 test cases in total

- depth = 8

  ~ 1900 test data files for positive and negative test cases
  → 19000 test cases in total