Github Pull Request lifecycle test

Peter Kokai

Github Pull Request lifecycle test

1. Abstract

This document provides a description of how https://github.com/ PR (Pull Request) works from user point of view, and how the fMBT tool is used to test its interface.

2. Description

Github describes itself as:

> "Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world."

It is a platform where communities could share code, in its core using git. The git tool itself does not define any process. But many communities and tools started to advertise their own. One of them is called github workflow. The main purpose of these processes is to share code written independently (time, person, space, etc...). Usually there are few branches (mostly one) that is considered Truth, and everybody wants to migrate their changeset into that. Github workflow describes a process how to do it, and github itself by default provides utilities to support that. (Note: it is possible to use other workflows on github, but there are no tools to help you from github itself.)

A current project aim is to create a model of the github workflow code migration as implemented by github and use that model to test their implementation. A short version of that workflow:

- Fork the project (if not forked before)
- Create a branch from `truth` (Note: other may use `master`, `main`, `dev`; the `truth` is used here as the branch to be desired.)
- Add the desired modification to the new branch
- Push changes into github server
- Create a Pull Request
- Quality check (review, CI)
- Integrate changeset into `truth` (merge, rebase, ...)

3. fMBT

The fMBT tool is a tool created by intel, that aim is to provide model based testing toolset such as:

- model editor
- test generator
- test executor
- language bindings(c++, python, java and many more)
- log analyser
- debugger

Each connected component interface is simply, and the toolchain does support their replacement. The tool support two model type:

- GT (older)
- AAL (newer, recommended)

The AAL format allows mixing the model and the SUT (software under test) code mainly to provide adapter code between SUT and model.

3.1 AAL

This is an introduction section for AAL model descriptor language used by fMBT with using an example from the fMBT repository. The example is called python-unittest, the SUT is a class that can increment a counter, reset that counter to the initial value and provide the current value it holds for the user.

```
class MyCounter:
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1

    def reset(self):
        # self.value = self.value / self.value - 1
        self.value = 0

    def count(self):
        return self.value
```

The uncommented line is an example bug that could be used to test how fMBT could find a faulty behaviour.

The AAL languages describes possible edges, and edges can connect nodes based on the guard conditions. The nodes are implied from the edges and variables. The `variables` block can be used to pre-define variables used in the model, and global variables that maybe needed for SUT. Variables that are used for the SUT should be excluded from the view itself. That can be done with listing variables

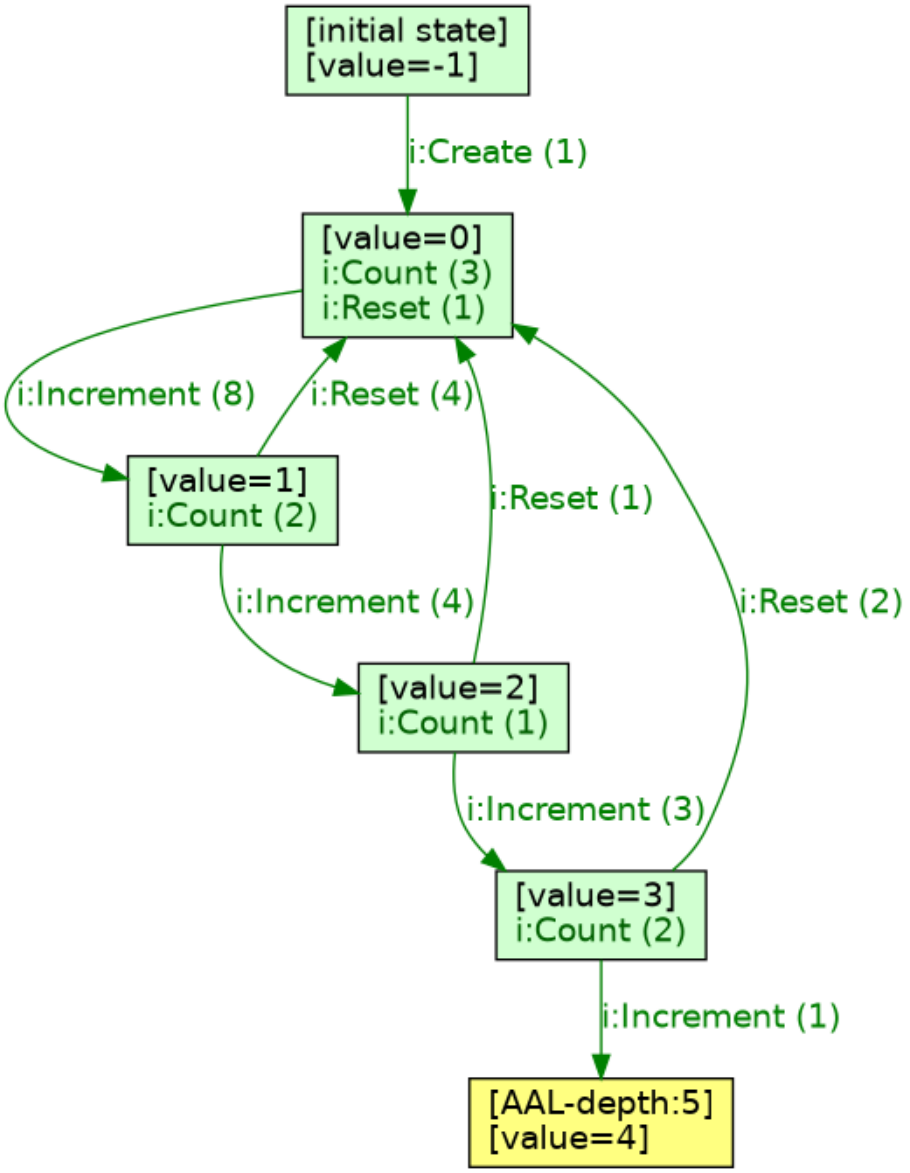
that are **used** in the model: `preview-show-vars`, effectively a node split if differs in shown variable. As in the example project if the **value** is displayed it creates a lot of nodes with possible different values, when marked as hidden only two node are created one with all the value and initial state. As a result of this, it is possible to create infinitely large graph, that is limited by the *AAL-depth* parameter. The actions can be described as `input` (in older version it was called `action`), these actions connects the vertices of our graph. Optionally it has blocks called `guard()`, `body()` and `adapter()`. The `adapter()` should contain code that calls SUT and not considered part of the model itself, also `assert` or exception should be thrown here if the test is to be failed. The `guard()` block should return with a boolean value, the code can use the variables defined in `variables` but should avoid using SUT. This determines if an input can be taken in a given state of the model, and if that action can be taken and taken the `body()` part is executed, after the `body()` also the `adapter()` is executed. For model view only the `body()` and `guard()` is enough to generate possible test executions.

The AAL also supports `tag()` blocks. The `tag()` block can contain `guard()` and `adapter()` and nodes are tagged if the `guard()` returns true, and `adapter()` is executed if the state changes. Tag can be used to group common checks. Also `input()` block can be embedded into `tag()` block, where `input()` block inherits the `guard()` of its parent `tag()`.

```
# preview-show-vars: value
aal "mycountertest" {
    language: python {
        import mycounter
    };
    variables { mcobject, value }
    initial_state {
        mcobject = None
        value = 0
    }
    input "Create" {
        guard() { return mcobject == None }
        body() { value = 0 }
        adapter() {
            mcobject = mycounter.MyCounter()
            assert mcobject.count() == 0
        }
    }
    input "Increment" {
        guard() { return mcobject != None }
        body() { value += 1 }
        adapter() {
            mcobject.inc()
        }
```

```
    }
    input "Reset" {
        guard() { return mcobject != None }
        body() { value = 0 }
        adapter() {
            mcobject.reset()
        }
    }
    input "Count" {
        guard() { return mcobject != None }
        adapter() {
            assert mcobject.count() == value
        }
    }
}
```

Using the fmbt-editor, we can generate the model as a graph. The following graph shows the model described before:



The node has labels with the displayed variables value, using the [] and black text colour, the edges that points to the same node are not drawn but written into the node. An edge must be started with "i:" (representing that it is an input), or "o:" (as an output), after that prefix it follows the name provided by the user and a number which indicates how many times it was triggered in the generated test. The colour of an edge shows edge coverage, if the edge text and arrow is black that means it was not triggered; on the other hand if it is green that was at least once used.

3.2 Configuration

The configuration file can be used to connect different components like `model`, `adapter` or previous execution results `history`. The `heuristic` option provides way for the user to select algorithm for test creation (graph walk), and with `pass`, `fail` or `incnc` the user can provide condition when to stop the execution/test generation.

Let's check a configuration line by line:

```
model   = "aal_remote(remote_pyaal -l pyaal.log mycountertest.py.aal)"
adapter = aal
coverage = perm(2)
pass    = steps(100)
fail    = duration(2 secs)
on_fail = exit(42)
```

The `model` and `adapter` lines here configured to use the built in model and adapter implementation, which are both provided based on the AAL file.

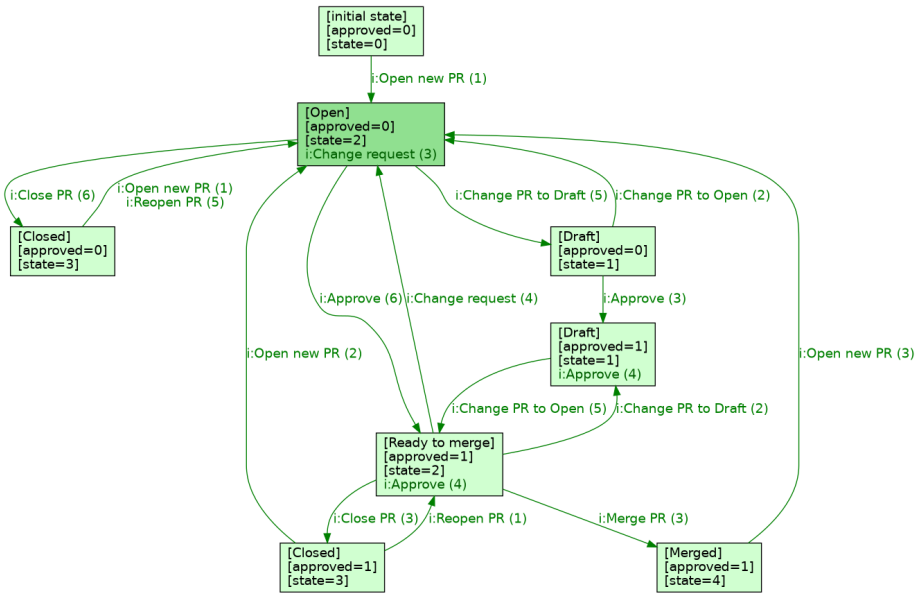
The `coverage` says, that 100% is reached if every 2 inputs combination is visited (`perm(2)`), `perm(1)` would mean that use every possible action at least once. (Note, this does not always mean to cover every vertices).

The `pass` criteria is to take 100 steps in the model, and if it take more than 2 seconds it should fail (`fail = duration(2 secs)`). When the test fails, `on_fail` is executed, here exiting with 42.

There are a lot of option for each of these parameters, and there are a few parameter left out. Please check documentation for more details.

4. Model

This parts going to describe the model used for covering github PR. The different state of github PR are open, ready to merge, closed, merged, draft. These states are marked with AAL `tag()` feature.



Variables:

- state: open, closed, merged, draft, ready to merge
- approved: the number of approve received from different users (currently 1), and PR creator cannot approve
- SUT: this is a hidden variables for `adapter()` not actually part of the model

Actions:

- Open new PR: create a new branch, pushes the code and create a PR on github
- Merge: if there are enough approve, it merges the branch
- Approve: marks PR ready if approve needed is configured
- Change request: denies merging via asking user to change the patch
- Convert to draft: the PR became unmergable
- Mark ready to review (from draft)
- Close: closes the PR without merging code

The Open new PR input is only allowed when the previous PR is closed, merged or there is no PR before. The creation of new PR after close and merge is part of the model, so the test could "reset" its state and reach higher coverage. It can be omitted, in which case multiple execution is required.

5. Adapter code

The main interface of github is the git protocol and its web page. In addition to the web page recently github started a CLI interface development. It's second

form is called **gh** as github cli. The github cli was a good candidate for the glue code between AAL and SUT, sadly the gh does not provide all the functionality which github is capable. It lacks two area, one regarding Draft PR. It cannot convert an already open PR to draft or a draft PR to an open(non draft) state. The second issue, which is really shocking, there is no good way to obtain a status of a PR, I mean if it is open, draft, there is change request etc. Due to this I started to use the web interface via selenium, that was fine. I could do anything, except Approving PR! Of course the PR approval works in github, it was just a simple conflict that if user A opens a PR, it cannot approve its own PR (which makes a tons of sense). When I changed users, it worked as expected. The blocking issue here was that simply changing the users were required, which sounds more simple than it is. In the day of 2 step authentication automating the second step usually hard (specially if your second step is a physical yubikey). At this point I made a decision using the fact that I want to test github internal state regarding the PR status, so the focus is not the web page nor the cli, those are just tool to interact. I choose to mix both of them, as for the github cli one could simply generate and use a token bypassing the 2FA, and the cli could do most of the task. Those tasks that was not possible via github cli were done via web page using selenium.

```
def OpenNewPR(self):
    self.prid = self.__get_next_pr_id()
    branch_name = git_new_branch(self.prid)
    sh.gh.pr.create("--fill", "--head",
                    branch_name,
                    _env=get_env("github-fmbt.token"))
    return self.prid

def ClosePR(self):
    sh.gh.pr.close(self.prid, _env=get_env())

def Open2Draft(self):
    self.__ClickOnText("Convert to draft")
    for_sure=self.driver.find_element_by_xpath("...")
    for_sure.click()
```

The **ClosePR** is an example of github cli usage, where once could observe the `_env=get_env()` argument, which fetches the access token. The second is **Open2Draft** that uses selenium for interacting with the web page itself, where the selenium browser can be authenticated before the process and store the session for later usage. As for the **OpenNewPR**, the github cli of course used, with the different user so the limitation of approving your PR is bypassed. This part of the code also had to create some code modification on a new branch, the new branch is created based on the next PR id (github uses an incremented id for PR/issues), this ensures that even if local github repository is used the PR names are unique. Also this is a place where git command must be used to push the new branch and code. These methods are wrapped into a class

`SUT=GithubPR("Kokan/github-fmbt"), so the` github.aal‘ code itself could be clean and hight level.

```
...
    input "Close PR" {
        guard()   { return state == PRState.OPEN }
        adapter() {
                    log("Close PR")
                    SUT.ClosePR()
                  }
        body()    { state = PRState.CLOSED }
    }
...
    tag "Closed" {
        guard()   { return state == PRState.CLOSED }
        adapter() { assert SUT.GetPRStatus() == PRState.CLOSED }
    }
...
```

6. Test execution

The configuration used to generate tests:

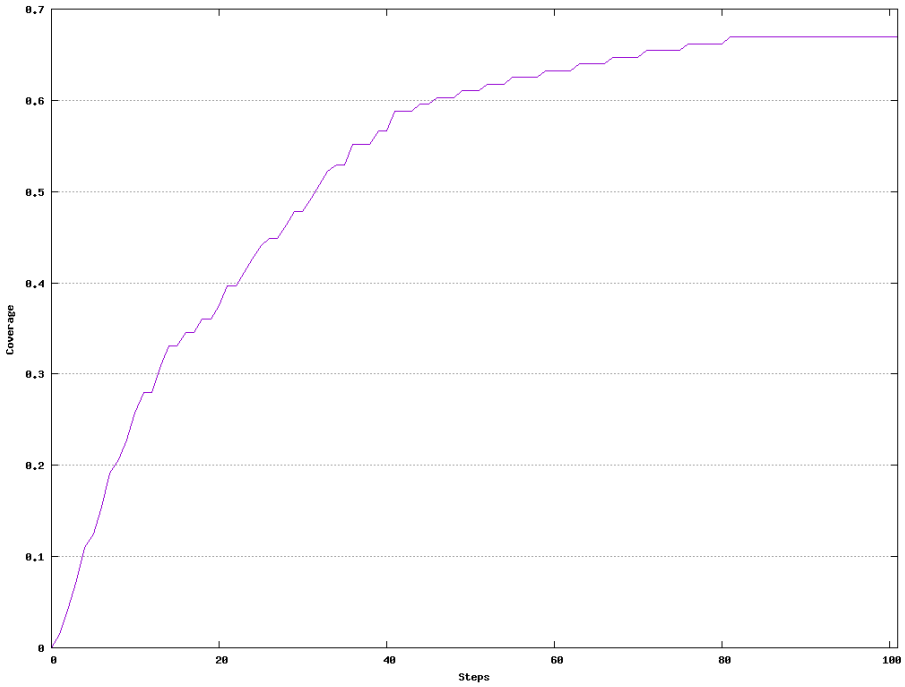
```
heuristic = lookahead(5)
coverage  = sum(perm(3, '.*', '.*', 'i:Merge PR'), perm(3, '.*', '.*', 'i:Close PR'))
pass      = noprogress(20)
```

The algorithm uses `lookahead(5)`, which generates the possible new walks with 5 length, and check if any of them increases the coverage. The `coverage` is defined as a sum of inputs trios where the input ends with either `Merge PR` or `Close PR`. The `pass` option here configured in a way, that if coverage is not increased in 20 step, it stops. (noprogress(20)), with this option it always terminates with the highest coverage it can reach. If the `pass` would ask for coverage not possible to reach, it would run forever.

Test step execution times

min[ms]	med[ms]	max[ms]	total[ms]	called	"input"
14.794	17.683	27.739	183	10	"i:Change PR to Open"
28.242	33.088	45.721	349	10	"i:Reopen PR"
14.828	24.331	58.756	425	16	"i:Open new PR"
48.486	51.015	83.872	488	9	"i:Merge PR"
43.141	53.403	99.563	555	10	"i:Change PR to Draft"
43.125	50.662	66.917	566	11	"i:Change request"
44.284	52.728	81.250	902	16	"i:Close PR"
43.311	45.685	77.796	960	19	"i:Approve"

The coverage displayed at every step:



The sequence of inputs also generated and saved into a log file, which can be queried with fmbt-log. A truncated example:

```
i:Open PR
i:Close PR
i:Reopen PR
i:Change PR to Draft
i:Change PR to Open
i:Approve
i:Approve
...
```

The stats with different **coverage** settings:

perm(2)

min[ms]	med[ms]	max[ms]	total[ms]	called	"action"
12.949	15.353	18.552	79	5	"i:Change PR to Open"
29.778	32.556	38.349	101	3	"i:Merge PR"
17.299	18.497	23.993	114	6	"i:Reopen PR"
27.619	31.057	36.338	156	5	"i:Change request"
8.778	17.009	26.651	162	10	"i:Open new PR"
26.551	33.040	41.318	191	6	"i:Change PR to Draft"

min[ms]	med[ms]	max[ms]	total[ms]	called	"action"
12.764	26.196	39.488	360	16	"i:Approve"
26.646	29.305	43.897	372	12	"i:Close PR"

perm(1)

min[ms]	med[ms]	max[ms]	total[ms]	called	"action"
11.425	11.425	11.425	11	1	"i:Open new PR"
12.892	12.892	12.892	13	1	"i:Change PR to Open"
17.716	17.716	17.716	18	1	"i:Reopen PR"
26.356	26.356	26.356	26	1	"i:Change request"
27.617	27.617	27.617	28	1	"i:Approve"
27.775	27.775	27.775	28	1	"i:Change PR to Draft"
28.518	28.518	28.518	29	1	"i:Close PR"
32.533	32.533	32.533	33	1	"i:Merge PR"

With the above method the coverage reaches 66.911800%. With `perm(1)` it quickly reaches 100%, and with `perm(2)` it stays below 60%. Also when `tag()` is used for coverage target (visiting every tag), reaching the 100% is not an issue.

Sadly the coverage must be pre-defined, and it is not possible to query different coverage values even with `fmbt-stats`. (As I can see in graphwalker, that provides stats for edges, vertices within one execution.) Comparing coverage of different execution are futile.

On the other hand, the `fmbt-editor` provides coverage information without calling the `adaptor()` code, so triggering the real test execution. It helps to experiment with configuration almost instantly.