Github Pull Request lifecycle test

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.

Description

Github describes itselfs as:

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 itsefl.)

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, . . .)

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 analyzer
- debugger

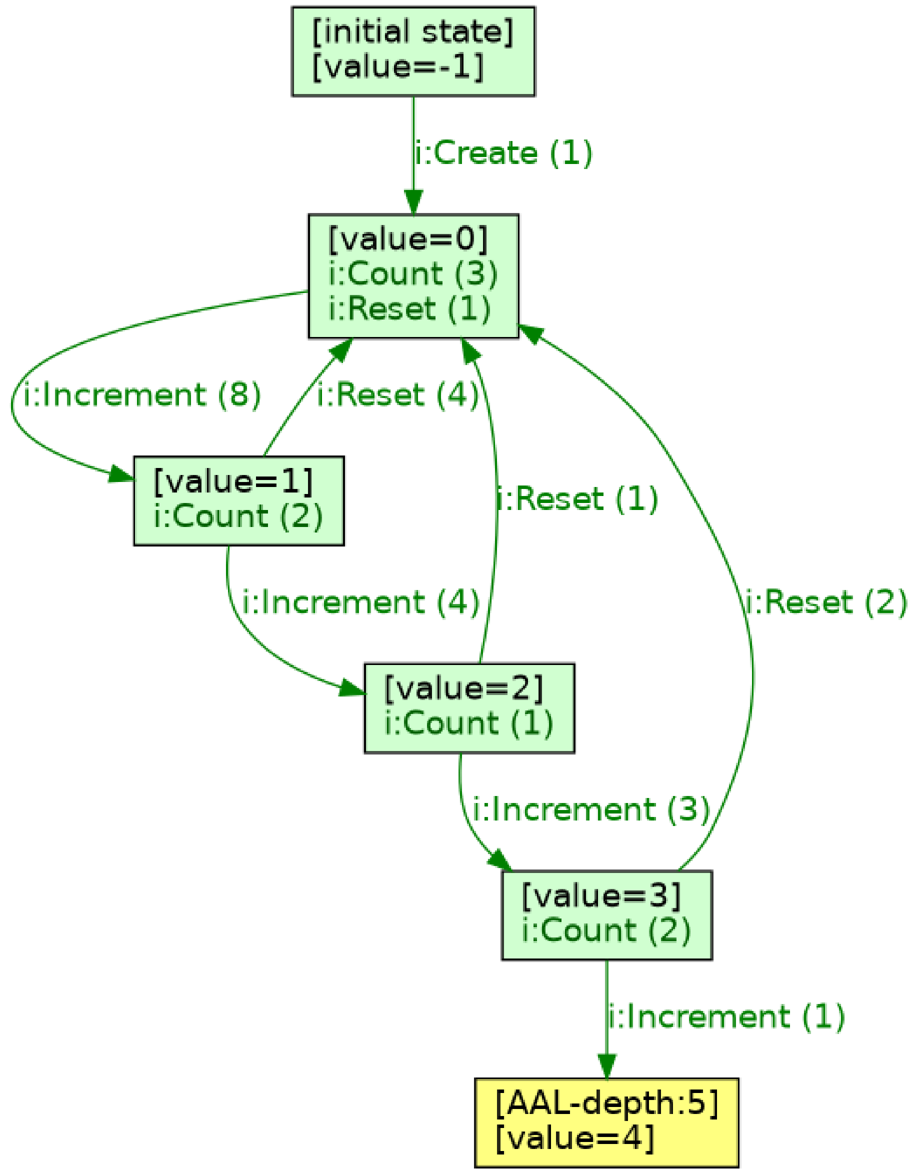
Each connected component interface is simply, and the toolchain does support their replacment. The tool support two model type:

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

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

AAL A brief introduction of AAL. Complete example is used from here.

The model graph view:



SUT is a simple counter implementation:

```
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 above model is generated from the following:

```
# 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
        }
    }
}
```

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`. The actions can be described as `input` (in older version it was called `action`), these actions connects the verteces 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 givven 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()`.

Configuration

The configuration file can be used to connect different components like `model`, `adapter` or previose execution results `history`. Provides way for the user to select algorithm for test creation (graph walk) with `heuristic` that describes the algorithm to traverse and stop condition with `pass`, `fail` or `inconc` and action on those event `on_pass`, `on_fail`.

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` line are used to use the built in model and adapter implementation, which are both provided based on the AAL file.

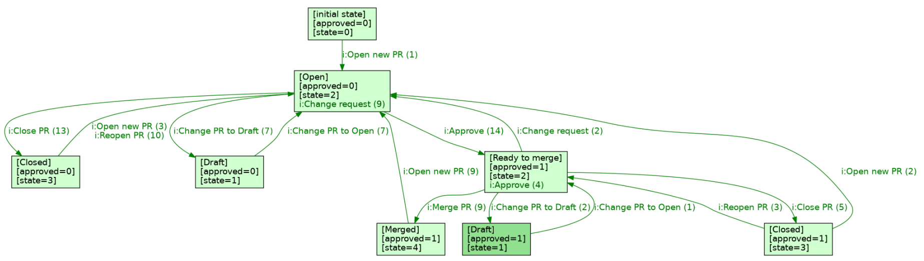
The `coverage` says, that 100% is reached if every 2 action combination is visited (`perm(2)`), `perm(1)` would mean that use every possible action at least once. (Note, this does not always mean to covery every verteces).

The `pass` criteria is to take 100 step in the model, and if it take more than 2 seconds it should fail (`fail = duration(2 secs)`), when fails it should return 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.

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
- approved: the number of approve recieved 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 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 action is only allowed when the previose 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 omited, in which case multiple execution is required.

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 actions trios where the action ands with either `Merge PR` or `Close PR`.

Test step execution times

min[ms]	med[ms]	max[ms]	total[ms]	count	"action"
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"

With the above method the coverages 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, verteces 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.