Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

Modelling and testing

IPM-18EszKVMTEG

A model based approach to testing Neovim's mode transitions

Zahorán Barnabás
idggb6@inf.elte.hu

April 28, 2022

Contents

1 Abstract **2**

2 Glossary **2**

3 Introduction **2**

3.1 System under test 2

3.2 Problem 3

3.3 Goal 3

4 Model **4**

4.1 States 4

4.2 Transitions 4

4.3 Creating the model 5

5 Test generation **8**

6 Test execution **9**

6.1 Test environment 10

6.2 Adaptation code 10

6.2.1 Remote controlling Neovim 10

6.2.2 The testing script 11

6.3 Test results 12

7 Future work **14**

8 Conclusion **14**

9 References **15**

1 Abstract

Modal text editors, like Vi and its derivatives, support different editing modes, each based around distinct tasks and each having a different set of keybindings. Reusing keys like this leads to a more efficient text editing workflow by decreasing the number of keystrokes required for frequent editing tasks. It is important for the modes and the transitions between them to be well defined and consistent. This study aims to present a model based approach to generating testcases that can verify the text mode transitions of Neovim.

Keywords: Neovim, nvim, modal editor, FSM, model based testing, test generation

2 Glossary

nvim: Neovim (nvim is the name we can use on the command-line to launch it)
mode: Editing mode of Neovim
FSM: Finite State Machine
MBT: Model-Based Testing
MTR: Model » Test » Relax MBT framework
TSV: Tab Separated Values text format
TT: Transition Tour MBT method
ATS: All-Transition-State MBT method
`<Esc>`: nvim's notation for the Escape key
`<Ins>`: nvim's notation for the Insert key
`<CR>`: nvim's notation for the Enter key (CR stands for Carriage Return)
`<C-c>`: nvim's notation for the CTRL-c key combination

3 Introduction

3.1 System under test

Neovim [1] is a fork and modernized version of Vim which itself is an ascendant of the Unix visual text editor Vi. They all support modal text editing and all Unix or Unix-like systems ship with at least Vi installed since about the seventies.

The program starts in `Normal` mode, where the user can navigate the cursor, search text or initiate a number of different text input commands. In the latter case, the editor prompts the user to enter text into the visual buffer. This is called `Insert` mode. `Visual`, `Visual line` and `Visual block` modes allow the user to select sections of the text. `Cmdline` allows us to run Vim commands, while `Ex` mode lets us issue legacy ex commands.

These are only a small portion of all modes. To get a complete list that also includes more obscure ones such as the different `Operator-pending` modes, run `:help mode()` in nvim.

The user can switch between these modes using certain keys, e.g. from `Normal` the `i` key takes us to `Insert` and `v` goes to `Visual`. In most modes, we can use the `<Esc>` key or `<C-[>` to return to `Normal`.

3.2 Problem

Neovim's user documentation [2] serves as a specification that describes for each mode what each key should do. If we only focus on the modes and mode switching, then we get a set of really simple requirements in the following form:

```
GIVEN we are at mode <M> IF key <K> is pressed, we must go to mode <N>
```

It is easy to see that most of the time `M` and `N` are the same, because with most keys we do not want to leave our current mode (e.g. `Insert` must stay in `Insert` for all letters, numbers and punctuations pressed). Focusing only on the modes, we can even call these "Nop" operations.

Testing whether a running nvim version fully complies its documented behavior in terms of mode switching is a challenging task for a number of reasons:

- Neovim has a total of 34 modes.
- On a standard US keyboard there are about 48 keys for letters, numbers and punctuation marks. (Not counting numpad keys, function keys and other special buttons). Each of them can be pressed with the Shift, Ctrl or AltGr modifiers. This leaves us with 192 key combinations.
- For a full test coverage we must also check all "Nop" operations discussed above.

This leaves us with at least 34 * 192 = 6528 cases to check. Later I will discuss about some multi-key cases too, but around 7000 is a good rough estimate. It is clear that checking all these is only viable through some automated form of testing.

3.3 Goal

We not only wish to automate the test execution, but also the testcase generation, because writing thousands or even hundreds of testcases by hand would take countless working hours. Thankfully, an FSM driven model naturally fits our problem, so we can apply model based testing techniques.

4 Model

4.1 States

The finite state machine's states represent nvim's modes.

4.2 Transitions

The FSM's inputs will represent the keypresses, or to be more precise: series of keypresses. Since no matter which mode the user is in, they can press any key, the FSM must be total. As a result, it must include all the reflexive transitions that leaves us in the current mode.

Here is a small example:

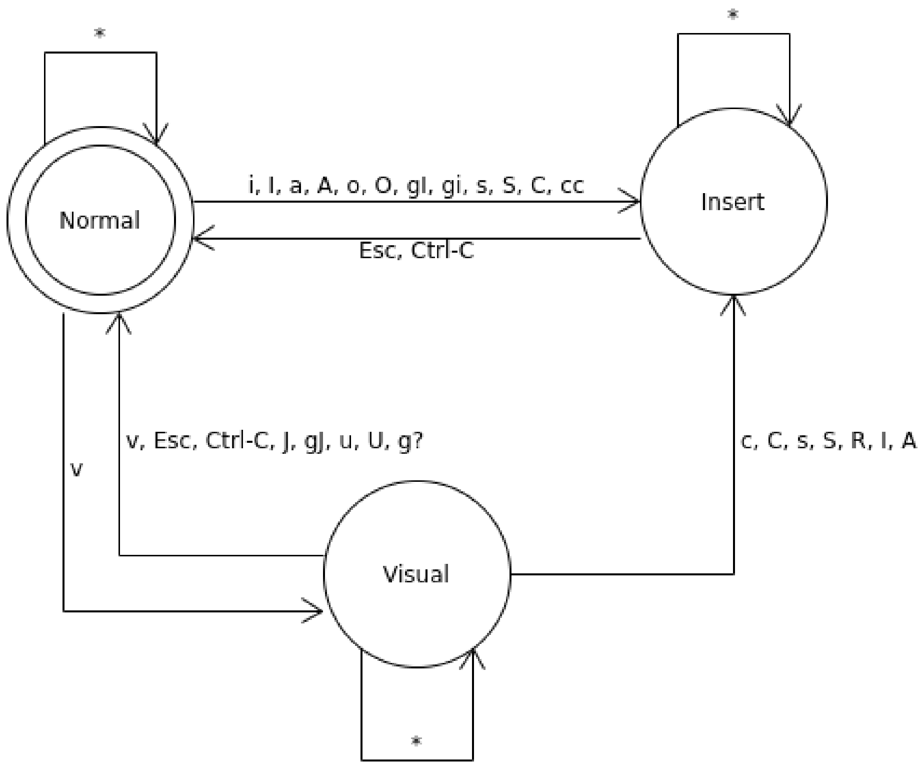


Figure 1: A simple example: FSM diagram

For the sake of simplicity, I took many liberties with the above diagram:

- Instead of drawing a separate arrow for each transition, I condensed all key variations that start and go to the same state into a single arrow.
- Here the `'*'` means "all other keys" and not the actual asterisk.
- Since this only contains a small subset of the actual modes, all transitions that go to undrawn states are also omitted.

- There are two kinds of problematic inputs both worth to discuss: `Ctrl-C` and the likes of `gi` and `cc`.

Multikey inputs such as `Ctrl-C` are perfectly fine, because `Ctrl` is only a modifier key and does not have any effect on its own (we can keep pressing `Ctrl` anytime, nothing will happen). As a result, we can consider combinations like `Ctrl-C` a unique (virtual) key. Note: capital letters are also in a similar boat, e.g. `A` is technically `Shift-a`.

Inputs like `gi` and `cc` could be considered invalid if and only if we only allowed single keypresses to count as inputs, but, in fact, we can consider entire key sequences as input for any transition. It will not create ambiguity, because `gi` and `gI`, despite having the same prefix, are not equal. We do not need to introduce additional intermediate modes to handle such cases. Whenever we press a key which needs some follow-up key(s) to determine its action, nvim always waits for those follow-up key(s). At test execution, we can simply feed nvim the `c` key twice in a row for example.

Darcy Parker created a comprehensive diagram [3] that lists all states and transitions, albeit it was created in 2012 and for regular Vim, not Neovim. Still, his work matches our expected model in 99%.

4.3 Creating the model

Simple example

I used Model » Test » Relax [4], an open source MBT framework developed at ELTE. It takes the FSM specification in JSON format as input and it can run various test generation algorithms on it. Here is the model for the small example discussed above:

```
{
  "models":[
    {
      "name":"Neovim_modes_FSM_small_example",
      "startElementId":"n",
      "id":"0",
      "vertices":[
        { "id":"n", "name":"Normal" },
        { "id":"i", "name":"Insert" },
        { "id":"v", "name":"Visual" }
      ],
      "edges":[
        { "id":"e0",  "name":"i",      "sourceVertexId":"n", "targetVertexId":"i", "input" : "i",      "output" : "" },
        { "id":"e1",  "name":"I",      "sourceVertexId":"n", "targetVertexId":"i", "input" : "I",      "output" : "" },
        { "id":"e2",  "name":"a",      "sourceVertexId":"n", "targetVertexId":"i", "input" : "a",      "output" : "" },
        { "id":"e3",  "name":"A",      "sourceVertexId":"n", "targetVertexId":"i", "input" : "A",      "output" : "" },
        { "id":"e4",  "name":"o",      "sourceVertexId":"n", "targetVertexId":"i", "input" : "o",      "output" : "" },
        { "id":"e5",  "name":"O",      "sourceVertexId":"n", "targetVertexId":"i", "input" : "O",      "output" : "" },
        { "id":"e6",  "name":"gI",     "sourceVertexId":"n", "targetVertexId":"i", "input" : "gI",     "output" : "" },
        { "id":"e7",  "name":"qi",     "sourceVertexId":"n", "targetVertexId":"i", "input" : "qi",     "output" : "" },
```

Figure 2: A simple example: JSON input for MTR (not all edges are on the image)

Some key notes:

- I used the abbreviated mode characters from nvim's manual as state ids (n, i, v).
- The modes' full names are written for state names.
- Edge ids can simply be unique numbered keys.

- I have given the same name to the transitions as the input they take.
- We do not need to define any output, because we will be able to query the current mode (state) at any given time in the tests (discussed later). Differentiating between the states will not be a problem.

Actual model

I did not include all 34 modes of nvim in the actual model for a number of reasons:

- This project is intended to be more of a proof of concept or a case study, than a fully comprehensive testing of nvim.
- There are many "pseudo modes" like the different `Operator Pending` ones. They can be accessed by pressing a key that starts an *action*, then nvim waits for a *motion* key, or series of keys that count as a motion. This would give us countless cases of different action-motion combinations, even if we ignore all negative cases where the action is not followed by a proper motion. It would be possible to extend the FSM to cover all these, but fully specifying all possible transitions for all possible keys with all operators would require a considerable amount of effort in itself. However, it could be a future improvement.
- For similar reasons, I also omitted more obscure and harder to test states like the `Insert Completion` one or `Special Character Pending`.
- If we really wanted to be pedantic, there would be an infinite number of states for some scenarios. For example, if we enter `Ex` mode, we get a prompt which we can only exit by typing `vi<CR>`. However, this only works if the prompt is empty beforehand, otherwise things like `arbitrary-text-vi<CR>` won't leave the mode.

The used FSM covers the following nine modes:

```
"vertices":[
   { "id":"n",      "name":"Normal"         },
   { "id":"v",      "name":"Visual"         },
   { "id":"V",      "name":"Visual line"    },
   { "id":"CTRL-V", "name":"Visual block"   },
   { "id":"i",      "name":"Insert"         },
   { "id":"R",      "name":"Replace"        },
   { "id":"Rv",     "name":"Virtual Replace" },
   { "id":"c",      "name":"Commandline"    },
   { "id":"cv",     "name":"Ex"             }
],
```

Figure 3: States of our model

In order to fill the JSON file with all transitions, I defined a simpler TSV format which I preprocess with an awk one-liner. The format (which also accepts hashmarked lines as comments):

```
<FROM-STATE> Tab <KEYS> Tab <TO-STATE>
```

```
# Normal -> Ex
n       gQ       cv

# Ex -> Normal
cv      vi<CR>   n

# Normal -> Insert
n       i        i
n       I        i
n       a        i
n       A        i
n       o        i
n       O        i
```

Figure 4: Portion from transitions.tsv

And the awk command that turns this format into the JSON lines for MTR:

```
$ awk '/^[^#]/ { print "{ \"id\":\"e",n++,"\", \"name\":\"",$2,"\",
\"sourceVertexId\":\"",$1,"\", \"targetVertexId\":\"",$3,"\",
\"input\":\"",$2,"\", \"output\":\"\" }," }' OFS= transitions.tsv
```


```
"edges":[
   { "id":"e0", "name":"Q", "sourceVertexId":"n", "targetVertexId":"cv", "input":"Q", "output":"" },
   { "id":"e1", "name":"vi<CR>", "sourceVertexId":"cv", "targetVertexId":"n", "input":"vi<CR>", "output":"" },
   { "id":"e2", "name":"i", "sourceVertexId":"n", "targetVertexId":"i", "input":"i", "output":"" },
   { "id":"e3", "name":"I", "sourceVertexId":"n", "targetVertexId":"i", "input":"I", "output":"" },
   { "id":"e4", "name":"a", "sourceVertexId":"n", "targetVertexId":"i", "input":"a", "output":"" },
   { "id":"e5", "name":"A", "sourceVertexId":"n", "targetVertexId":"i", "input":"A", "output":"" },
   { "id":"e6", "name":"o", "sourceVertexId":"n", "targetVertexId":"i", "input":"o", "output":"" },
   { "id":"e7", "name":"O", "sourceVertexId":"n", "targetVertexId":"i", "input":"O", "output":"" },
```


Figure 5: Generated JSON lines for the above transitions

The MBT framework can also visualize the model by adding the `--graphviz` [5] option. This outputs a .dot file from which we can generate images:

```
$ dot -Tpng Neovim_modes_FSM.dot > graphviz_model.png
$ dot -Tpng Neovim_modes_FSM-augmented.dot > graphviz_model_Eulerian.png
```

Note: I also added the `ratio=1.0;` line to the .dot files so the images' widths and heights are about the same.

The below image is attached at `images/graphviz_model.png`.
There is also `images/graphviz_model_Eulerian.png`, which depicts the Eulerian-augmented graph that MTR generated. Most algorithms require the FSM to have an Euler path (path touching every edge exactly once).

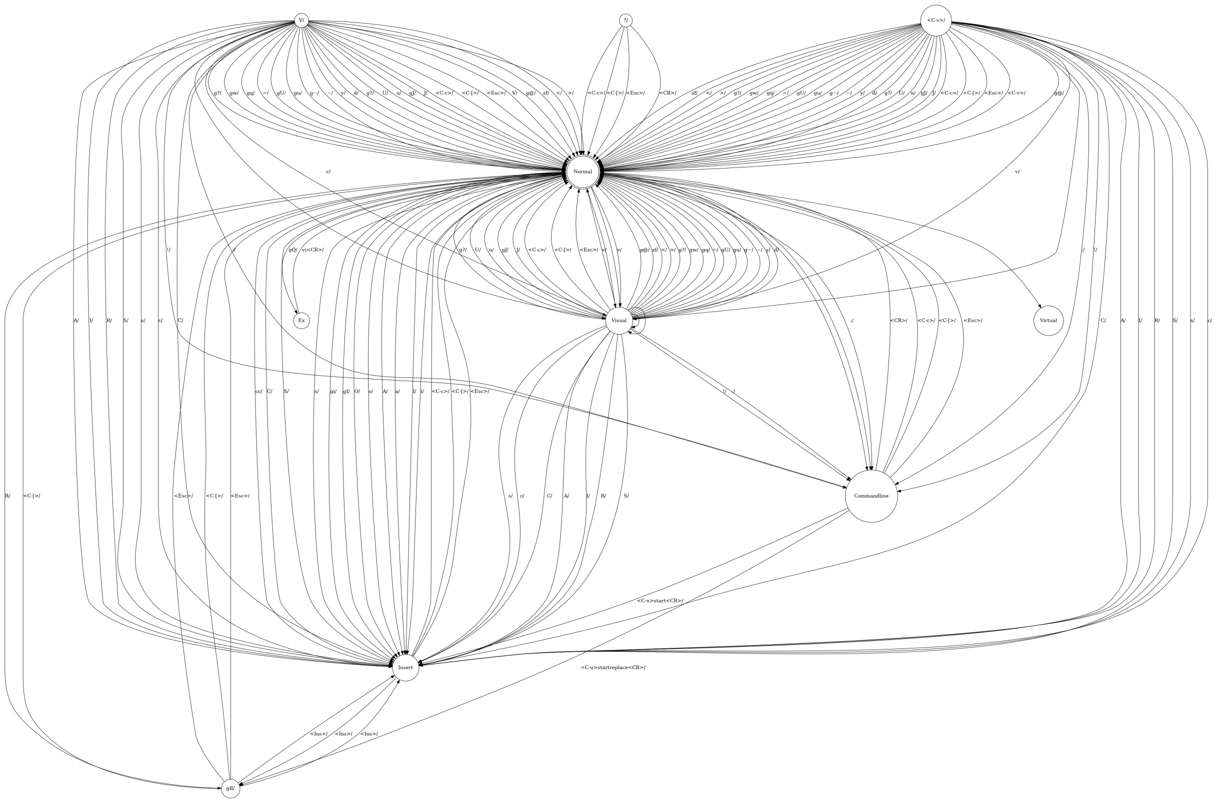


Figure 6: Base model generated by MTR using graphviz

5 Test generation

Now that Neovim_modes_FSM.json is complete, we can run MBT algorithms on it using the MTR framework: Random walk [6], Transition Tour (TT) [7] and All-Transition-State (ATS) [8].

```
$ ./MTR -m Random -f Neovim_modes_FSM.json --pretty_json
$ ./MTR -m TT     -f Neovim_modes_FSM.json --pretty_json
$ ./MTR -m ATS    -f Neovim_modes_FSM.json --pretty_json
```

The TT method generated the shortest test sequence: 274 inputs (which includes mostly single keypresses and some 2-3 long keystrokes) is enough to traverse all transitions.

Random walk generates random inputs until it reaches its stopping condition. This condition was set to be 100% transition coverage. (I ran the algorithm a couple of times, the resulting input sequence was always around 8000-11000 in length). The TT method was run with 100% state- and transition coverage. ATS also guarantees covering all transitions. So the tests will cover all possible mode changes that we included in our model. Using these input sequences, we will be able to run automated tests to verify if all transitions are correct.

Algorithm	Runtime	Test suite size	Test sequence length	Summary
Random walk	0.022228 s	434 KB	8183	rand_result.csv
TT	0.042106 s	15 KB	274	tt_result.csv
ATS	0.096331 s	35 KB	652	ats_result.csv

Table 1: MBT algorithm runs and resulting test suites

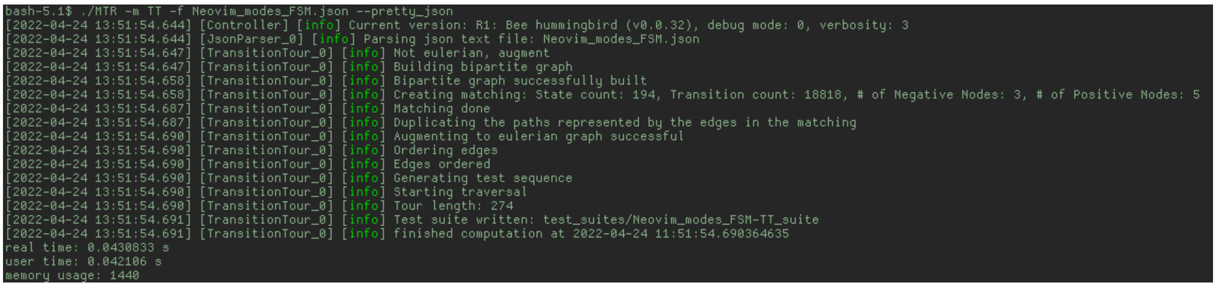


Figure 7: Running the TT method with the MBT framework

```
"input_list": [
    "v",
    "<C-v>",
    "V",
    "<C-v>",
    "v",
    "V",
    "v",
    "!",
    "<C-u>startreplace<CR>",
    "<Esc>",
    "v",
    ":",
    "<C-u>start<CR>",
    "<C-c>",
    "v",
    "A",
    "<C-c>",
    "V",
    "!",
    "<CR>",
```

Figure 8: Start of the test input sequence generated by the TT method
It starts with checking the transitions between the three Visual modes

6 Test execution

There are two special commands that transition from Cmdline to other modes: start<CR> and startreplace<CR>. Initial testing revealed a problem with them: we can reach Cmdline from any of the Visual modes by pressing : or even !. In this case, nvim adds the selection indicating prefix to the command: :'<,'>. However, if we insert either of the above commands, we will get a prompt like :'<,'>start<CR>, which, instead of switching mode, will only throw an error. Solution: I added a <C-u> prefix to these transitions, i.e., <C-u>start<CR> and <C-u>startreplace<CR>. <C-u> deletes all text till the beginning of the prompt. This solves a more general problem: no matter what got inserted to the command prompt so far, these two corrected keystrokes will always transition correctly.

Another issue was that the two search keys `/` and `?` transition to `Cmdline` mode and the above two commands will not work with them, not even with the `<C-u>` prefix. The only solution is to introduce a new virtual mode to our model: "Cmdline search" (with id="cs"). But how can we differentiate this from regular `Cmdline`, when nvim's `mode()` function only returns 'c' for both? Luckily, nvim has another function called `getcmdtype()`, which tells us what type of cmdline we currently have. Now our FSM has 10 states.

6.1 Test environment

OS: x86_64 GNU/Linux 5.15.32-1-MANJARO
SUT: NVIM v0.7.0, Build type: Release
Tools: GNU Awk 5.1.1; Python 3.10.4; MTR version R1: Bee hummingbird (v0.0.32)

6.2 Adaptation code

6.2.1 Remote controlling Neovim

For the test execution, we can utilize nvim's remote controlling capabilities. It provides an API [9] for running RPC calls through a TCP socket.

In order to easily connect and use this API from a Python script, we must first install the `Pynvim` package:

```
$ pip3 install pynvim
```

The next step is to start nvim with its `--listen` option. In this example, it will listen for calls at port 10000 of localhost:

```
$ nvim -u NONE --listen 127.0.0.1:10000
```

Note: the `-u NONE` option makes nvim ignore any startup configuration (e.g. .vimrc file) and it also does not load any plugins. This way, we can make sure that the SUT is truly a vanilla nvim instance and no user configurion or plugin will distort the test results. (For example, the user config could easily remap keys from their defaults which would lead to false results).

Now we can connect to the API from python with the following lines:

```
>>> from pynvim import attach
>>> nvim = attach("tcp", address="127.0.0.1", port=10000)
```

We will use three nvim functions:

- **call**: We can call nvim's internal "mode()" function, which returns the string that represents the current mode.
- **feedkeys**: Simulates keypresses as if they were feeded to nvim directly from the keyboard.
- **replace_termcodes**: Needs to be used in conjunction with feedkeys so nvim's key representations - also used in our model - get translated to proper terminal codes, e.g. <Esc> to \x1b.

For example, to check if nvim starts in Normal mode, then by pressing i it goes to Insert, then back to Normal with <Esc>, we can use the code snippet below:

```
>>> nvim.call('mode') == 'n'
>>> nvim.feedkeys('i')
>>> nvim.call('mode') == 'i'
>>> nvim.feedkeys(nvim.replace_termcodes("<Esc>"))
>>> nvim.call('mode') == 'n'
```

6.2.2 The testing script

The script is named test_exec.py, its short manual reads as follows:

```
bash-5.1$ ./test_exec.py --help
Usage: ./test_exec.py [-h|--help] | <transitions.tsv> <test_suite.json> [--verbose]

<transitions.tsv>   File containing keys that transition between modes in TSV format
<test_suite.json>   Output file of MTR framework in JSON format
--verbose           If set, all assertions will be printed to standard output
-h, --help          Prints this help message

Precondition: nvim must listen on port 10000 of localhost, i.e. it was started like:
$ nvim -u NONE --listen 127.0.0.1:10000
```

Figure 9: Running the script with the -h or --help option

Its first argument must be the transitions.tsv we used to generate the model and the second must be MTR's test suite output.

After parsing its input files, it connects to the running nvim instance. Then it iterates over the test suite's input_list and feeds each key sequence in it to the SUT. Before and after feeding each input, it queries nvim's current mode and then it checks whether we arrived to the correct destination mode based on the specification from transitions.tsv.

Finally, it writes a test report that contains statistics about the number of assertions (total, ok, failed), the success rate and the execution time. If the --verbose option is set, it also outputs all assertions.

6.3 Test results

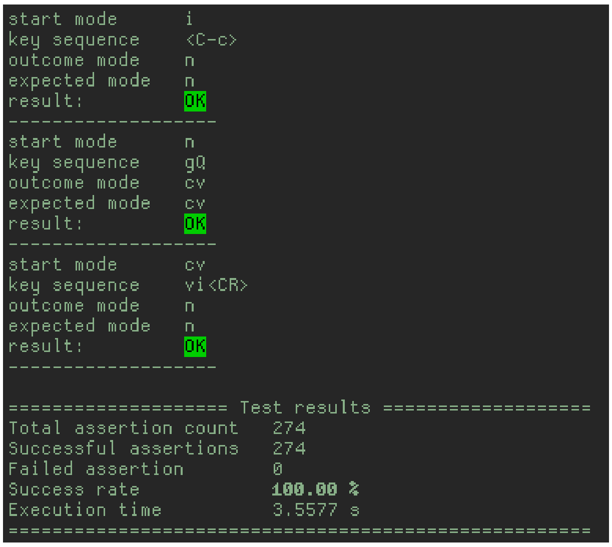


Figure 10: Part of the verbose output of `test_exec.py` on the TT generated test suite

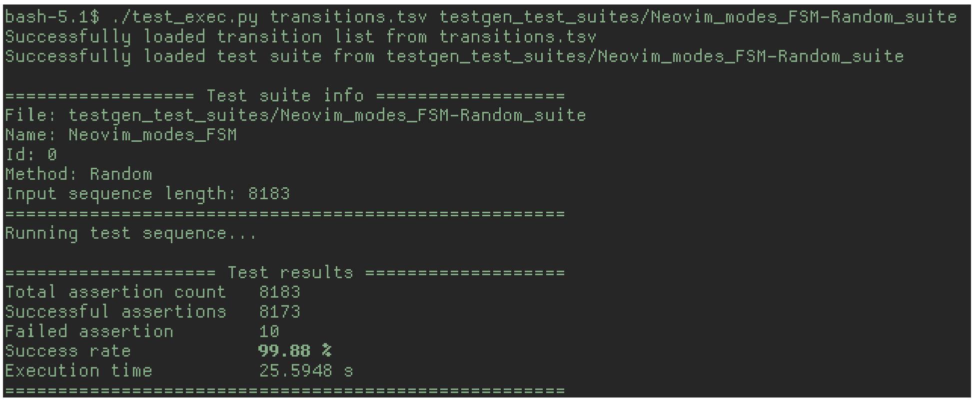
Algorithm	Runtime	No. assertions	Passed	Failed	Success rate
Random walk	25.7596 s	8183	8173	10	99.88 %
TT	3.5577 s	274	274	0	100 %
ATS	10.2623 s	652	652	0	100 %

Table 2: Test execution results

The test suites generated by TT and ATS both fully passed the tests. The Random walk method's input threw 10 failed assertions out of 8183. This proves that the Random method, despite it being inefficient, can be useful for revealing defects the other methods failed to catch. But if the other two method also covered all transitions, how could a failure happen with the Random walk? The problem hides in the model. There is a case where a transition needs a longer state history, then solely its preceding mode, i.e., at A->B->C, what C should be not only depends on B, but also A. All ten failed assertions are the same problem: from both `Replace` and `Virtual Replace`, we can press the <Ins> key to enter `Insert` mode, then <Ins> again to return to where we came from. So if we transitioned from simple `Replace`, we return there, if we came from `Virtual Replace`, then we return there. Neovim remembers which replace mode it came from. There could be two ways to fix this:

- Extend the model with a new mode, e.g. `Insert-coming-from-Rv`. It should work exactly like `Insert`, with the only difference being that on <Ins> it would go to `Virtual Replace` instead of `Replace`.
- Augment the FSM to an EFSM (Extended Finite State Machine) with a variable that can tell us if we came from a replace mode to `Insert` and if yes, which one. The

MTR framework could convert the EFSM to an FSM for us with a command like:
`$ ./MTR -o conversion -m efsm_to_fsm -f Neovim_modes_EFSM.json --efsm`
Once we got to the FSM, the workflow would be the same. This solution would be more laboursome, but maybe going for an EFSM model could prove useful in the long-run for similar technicalities.

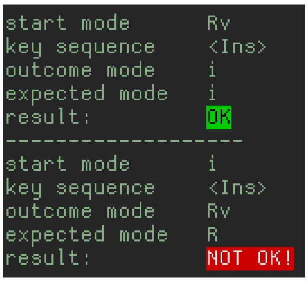


```
bash-5.1$ ./test_exec.py transitions.tsv testgen_test_suites/Neovim_modes_FSM-Random_suite
Successfully loaded transition list from transitions.tsv
Successfully loaded test suite from testgen_test_suites/Neovim_modes_FSM-Random_suite

================= Test suite info =================
File: testgen_test_suites/Neovim_modes_FSM-Random_suite
Name: Neovim_modes_FSM
Id: 0
Method: Random
Input sequence length: 8183
===================================================
Running test sequence...

=================== Test results ==================
Total assertion count   8183
Successful assertions   8173
Failed assertion        10
Success rate            99.88 %
Execution time          25.5948 s
===================================================
```

Figure 11: Output of `test_exec.py` on the Random walk generated test suite



```
start mode       Rv
key sequence     <Ins>
outcome mode     i
expected mode    i
result:          OK
------------------
start mode       i
key sequence     <Ins>
outcome mode     Rv
expected mode    R
result:          NOT OK!
```

Figure 12: Failed assertion in the Random walk

7 Future work

There are several ways to extend or improve this test solution:

- Cover all 34 modes of nvim and all transitions between them.
- Add all reflexive transitions by defining a comprehensive set of key inputs. We would only need to write out all key inputs only once, because the MTR framework can fill up the rest of the unspecified transitions with loops using the `-o conversion -m ps_to_cs` options.
- The MTR tool could be further utilized. We could run other MBT methods like ATT or HSI, or we could even use the framework for injecting random faults to the model to check how good are the testcases in detecting all defects.
- Outlook: some techniques (e.g. use of the RPC API) could be applied to test other aspects of nvim too. For example, to test where different motions move the cursor, or how different actions affect the buffer. These do not align so naturally with an FSM model, but it may still be possible to translate these problems to some extent.

8 Conclusion

This study has introduced readers to translating software behavior to a finite state machine, constructing a model from it, generating executable testcases through different MBT methods with the MTR framework, and successfully running and evaluating these tests using API calls and minimal logic in a small script.

The example shows that model based approaches to test generation and automatic test execution can greatly reduce the amount of manual work. The only hand-written part of the entire testing workflow, besides the script, is the initial transitions.tsv file (214 lines of simple triples, a fully specified model for nvim would also probably be less than 1000 lines here).

The lengths and runtimes of the resulted tests are also in an acceptable range. The longest taking 25 seconds to complete, I think it is safe to assume that running on even a fully specified model of nvim could be measured on the scale of minutes.

Generated testcases and adaptation code like this could be used in regression testing nvim itself as well as similar software with minimal effort.

9 References

[1] Neovim homepage,
https://neovim.io/ (2022.04.28)

[2] Neovim's user manual,
https://neovim.io/doc/user/ (2022.04.28)

[3] Darcy Parker's Vim mode state diagram,
https://gist.github.com/darcyparker/1886716 (2022.04.28)

[4] Model » Test » Relax MBT framework,
https://gitlab.inf.elte.hu/nga/ModelTestRelax (2022.04.28)

[5] Graphviz homepage,
https://graphviz.org/ (2022.04.28)

[6] Random walk method's documentation in MTR,
https://gitlab.inf.elte.hu/nga/ModelTestRelax/
-/blob/master/docs/user_guide.md#61-random-walk (2022.04.28)

[7] Transition Tour method's documentation in MTR,
https://gitlab.inf.elte.hu/nga/ModelTestRelax/
-/blob/master/docs/user_guide.md#62-transition-tour (2022.04.28)

[8] All-Transition-State method's documentation in MTR,
https://gitlab.inf.elte.hu/nga/ModelTestRelax/
-/blob/master/docs/user_guide.md#63-all-transition-state (2022.04.28)

[9] Neovim's API reference,
https://neovim.io/doc/user/api.html (2022.04.28)

[10] Pynvim's GitHub page,
https://github.com/neovim/pynvim (2022.04.28)