

DEVICE AND METHOD FOR RE-EXECUTING OF TEST CASES IN SOFTWARE APPLICATION

TECHNICAL FIELD

[0001] Various aspects of this disclosure relate to devices and methods for re-executing of test cases in a software application.

BACKGROUND

[0002] An essential part of developing software applications is validation and testing. There exist a variety of test tools to run test cases for applications being developed. In a micro service environment, test case executions are usually in different machines or networks. Test executions may fail due to micro service network delay, latency, virtual machine or docker instance issues. Therefore, test cases should be repeatable and deterministic.

[0003] Typically, a Go language (Golang) testing framework does not support automated re-execution of failed test cases. Accordingly, without the automated re-execution feature, each failed case has to be re-executed manually when doing test failure triaging, requiring a high effort to re-execute the failed test cases. Under such circumstances, when running a plurality of Golang test cases, supporting re-execution of test cases is critical to achieve good return on investment (ROI) of test automation, and to achieve higher testing efficiency.

[0004] Accordingly, approaches are desirable which allow more efficient identification, re-execution and management of test cases in a software application.

SUMMARY

[0005] Various embodiments concern a method for re-executing of test cases in a software application including (i) obtaining a first status of an execution of a test case for the software application under test, (ii) re-executing the test case if the first status of the execution of the test case indicates a fail status, and obtaining a second status of the re-execution of the test case, repeating steps (i) and (ii) for each test case in a plurality of test cases; obtaining a log of a plurality of first statuses of the plurality of test cases and at least one second status of at least one re-executed test case, searching for matches between test case names of the plurality of test cases and test case name of the at least one re-executed test case and if a match between the

test case names has been found, changing the first status of the match to a skip status and using the second status as a final status of the match.

[0006] According to one embodiment, the method includes removing test cases with the skip status in the log.

[0007] According to one embodiment, the method includes checking a status of a rerun flag of the test case.

[0008] According to one embodiment, the method includes re-executing the test case if the first status of the execution of the test case indicates a fail status and the status of the rerun flag indicates that the test case is under a rerun mode.

[0009] According to one embodiment, the software application is written in Golang.

[0010] According to one embodiment, obtaining the log includes receiving test result data for the plurality of test cases and parsing the log from test result data.

[0011] According to one embodiment, the method includes outputting information about the test by writing the information about the test into the log.

[0012] According to one embodiment, the method includes determining a source of error if the first status of the execution of the test case indicates a fail status.

[0013] According to one embodiment, the method includes outputting information regarding the source of error.

[0014] According to one embodiment, outputting information about the test and the source of error includes uploading the report log to a server.

[0015] According to one embodiment, outputting information about the test and outputting the error type includes displaying the information about the test and the source of error.

[0016] According to one embodiment, displaying information about the test including displaying the information in association with the test in a list of tests.

[0017] According to one embodiment, a server computer is provided including a communication interface, a memory and a processing unit configured to perform the method according to one of the embodiments described above.

[0018] According to one embodiment, a computer program element is provided including program instructions, which, when executed by one or more processors, cause the one or more processors to perform the method according to one of the embodiments described above.

[0019] According to one embodiment, a computer-readable medium is provided including program instructions, which, when executed by one or more processors, cause the one or more processors to perform the method according to one of the embodiments described above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0020] The invention will be better understood with reference to the detailed description when considered in conjunction with the non-limiting examples and the accompanying drawings, in which:

- FIG. 1 shows a computer for development of software applications.
- FIG. 2 shows a flow diagram illustrating the re-execution of test cases according to an embodiment.
- FIG. 3 shows a flow diagram illustrating a report cleanup of test cases according to an embodiment.
- FIG. 4A shows a table in the report showing exemplary statuses of test cases according to an embodiment.
- FIG. 4B shows a table in the report showing exemplary statuses of test cases before and after cleanup according to an embodiment.
- FIG. 5A shows a flow diagram illustrating exemplary percentages of failures without the rerun function according to an embodiment.
- FIG. 5B shows a flow diagram illustrating exemplary percentages of failures with the rerun function according to an embodiment.
- FIG. 6 shows a flow diagram illustrating a method for re-executing of test cases in a software application according to an embodiment.
- FIG. 7 shows a server computer system according to an embodiment.

DETAILED DESCRIPTION

[0021] The following detailed description refers to the accompanying drawings that show, by way of illustration, specific details and embodiments in which the disclosure may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the disclosure. Other embodiments may be utilized and structural, and logical changes may be made without departing from the scope of the disclosure. The various

embodiments are not necessarily mutually exclusive, as some embodiments can be combined with one or more other embodiments to form new embodiments.

[0022] Embodiments described in the context of one of the devices or methods are analogously valid for the other devices or methods. Similarly, embodiments described in the context of a device are analogously valid for a vehicle or a method, and vice-versa.

[0023] Features that are described in the context of an embodiment may correspondingly be applicable to the same or similar features in the other embodiments. Features that are described in the context of an embodiment may correspondingly be applicable to the other embodiments, even if not explicitly described in these other embodiments. Furthermore, additions and/or combinations and/or alternatives as described for a feature in the context of an embodiment may correspondingly be applicable to the same or similar feature in the other embodiments.

[0024] In the context of various embodiments, the articles “a”, “an” and “the” as used with regard to a feature or element include a reference to one or more of the features or elements.

[0025] As used herein, the term “and/or” includes any and all combinations of one or more of the associated listed items.

[0026] In the following, embodiments will be described in detail.

[0027] Fig. 1 shows a computer 100 for development of software applications.

[0028] The computer 100 includes a CPU (Central Processing Unit) 101 and a system memory (RAM) 102. The system memory (e.g., a random-access memory (RAM)) 102 is used to load program code, i.e. from a hard disk (i.e. HDD) 103, and the CPU 101 executes the program code.

[0029] In the present example it is assumed that a user intends to development a software application using the computer 100. For this, the user executes a software development environment 104 on the CPU 101. The user may use a computing language such as Golang, which is a statically typed, compiled programming language, to develop the application.

[0030] The software development environment 104 allows the user to develop an application 105 for various devices 106, for example smartphones. For this, the CPU 101 runs, as part of the software development environment 104, a simulator to simulate the device for which an application is developed, e.g. for a mobile device.

[0031] When the user has successfully developed an application, the user may distribute it to corresponding devices 106 via a communication network 107, e.g. distribute it to smartphones by uploading it to an app store.

[0032] Before that happens, however, the user should test the application 105 to avoid that an application that does not work properly is distributed to devices 106. To do this, the user further runs a testing tool 108 on the CPU 101, for example using a Golang testing framework in a heterogeneous environment, for example to perform automated testing Representational State Transfer (REST) on Application Programming Interfaces (APIs). The heterogeneous environment may mean a system with hardware and system software from different vendors.

[0033] Current Golang testing framework only supports three test statuses for a test case: PASS, FAIL, and SKIPPED. There is no existing way to represent a “RERUN” case. This may lead to testing inefficiency as manual testing may be needed to re-execute a test case.

[0034] In a typical automation testing, users may simply deploy all test cases in the same environment or network which is less complicated than deploying test cases in a system in the heterogeneous environment. This inevitably exponentially increases the complexity and likelihood of intermittent failures. To achieve higher automation ROI and testing efficiency within the heterogeneous environments, a re-execution or a re-run feature is highly desirable.

[0035] Therefore, according to various embodiments, a system for re-execution of test cases is demonstrated. This allows better testing efficiency.

[0036] In various embodiments, the testing tool 108 may be used for re-executing of test cases in a software application. The testing tool 108 may obtain a first status of an execution of a test case for the software application under test. The first status may be for example a pass status or a fail status.

[0037] The testing tool 108 may re-execute the test case if the first status of the execution of the test case indicates a fail status.

[0038] The testing tool 108 may obtain a second status of the re-execution of the test case. The second status may be for example a pass status or a fail status.

[0039] The testing tool 108 may repeat steps of execution of a test case and re-execution of the test case if the initial execution of the test case indicates a fail status for each test case in a plurality of test cases.

[0040] The testing tool 108 may obtain a log (or a report) of a plurality of first statuses of the plurality of test cases and at least one second status of at least one re-executed test case. The testing tool 108 may search for matches between test case names of the plurality of test cases and test case name of the at least one re-executed test case. If a match between the test

case names has been found, the first status of the match may be changed to a skip status and the second status may be used as a final status of the match.

[0041] As used herein, the term “re-execution” is used interchangeably with the term “re-run”.

[0042] In various embodiments, the computer 100 may remove test cases with the skip status in the log. Since the test cases with the skip status in the log have been re-executed, by removing the skipped test cases may simplify the log for more efficient data management.

[0043] In various embodiments, the computer 100 may check a status of a rerun flag of the test case. The rerun flag may indicate whether the test case is under a rerun mode or not under a rerun mode. For example, if the rerun flag is set to “1”, a rerun mode may be activated, and when the rerun flag is set to “0”, a rerun mode may be deactivated.

[0044] In various embodiments, the computer 100 may re-execute the test case if the first status of the execution of the test case indicates a fail status and the status of the rerun flag indicates that the test case is under a rerun mode.

[0045] In various embodiments, obtaining the log may include receiving test result data for the plurality of test cases and parsing the log from test result data. In an embodiment, parsing the log from the test result data may include error detection or source of error analysis from the test result data. The log may be parsed by command line tools.

[0046] In various embodiments, the computer 100 may output information about the test by writing the information about the test into the log. For example, information about the test may be written into the log which may be in XML format. The log may be shown in a tabular format.

[0047] In various embodiments, the computer 100 may determine a source of error if the first status of the execution of the test case indicates a fail status.

[0048] In various embodiments, the computer 100 may output information regarding the source of error. The source of error may be for example an interim failure or a product code issue.

[0049] In various embodiments, outputting information about the test and the source of error may include uploading the report log to a server. In various embodiments, outputting information about the test and outputting the error type includes displaying the information about the test and the source of error. In various embodiments, displaying information about the test including displaying the information in association with the test in a list or table of tests.

[0050] In various embodiments, a server computer is provided including a communication interface, a memory and a processing unit configured to perform the method according to one of the embodiments described above.

[0051] In various embodiments, a computer program element is provided including program instructions, which, when executed by one or more processors, cause the one or more processors to perform the method according to one of the embodiments described above.

[0052] In various embodiments, a computer-readable medium is provided including program instructions, which, when executed by one or more processors, cause the one or more processors to perform the method according to one of the embodiments described above.

[0053] FIG. 2 shows a flow diagram 200 illustrating the re-execution of test cases according to an embodiment.

[0054] In flow diagram 200, at step 202, the execution of a test suite (i.e., validation suite) for the software application (e.g., Golang testing) starts. The test suite is a collection of test cases that may be intended to be used to test a software application to show that it has some specified set of behaviours. At step 203, the test suite will initialise with the necessary setups or data preparation.

[0055] At step 204, iteratively, each test case in the test suite will start to run beginning from a first test case. At step 205, if the test case (e.g., first test case) passes, a rerun will not be triggered, and will therefore continue (back to step 203) to the next test case (if any). If the test case fails, step 206 is triggered. At step 206, a rerun flag is checked to see if “RERUN” is set. If “RERUN” is set, the same case (i.e., the failed case) will be triggered again at step 204. At step 206, if the rerun flag is not set, the failed case will be recorded as a “FAIL” status and the system will proceed to a next test case at step 203. If the test case passes at step 205, step 207 is triggered to check whether the test suite has ended. If the test suite has not ended, the system will proceed to a next test case at step 203. If the test suite has ended, the process ends at step 208.

[0056] In various embodiments, an exemplary source code for re-execution of the test case may be the following:

```
func runTests(matchString func(pat, str string) (bool, error), tests []InternalTest) (ran, ok bool)
{
    rerunFlag := false
    reran := false
```

```

if strings.ToUpper(os.Getenv("RERUN")) == "TRUE" {
    rerunFlag = true
}
ok = true
for _, procs := range cpuList {
    runtime.GOMAXPROCS(procs)
    for i := uint(0); i < *count; i++ {
        if shouldFailFast() {
            break
        }
        ctx := newTestContext(*parallel, newMatcher(matchString, *match, "-
test.run"))

        t := &T{
            common: common{
                signal: make(chan bool),
                barrier: make(chan bool),
                w:      os.Stdout,
                chatty: *chatty,
            },
            context: ctx,
        }
        tRunner(t, func(t *T) {
            for _, test := range tests {
                r := t.Run(test.Name, test.F)
                if rerunFlag && !r {
                    time.Sleep(5 * time.Second)
                    reran = true
                    t.reRunFailed = !t.Run(test.Name, test.F)
                }
            }
        })
        // Run catching the signal rather than the tRunner as a separate
        // goroutine to avoid adding a goroutine during the sequential

```

```

// phase as this pollutes the stacktrace output when aborting.
go func() { <-t.signal }()
})
if !rerunFlag {
    ok = ok && !t.Failed()
} else {
    if !reran {
        ok = ok && !t.Failed()
    } else {
        ok = ok && !t.reRunFailed
    }
}
ran = ran || t.ran
}
}
return ran, ok
}.

```

[0057] In the exemplary source code above, the system checks if a rerun flag is set. If the rerun flag is not set, the test case will not be rerun. For example, the exemplary source code may be: `rerun flag := false, rerun := false`. If rerun flag is set and the test case fails, the test case will be rerun. For example, the exemplary source code may be: `!rerunFlag { ok = ok && !t.Failed()`. The testing process may be in the form of a loop (e.g., a For loop), and may be implemented for each test case of the test suite. The testing process may also include conditional statements (e.g., if else statements) to handle different scenarios of having the rerun flag set or not set, and having the status of the test case be a pass or a fail status.

[0058] FIG. 3 shows a flow diagram 300 illustrating a report cleanup of test cases according to an embodiment.

[0059] At step 301, the testing suite 301 ends and the log (XML file) is generated. At step 302, a log (report) cleanup will be triggered to clean up the log. At step 303, each test cases of the test suite in the log will be processed. At step 304, for each of the test cases, the testing tool checks if it is the same test case name as the next test case. If yes, at step 305, the current test case will be marked as “Skipped”. Else, a status (e.g., Pass/Fail) of the test case remains the

same and the system proceeds to step 306 to check if the current test case is the last test case. In other words, the system checks if there are any more test cases to be processed. If the current test case is not the last case, the system proceeds to step 303 to process a new test case. If the current test case is the last case, the proceed ends at step 307.

[0060] In various embodiments, an exemplary source code for the report cleanup may be the following:

```
for i := 0; i < len(suites.Suites); i++ {
    for j := 0; j < len(suites.Suites[i].TestCases); j++ {
        if suites.Suites[i].TestCases[j].Failure != nil {
            //fmt.Println(k.Name)
            if j+1 < len(suites.Suites[i].TestCases) {
                if suites.Suites[i].TestCases[j].Name ==
suites.Suites[i].TestCases[j+1].Name {
                    suites.Suites[i].TestCases[j].SkipMessage =
&JUnitSkipMessage{Message: "RERUN: " + suites.Suites[i].TestCases[j].Failure.Contents}
                    suites.Suites[i].TestCases[j].Failure = nil
                }
            }
        }
    }
}.

```

[0061] In the exemplary source code above, the system checks if the current test case has the same test case name as the next test case. If yes, the current test case will be marked as “Skipped”. For example, the exemplary source code to mark the current test case as “Skipped” may be: `if suites.Suites[i].TestCase[j].Name == suites.Suites[i].TestCase[j+1].Name { suites.Suites[i].TestCase[j].SkipMessage`. The cleanup process may be in the form of a loop (e.g., a For loop), and may be implemented for each test case entry in the log. The cleanup process may also include conditional statements (e.g., if else statements) to handle different scenarios of having the same test case name or a different test case name.

[0062] FIG. 4A shows a table 400 in the report showing exemplary statuses of test cases according to an embodiment.

[0063] In exemplary table 400, for each test case (e.g., Test Case A 406, Test case B 408 and Test Case C 410), there may be a first status 402 (i.e., status of initial run) and/or a second status 404 (i.e., status of rerun).

[0064] In the example, when a test case (e.g., Test Case A 406) has a pass status as the first status 402, a rerun may not be required and the second status 404 may be labelled as a nil status. As another example, when a test case (e.g., Test Case B 408 or Test Case C 410) has a fail status as the first status 402, a rerun may be done on the test case (e.g., Test Case B 408 or Test Case C 410). As shown in the example, the second status 404 may be a pass status or a fail status. In an embodiment, if the second status 404 is a fail status, the system may allow for a further rerun of the test case.

[0065] FIG. 4B shows a table 420 in the report showing exemplary statuses of test cases before and after cleanup according to an embodiment.

[0066] In exemplary table 420, for each test case, there may be a before cleanup status 422 and/or an after cleanup status 424.

[0067] For example, for Test Case A 426, since the before cleanup status 422 is a pass status, a rerun may not be needed, and the after cleanup status 424 also reflects a pass status. As another example, for an initial run Test Case B 428A, an initial run may show a fail status before cleanup, while a rerun Test Case B 428B may show a pass status in the rerun before cleanup. Since the initial run Test Case B 428A and the rerun Test Case B 428B has the same name (i.e., Test Case B), the initial run Test Case B 428A may have an after cleanup status 424 of “skipped”. As another example, for an initial run Test Case C 430A, an initial run may show a fail status before cleanup, while a rerun Test Case C 430B may show a fail status in the rerun before cleanup. Since the initial run Test Case C 430A and the rerun Test Case C 430B has the same name (i.e., Test Case C), the initial run Test Case C 430A may have an after cleanup status 424 of “skipped”. In other words, if an initial test case has the same test case name as the rerun test case, the initial test case will have the status “skipped” regardless of whether the rerun test case shows a pass or fail status.

[0068] FIG. 5A shows a diagram 500 illustrating exemplary percentages of failures without the rerun function according to an embodiment.

[0069] In the exemplary diagram 500, out of 10,000 cases 502 under test, there may be 9900 pass cases 504 or 99% pass cases, and there may be 100 failures 506 or 1% failures. Out of the

100 failures 506, there may be 80 interim failures 508 or 0.8% interim failures and 20 product code issues 510 or 0.2% product code issues.

[0070] FIG. 5B shows a diagram 550 illustrating exemplary percentages of failures with the rerun function according to an embodiment.

[0071] In the exemplary diagram 550, out of 10,000 cases 552 under test, there may be 9970 pass cases 554 or 99.7% pass cases, and there may be 30 failures 556 or 0.3% failures. Out of the 30 failures 556, there may be 10 interim failures 558 or 0.1% interim failures and 20 product code issues 560 or 0.2% product code issues.

[0072] In various embodiments, if each case requires 5 minutes to triage, the failures in the system without rerun will require 500min (or 8 hours) to resolve. With the rerun function, interim failures 558 may be reduced down to 0.1%, effectively saving about 6 hours of error resolving time per day. Thus, this may lead to better engineering productivity and achieving better engineering ROI since triaging test failures may be time consuming. In a heterogeneous environment, a test case may fail due to unstable network, unstable operating system (OS), and/or third-party interim outage. Having a tool to support test case RERUN when it fails in the previous run, can increase testing stability and improve engineering productivity.

[0073] In summary, according to various embodiments, a method is provided as illustrated in FIG. 6.

[0074] FIG. 6 shows a flow diagram 600 illustrating a method for re-executing of test cases in a software application.

[0075] In 602, a first status of an execution of a test case for the software application under test may be obtained.

[0076] In 604, the test case may be re-executed if the first status of the execution of the test case indicates a fail status and a second status of the re-execution of the test case may be obtained.

[0077] In 606, steps 602 and 604 may be repeated for each test case in a plurality of test cases.

[0078] In 608, a log of a plurality of first statuses of a plurality of test cases and at least one second status of at least one re-executed test case may be obtained.

[0079] In 610, matches between test case names of the plurality of test cases and test case name of the at least one re-executed test case may be searched.

[0080] In 612, if a match between the test case names has been found, the first status of the match may be changed to a skip status.

[0081] Steps 602 to 612 are shown in a specific order, however other arrangements are possible. Steps may also be combined in some cases. Any suitable order of steps 602 to 612 may be used.

[0082] The approach of FIG. 6 can be used for different kinds of software automation testing like API automation for different platforms. According to various embodiments, a tool is provided to improve the productivity and efficiency of Automation Testing lifecycle by providing a tool to perform rerun of test cases with the inherent ability to serve as a unified test automation report platform. It may be used by any software company that utilizes Automation testing across multiple platforms (Web, Mobile, API), especially for software applications in Golang format.

[0083] The method of FIG. 6 is for example carried out by a server computer as illustrated in FIG. 7.

[0084] FIG. 7 shows a server computer system 700 according to an embodiment.

[0085] The server computer system 700 includes a communication interface 701 (e.g. configured to receive test result data or the log and configured to output the information about the test for example statuses or source of error). The server computer 700 further includes a processing unit 702 and a memory 703. The memory 703 may be used by the processing unit 702 to store, for example, data to be processed, such as the log. The server computer is configured to perform the method of FIG. 6. It should be noted that the server computer system 700 may be a distributed system including a plurality of computers.

[0086] The methods described herein may be performed and the various processing or computation units and the devices and computing entities described herein (e.g. the processing unit 602) may be implemented by one or more circuits. In an embodiment, a "circuit" may be understood as any kind of a logic implementing entity, which may be hardware, software, firmware, or any combination thereof. Thus, in an embodiment, a "circuit" may be a hard-wired logic circuit or a programmable logic circuit such as a programmable processor, e.g. a microprocessor. A "circuit" may also be software being implemented or executed by a processor, e.g. any kind of computer program, e.g. a computer program using a virtual machine code. Any other kind of implementation of the respective functions which are described herein may also be understood as a "circuit" in accordance with an alternative embodiment.

[0087] While the disclosure has been particularly shown and described with reference to specific embodiments, it should be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention as defined by the appended claims. The scope of the invention is thus indicated by the appended claims and all changes which come within the meaning and range of equivalency of the claims are therefore intended to be embraced.

CLAIMS

1. A method for re-executing of test cases in a software application comprising:
 - (i) obtaining a first status of an execution of a test case for the software application under test;
 - (ii) re-executing the test case if the first status of the execution of the test case indicates a fail status and obtaining a second status of the re-execution of the test case;repeating steps (i) and (ii) for each test case in a plurality of test cases;
obtaining a log of a plurality of first statuses of the plurality of test cases and at least one second status of at least one re-executed test case;
searching for matches between test case names of the plurality of test cases and test case name of the at least one re-executed test case;
if a match between the test case names has been found, changing the first status of the match to a skip status and using the second status as a final status of the match.
2. The method of claim 1, further comprising:
removing test cases with the skip status in the log.
3. The method of claim 1 or 2, further comprising:
checking a status of a rerun flag of the test case.
4. The method of any one of claims 1-3, further comprising:
re-executing the test case if the first status of the execution of the test case indicates a fail status and the status of the rerun flag indicates that the test case is under a rerun mode.
5. The method of any one of claims 1-4, wherein the software application is written in Golang.
6. The method of any one of claims 1-5, wherein obtaining the log comprises receiving test result data for the plurality of test cases and parsing the log from test result data.
7. The method of any one of claims 1-6, further comprising:

outputting information about the test by writing the information about the test into the report log.

8. The method of any one of claims 1-7, further comprising:

determining a source of error if the first status of the execution of the test case indicates a fail status.

9. The method of claim 8, further comprising:

outputting information regarding the source of error.

10. The method of any one of claims 1-9, wherein outputting information about the test and the source of error comprises uploading the report log to a server.

11. The method of any one of claims 1 to 10, wherein outputting information about the test and outputting the error type comprises displaying the information about the test and the source of error.

12. The method of any one of claims 1 to 11, wherein displaying information about the test comprising displaying the information in association with the test in a list of tests.

13. A server computer comprising a communication interface, a memory and a processing unit configured to perform the method of any one of claims 1 to 12.

14. A computer program element comprising program instructions, which, when executed by one or more processors, cause the one or more processors to perform the method of any one of claims 1 to 12.

15. A computer-readable medium comprising program instructions, which, when executed by one or more processors, cause the one or more processors to perform the method of any one of claims 1 to 12.