Descriptive Review for Software Testing Algorithms

Chew Kean Ho[1], Lim Lee Booi[2]

[1]*ZORALab Enterprise*
kean.ho.chew@zoralab.com
[2]*Independent*
lee.booi.lim@gmail.com

October 1, 2018

Abstract

Software testing tools are available in the market. However, they are very specific to their software domains such as programming languages. That made the tools not portable and very difficult to use across different software domains. Hence, developing software products under those software domains has a risk of missing quality assurance via software testing.

To make these tools available in those domains, one needs to understand the software testing concepts, processes and ultimately, its algorithms. This paper reviews all the knowledge above and derive a common software testing algorithms.

It reviews the history of software testing, definition of quality, software development lifecycles, and working environments that potentially influences the software testing requirements. Then, it reviews the technicalities of software testing such as test coverages, test approaches, like CFG coverages, unit testing practices approach. Lastly, it analyzes the existing test tools across different programming languages to form the common software testing algorithm.

The paper then discusses the derived common testing algorithms and some new findings. It concludes with further separate research requirements to ensure the algorithm is tested and stable for usage.

1. Introduction

Software testing begun since World War II era[6]. It was all the way back in 1967 where Herm Schiller created the first software code coverage monitor[7]. Today, software testing is mainly for systematically strengthens the stability of a software operations across different requirements[6], assuring the software performing up to a certain quality.

Similar to any software product, software testing tools are software products in nature. Like any industrial-ready or commercial software products, they undergo thorough testing before releasing to customer. However, there is one crucial problem: portability.

These tools are very specific to its software domains like the programing language, operating system etc. Therefore, not all test tools are cross-available in different software domains like programming language barrier such as javascript, and shell scripts[1][5]. This means any software product development under those domains are risking the absent of software testing and quality assurance.

To make them available, one requires to understand software testing knowledge like its concepts, processes, and algorithms. Unlike test tools, these knowledge are portable and transferrable. One can spin a test tools easily using said knowledge. That is the purpose of this paper: to investigate the common software testing algorithms.

In this paper, we reviewed the software development fundamentals like concept, lifecycles, and work environment to get an understanding of today’s software world. We then reviewed and analyzed the software testing technicalities like test coverage, test approaches, test processes, and readily available test tools. Lastly, we discussed the algorithm we derived from our learning and concluded the paper.

2. The Problems

As we develop software, we often overlook the fact that software test tools are always available for all domains. This assumption doesn’t happen in reality. There are 3 main causes triggered us to perform this research.

2.1. Programming Languages Barrier

One big problem with the software testing tools is their specificity to programming language of origin. This is valid as good test tools must be able to understanding the language in order to test thoroughly. In another word, it is a strength for the test tools.

Unfortunately, the greatest strength is also its weakness: portability. Since the test tools are rooted deeply inside one language, to use the same tools for another software in a different programming language is very troublesome. One needs to build a large scale programming language interpreters to make the test tools talk in the new language. This is known as the interpreters approach.

Using this approach creates the second problem: dependencies.

2.2. Unwanted Dependencies

If we opted for the interpreter approach, we created a minimum of 3 dependencies to the overall project:

1. The programming language interpreter
2. The software test tools
3. The software test tool original programming language

This creates unnecessary dependencies just to perform software testing. We now need to include 2 set of programming languages into the project and an additional interpreter layer. This leads to the 3rd problem: complexity.

2.3. Transferring Complexities

If we operate using the interpreter approach, we introduce unnecessary process complexities into the project as well. A bug caused by the interpreter process and not the project would take a longer time to root cause and investigate.

Despite that, it also means that the project now has a steeper learning curve to operate. This is especially painful for new comer in the development team and potentially creates unnecessary discussion and issues due to the said complexity.

2.4. Our Approach

Based on the problems we presented, we now have 3 course of actions:

1. Develop the test tool from scratch
2. Workaround with the interpreter approach
3. Choose a different programming language for the software

If option 3 is not available, we can either choose option 2 which is fast but problematic or option 1. Option 1 is the primary motivation for this research.

Hence, we want to create the common software testing algorithms based on the existing software tools. These algorithms are transferable across different software domains and allow us to quickly develop the necessary test tools.

To create the said algorithms, we have to understand the current software development world and software testing technicalities from an overview perspectives. This allows us to stay on-course with existing industrial practices.

Also, by understanding software testing from a holistic perspective, we can understand the current software development and testing requirements. This allows us to adjust the algorithms for meeting the current demands.

In the following sections, we present our reviews, analysis outcomes, the algorithms, and some discussions.

3. Overview - Software World

In this section, we review the current industrial practices and software quality management. We begin by studying the definition of quality. This is to clarify the meaning and expectation as we mention “quality”.

Then, we review the software development processes influences towards the software quality management. This allows us to understand the timing and to set test direction, test objectives, and compatibility for software testing.

Lastly, we review the work environments that have potential influences towards test tools’ requirements.

3.1. Definition of Quality

Definition of quality is subjective and carries high degree of perception. From years of software advancements, defining quality can be very philosophical. For example:

- Walter Edwards Deming, Walter A. Shewhart, Philip B. Crosby believes that **quality is manageable, measurable, and ultimately, conformance to specifications**[4];
- Armand Vallin Feigenbaum, Kaoru Ishikawa, and Walter A. Shewhart, believes that **quality means meeting customer needs**[4];
- Walter Edwards Deming and Joseph M. Juran believes that **quality that comes from organizational management, practices and processes**[4].

Moreover, the definition is also industry specific. Some industries like avionics, automobile, and medical have very strict software quality control due to any defect could kills life and create physical damage to the environment and ecosystem[9]. Others like banking, payment, and other finance software must comply to compulsory compliances, such as Payment Card Industry (PCI) Security Standards for payment card related softwares[11].

Based on the philosophies above, we concluded that "quality" means software testing that:

1. has detailed requirements from:
   1.1. customers
   1.2. standards compliances
   1.3. industrial standards
2. is measurable
3. is manageable
4. conformance to specification

We then review quality from 2 distinctive directions: software quality and data quality.

3.1.1. Software Quality Aspects

In this section, we review the aspects for measuring software test quality. This allows us to execute software testing in a manageable, measurable, directional, and empirical way.

Based on industrial practices, ISO 25000 defined software product quality into 8 primary aspects: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability[2]. These primary aspects hold their respective subsidiary expectations that serve as the testing objectives[2]. Table 3.1.1-1 to Table 3.1.1-8 tabulated all aspects accordingly[2].

Ideally, a software should achieve all quality aspects. In reality, developers often prioritize some expectations over another based on requirements[3]. This is influenced by various reasons like definition of quality differences, and/or resources constraints like short development time[3].

Table 3.1.1-1 - Functional Suitability Quality Aspect from ISO 25010

Expectations	Descriptions
Functional Correctness	Degree of function fulfilling all specified objectives.
Functional Completeness	Degree of function delivers result with accuracy and precision.
Functional Appropriateness	Degree of function facilitate accomplishment for all specified objectives.

Table 3.1.1-2 - Performance Efficiency Quality Aspect from ISO 25010

Expectations	Descriptions
Time Behavior	Degree of processing time required for producing results.
Resources Utilization	Degree of amount and types of resources used for producing results.
Capacity	Degree of minimum and maximum limit for producing results.

Table 3.1.1-3 - Compatibility Aspect from ISO 25010

Expectations	Descriptions
Co-Existence	Degree of performance stability when deployed in shared environment and resources.
Interoperability	Degree of information exchange with other products.

Table 3.1.1-4 - Usability Aspect from ISO 25010

Expectations	Descriptions
Appropriate Recognizability	Degree of recognizability for users to match their requirements.
Learnability	Degree of easiness for user in mastering the use technique.
Operability	Degree of easiness for user to operate and control.
User Error Protection	Degree of user error protection and prevention.
User Interface Aesthetics	Degree of pleasing and satisfaction from using the user interface.
Accessibility	Degree of support for wide range of characteristics and capability

Table 3.1.1-5 - Reliability Aspect from ISO 25010

Expectations	Descriptions
Maturity	Degree of requirements for meeting the system reliability confidence under normal operation.
Availability	Degree of operational readiness for use and consumption.
Fault Tolerance	Degree of operational steadiness in faulty environment.
Recoverability	Degree of self-heal and re-initialization for process, system, data, and state.

Table 3.1.1-6 - Security Aspect from ISO 25010

Expectations	Descriptions
Confidentiality	Degree of secrecy protection and controlled accessibility.
Integrity	Degree of access management and illegal access prevention.
Non Repudiation	Degree of logging and solicit evidence for events.
Accountability	Degree of tracing actions uniquely to an entity.
Authenticity	Degree of identity management and claims.

Table 3.1.1-7 - Reliability Aspect from ISO 25010

Expectations	Descriptions
Modularity	Degree of impacts for changes to its own and other components.
Reusability	Degree of modification required for deployment in more than one system.
Analysability	Degree of diagnostic capability and easiness in analysis.
Modifiability	Degree of modification easiness without degrading software quality.
Testability	Degree of effectiveness and efficiency for testing the system.

Table 3.1.1-8 - Portability Aspect from ISO 25010

Expectations	Descriptions
Adaptability	Degree of stability in different operating environment.
Installability	Degree of effectiveness and efficiency installation.
Replaceability	Degree of effectiveness and efficiency for product substitution.

3.1.2. Data Quality Aspects

Apart from software quality aspects, data quality is equally important since end-users consume data produced by the software. In this section, we review the aspects for measuring data quality.

Based on the industrial practices, ISO 25000 offers ISO/IEC 25012 standards for data quality compliances[8] depending on the data nature.

Data can be described in 3 distinct natures:

1. Inherit Data Quality - ***data under immediate consumption or use through control logics and controls***[8]
2. System-Dependent Data Quality - ***data depending on system for reachability and preservation***[8]
3. Inherit and System-Dependent Data Quality - ***data under the influences of both inherit data quality and system-dependent data quality***[8]

Hence, a piece of data's nature can be any of the 3. Table 3.1.2-1 - 3.1.2-3 all shows the expectations for the 3 data natures respectively[8].

Table 3.1.2-1 - Inherit Data Nature Aspect from ISO 25010

Expectations	Descriptions
Accuracy	Degree of correctness for data syntactically and semantically.
Completeness	Degree of fulfillment for all data attributes.
Replaceability	Degree of correctness free from contradiction and coherences.
Credibility	Degree of truthfulness, authenticity, and trustworthy.
Currentness	Degree of age and lifecycle.

Table 3.1.2-2 - Inherit and System Dependent Data Nature Aspect from ISO 25010

Expectations	Descriptions
Accessibility	Degree of reachability for wide range of user's characteristics and capability.
Compliance	Degree of adhering standards and specifications.
Confidentiality	Degree of secrecy protection and controlled accessibility, complying ISO/IEC 13335-1:2004.
Efficiency	Degree of performance and resources needs.
Precision	Degree of exactness.
Traceability	Degree of capability for tracking data changes.
Understandability	Degree of data consumption difficulties.

Table 3.1.2-3 - System Dependent Data Nature Aspect from ISO 25010

Expectations	Descriptions
Availability	Degree of reachability at the point of time.
Portability	Degree of freedom in shifting from one system to another.
Recoverability	Degree of preservation and operation quality enabling data self-healing.

3.2. Software Development Lifecycle

In this section, we review the Software development lifecycles (SDLC) influences to software testing and quality control. Software testing is an important part of the process in SDLC. Therefore, we must review the compatibility and feasibility of various software testing approaches.

SDLC is commonly categorized into few known stages: gathering requirements, plan and develop, test and validation, package and deploy/distribute[11][12][13][14][15]. However, they are all different in terms of execution across different process model.

3.2.1. Ad-Hoc

Back before cross-compiling languages like C exist, software development and deployment is very limited[8]. The software requirements are very simple that it doesn't require much testing[8]. Moreover, development efforts are performed in ad-hoc manner. Since there is no need for a process to govern the development[15], this SDLC is known as "ad-hoc development".

The advantages for adopting this SDLC is obvious: low to no overhead, adaptable, and easy to use[15]. Developers can jump straight into development without much consideration. However, the caveats is that it ignores important values like maintenance, testing, scaling, recoverability, and possibly becoming non-reusable[15].

This SDLC is commonly seen in experimentation in the industrial or commercial practices. It is useful for developer to verify some light-weight assumptions where the software product is either one-time or limited use.

Since ad-hoc development process model excludes testing explicitly, software testing is not required in this SDLC.

3.2.2. Waterfall

Waterfall SDLC model started in 1956, first documented by Benington, modified by Winston Royce in 1970[14]. It is a cascade model where one stage is fully completed before transitioning to next stage[12]. Diagram 3.2.2-1 illustrates the general process model.

The advantage of implementing waterfall process model is that the tasks and specifications are rigid and detailed so execution is far straightforward[15]. This type of process model works well in handling complex but well-understood project, and friendly to inexperience member in the team[15].

The downside is that it is very difficult to predict the precise and accurate requirements upfront, in the requirement specification stage[12][15]. Secondly, the SDLC structure is very rigid to the point where the next stage can't start without the previous getting completed[12]. Hence, if there is a requirement change during the production, there is no way to accommodate such changes[11].

Waterfall model has been deployed in structural and stable project development, software and hardware environments[12]. This is especially noticeable in the industrial and military sector where many standards like MIL-STD2167A, MIL-STD 498, IEEE-STD-016, and ISO 12207[13].

Executing software testing in this SDLC is straightforward since itself is an independent process block. Test developers only start developing test plan in the testing stage based on the inputs from the requirements, development, and implementation stages[11]. During test stage, all roles from business analyst, user experience testers, engineers, and quality control experts perform their test together[11]. Upon stamping a "pass" in test phase, the product can be safely said a releasable to customer[11].

Based on Diagram 3.2.2-1, we can see that the software testing stage only starts at Week-25 of the entire project timeframe. While this is good if the project develops its own test tool, this is not feasible since testing the product is difficult due to large complexity. This means that the test tool will need to be equally complex and becoming not portable for other project. Moreover, if there is a change in requirement, the entire product development undergo an overall reset and likely going through a project restart.

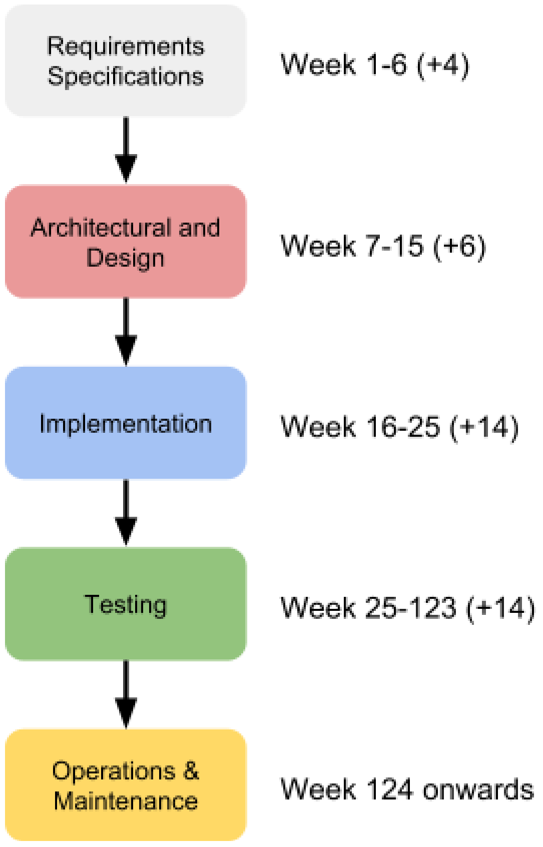


Diagram 3.2.2-1 Generic Waterfall Process Model

3.2.3. Agile

As the software industry advances to the point where requirements are changing faster than the development does, numerous new SDLC models are born. These fast, adaptable models are categorized under the common name "Agile". Agile SDLC models are a series of waterfall processes focusing on task or feature level instead of project management level[11]. Their manifestos are[12]:

1. Individuals focused, not tool
2. Working software over documentation
3. Customer collaboration over contract negotiation
4. Embrace changes

To do that, they comply to a list of strict discipline[12] like:

1. Early and continuous delivery
2. Maintain delivery consistency
3. Prioritize technical excellency
4. Maintain simplicity (art of achieving goal with maximizing work not done)
5. Welcome continuous requirements (embrace changes)
6. Perform self-improvements (kaizen)

A generic agile process is shown in Diagram 3.2.3-2.

There are various agile process models based on this incremental development concept[13]. Some examples are Scrum, Unified Process Model, Crystal, Feature Driven Development, Adaptive Software Development, eXtreme programming[12], Six Sigma[4], Rapid Prototyping Development / Rapid Application Development[13][14], Joint Application Development[13][14], and Lean Development[14]. As there are various SDLC models to choose, many suggested to considers a series of inputs before choosing one[15]:

1. Tasks at Hand
2. Risk Management
3. Quality or Cost Control
4. Predictability
5. Progress Visibility
6. Customer Involvement and Feedback

In terms of value, the process must consider the feature, time, quality, and resources cost[4][11][15]. Table 3.2.3-1 shows the rating for some processes based on the inputs above[15]. Diagram 3.2.3-3 illustrates the feature, time, quality, and resources cost triangle. These values however, are only as selectable since their extremes contradicts one another[15].

Software testing for any agile process must comply to their manifesto and execution. Since the testing is inside each cycle of development[11] and acting as quality control backbone, it must be modular, fast, non-brittle and portable. Due to fast and changing requirements nature, test techniques such as acceptance test, pair programming, and test-driven development are primary habits in these processes[12].

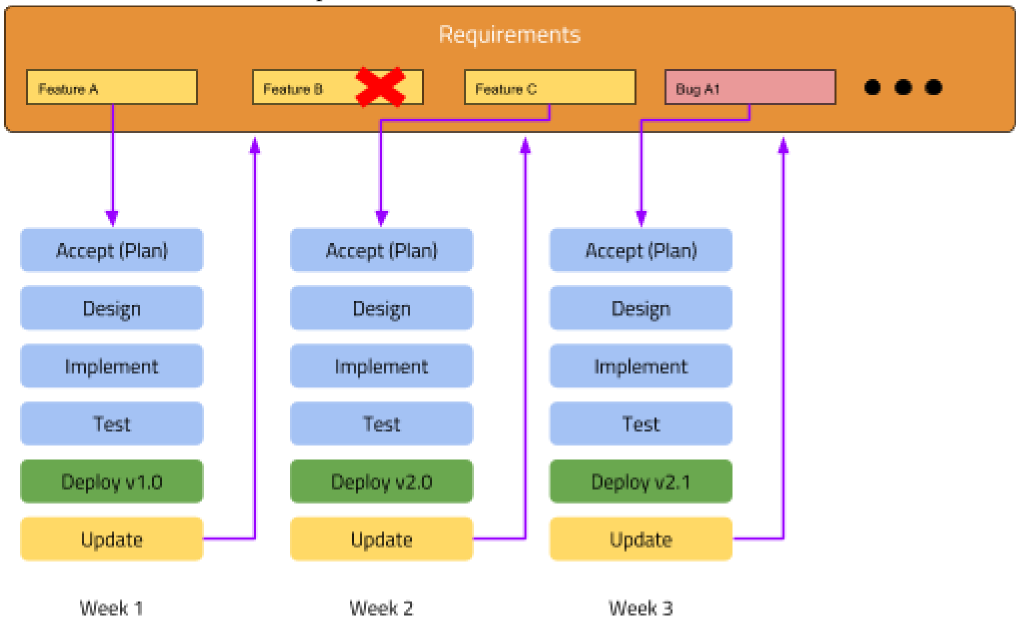


Diagram 3.2.3-2 Generic Agile Process Model

Table 3.2.3-1 - Model Selection based on Input Rating[15]

Model	Risk Management	Quality Control	Predictability	Progress Visibility	Customer Involvement
Ad Hoc	1	1	1	3	2
Waterfall	2	4	3	1	2
Spiral	5	5	3	3	3
Rapid Application Development	3	3	2	5	5
Feature Driven Development	3	5	3	3	4
Design to Schedule	4	3	5	3	2

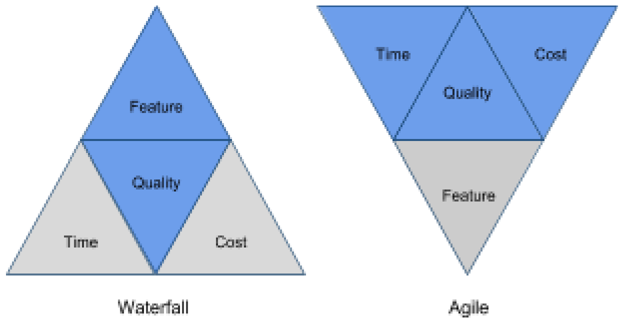


Diagram 3.2.3-3 - Software values triangles prioritization where left is waterfall model, right is agile.[11]

3.3. Work Environment

In this section, we review the impacts and influences of a work environment towards software testing. This allows us to gain understanding about the work environment influencing the nature of the test tools being built or modified for suiting a particular environment.

3.3.1. Remote-Only

Today, software development and testing has reaches a point where physical office is no longer a requirement. Most infrastructures are now facilitated via cloud services like software-as-as-service (SaaS) such as GitLab CI[41], or infrastructure-as-a-service (IaaS) such as DigitalOcean[42]. Also, the current technological enablements such as fast internet everywhere, 100Mb/s+ cable, 5GHz Wifi, 4G cellular, video conferencing software, virtual office and infrastructures[16] made this work environment feasible.

In order to achieve remote-only work environment, the nature of the product must promote high portability. Out of the 37 successful companies listed in remoteonly.org, all of them are producing software products that does not involves physical resources for logistic delivery[16].

Their testing methodologies also reveals that their testing technologies relies heavily on several factors such as output type, portability, learning-on-the-job, and team communications[16]. Without these attributes, software testing can results in a non-productive way[37]. Hence, these factors affect software testing and automation tools development from time-to-time. A good example is the continuous integration (CI) infrastructures development.

Before GitLab, CircleCI was the best choice of service for facilitating CI services[43] on Github. However, it is limited to non-physical products that are suitable to run inside a container[43]. This is not feasible for physical based products such as firmware or electronics, and security related tasks like key signing, encryption, and software packaging. Moreover, CircleCI requires external integration with GitHub, which means extra processes and security concerns when dealing with sensitive data.

GitLab foresaw such issues and developed their own GitLab remote CI robot to facilitate all testing tools across any SDLC. This allows users to have full freedom in their development and testing[41]. Additionally, it facilitates a more flexible development environment and testing friendly infrastructure, mitigating sensitive data security concerns[41]. To date, GitLab CI automation services had included the Kubernetes production deployment[41], leaving developers to focus on only development while the automation takes over build, test, package, deploy, and monitor[41].

3.3.2. Office-Only

Office-only work environment requires the tester and testing infrastructure to work only inside a physical premises on-site[36]. Normally, a company enforces office-only work policy company wide due to various reasons:

1. Policy abusement by employee[36][37]
2. Job nature and social requirements[36][37]
3. Security and intellectual properties protection[38]

This environment allows tester to not only perform software testing with great freedom, it also offering an easier testing method for physical products[16]. Since the test infrastructures cannot leave the premise, high portability and remote facility requirements are no longer mandatory and sometimes, banned for security reason[38].

3.3.3. Mixed Environment

The last work environment is a mix of both remote and office work environment. It is commonly known as “work-from-home” policy[40], or “distributed team” culture[39]. Many companies implement this type of work environment like InVision[39] and Accenture[40].

Since this work environment promotes remote access, the test infrastructures and methodologies should aligns to “remote-only work environment” due to its complex requirements and challenges.

4. Overview - Software Testing

In this section, we review software testing from the technical perspective. This allows us to understand software testing from a software developer or tester point of views.

We start off by reviewing the qualitative aspects of software testing and their associated technicalities. Here, we will learn the test coverages, requirements, expectation, and the testing objectives for measuring test quality.

Next, we review the software testing approaches available in the industry. This facilitates us with approaches to achieve the software testing coverage goals.

After reviewing both test coverage and testing approaches, we then analyze the existing test tools in its dedicated analysis section.

4.1. Test Coverages

Test coverages is a quality measurement for software testing. It is achievable via 2 main test methodologies: white-box testing and black-box testing[17][18]. Developers can deploy either or both methodologies simultaneously in their testing strategy[17].

Black-box testing methodology focuses on testing at the overall application/system level with a set of inputs against the expected outcome without considering the application’s interior working functionalities[18]. It is easy to implement with low learning curve, develop test cases quickly, and by its own nature, simple[18]. The caveat however, is the lack of in-depth of the test analysis and create a large coverage challenges to cover the required stability[18]. Some examples are product behavioral driven testing and system level testing.

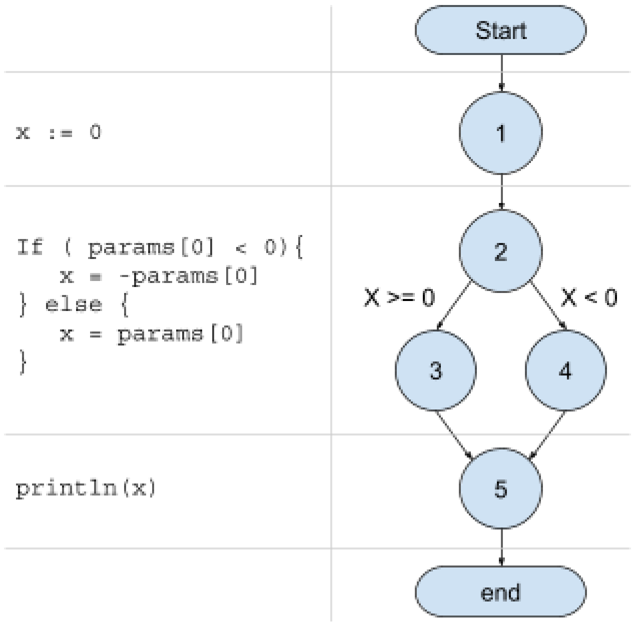
White-box testing methodology focuses on testing the application’s interior working functionalities[18]. Due to its ability for accessing the application internal functionalities, it executes tests with a more introspective, thoroughness, and stability comprehensions.[18]. The downside however, is that it introduces a high-level of application-specific integration, making it not portable; the complexity for all the test scripts; and creating a fragile or brittle test environment[18]. Some examples are unit testing or product functional testing.

Both methodologies have their pros and cons. One must prioritize the value of using them in the project to achieve large test coverage[18]. We will review the different types of test coverages and approaches before analyzing test tools.

4.1.1. Statement Node CFG Coverage

Statement node control flow graph (CFG) test coverage is a white-box testing coverage criterion[17]. It focuses on a hypothesis where defect is discoverable if containing code parts not executed[19]. With this coverage, It minimizes the number of test cases required for achieving maximum coverage.

Diagram 4.1.1-1 illustrates an example for statement Node CFG coverage. In this example, if the test cases only cover 1-2-4-5 node progression, node 3 is not tested and therefore has defect possibility[17][19]. This implies that if we implement only this coverage alone, it creates a coverage incompleteness[17]. Therefore, it needs other criterias to overcome this limitation.



```
x := 0

If ( params[0] < 0){
   x = -params[0]
} else {
   x = params[0]
}

println(x)
```

Diagram 4.1.1-1 - Node mapping comparing to source code

4.1.2. Edge CFG Coverage

Edge CFG coverage is another white-box testing criterion[17]. It focuses on the range of parameters and conditions[17]. Back to Diagram 4.1.1-1 example, if we feed params[0] with edges data like:

- A maximum negative number
- A minimum negative number
- Zero
- A minimum positive number
- A maximum positive number
- Type of data - round number
- Type of data - decimal float number
- Type of data - any others like string

It covers the spectrum of inputs for node 2. This triggers the necessary node 3 and 4 control flow and raise error accordingly. Therefore, it helps overcoming the statement nodes CFG coverage's limitation. Edge CFG Coverage works hand-in-hand with Statement Node CFG Coverage almost all the time.

4.1.3. Condition CFG Coverage

Conditional CFG coverage is another white-box testing criterion, focusing on controlling all possibilities for multiple conditions[17]. Diagram 4.1.3-1 is an example for the "if" condition which has two independent comparisons.

```
x := 0

if (params[0] == "" &&
    params[1] == "2")
{
  x = 1
}


println(x)
```

```
x := 0

if (params[0] == "") {
  if (params[1] == "2") {
    x = 1
  }
}

println(x)
```

Diagram 4.1.3-1 - Multiple condition coverage

Based on the example in Diagram 4.1.3-1, developer commonly writes the code shown on the top[17]. Both top and bottom cases create a total of 4 cases to test. However, when it comes to test execution, the code on the top for params[1] gets neglected when params[0] is false. This yields the coverage truth table shown in Table 4.6.3-1.

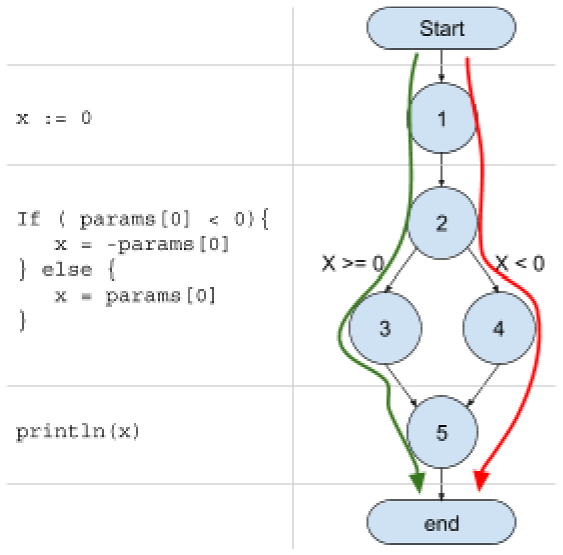
Table 4.6.3-1 - Truth table for condition coverage

ID	[0]	[1]	Condition Coverage
1	True	True	True positive
2	True	False	negative by params[1]
3	False	True	negative by params[0]
4	False	False	negative by params[0]

Complying to statement node CFG coverage criterion, the negative execution codes for params[1] is true is considered not covered since that path is blocked out by params[0] false condition[17]. Condition CFG Coverage looks into such matters and create the necessary coverage points.

4.1.4. Path CFG Coverage

Path CFG coverage is another white-box testing criterion. It identifies the desired control path for each scenarios[17].



```
x := 0

If ( params[0] < 0){
   x = -params[0]
} else {
   x = params[0]
}

println(x)
```

Diagram 4.1.4-1 - Path identification coverage

The purpose is to identify and to ensure critical paths in execution are done safely[17]. Hence, it normally deals with multiple cases conditions, loops and recursive functionalities, ensuring all the edge conditions are covered for the paths[17]. Diagram 4.1.4-1 shows the path identifications for program illustrated in Diagram 4.1.1-1.

4.1.5. Equivalence Class Testing

Equivalence class testing (ECT) is a black-box test coverage criterion[17]. It focuses on providing complete application testing and avoiding redundancy[17].

To execute a strong ECT (known as SECT), the developer should covers all parameters for each scenarios[17]. Weak ECT (known as WECT), the testing by cherry-picking critical parameters for testing representing the overall test coverages[17].

Both SECT and WECT has their pros and cons depending on requirements. WECT covers all extreme cases to ensure the application works within the boundary of parameters; SECT covers stronger stability testing. If error handling is a priority, developer can then expand the WECT to SECT where it evaluates more test values and expected errors[17].

There is a problem with ECT: the parameters can be infinite[17], taking infinite time for test execution. This is mitigatable by using various other black-box test coverage criterion.

4.1.6. Boundary Value Analysis

Boundary value analysis (BVA) is a black-box test coverage criterion working alongside with ECT. It systematically tests the boundary values for WECT testing[17], focusing on all possible parameters falls on the the acceptance boundary[17]. In short, it enriches the WECT by covering not only its parameters but establish a value boundary near them for error handling[17].

4.1.7. Category Partition Testing

Category Partition Testing is a black-box test coverage criterion working alongside with ECT. It categorizes the approach for ECT[17]. Since ECT has an infinite parameters to cover, Category Partition Testing coverage categorizes them in to groups[17]. This helps in scoping down critical parameters based on a characteristized conclusion, making testing robust yet compact to achieve large coverage[17].

Example: for an array sorting function, we can characterize the test parameters into:

1. Length of array
2. Type of elements
3. Boundary values (over max, max, ideal, min, below min)
4. Mix of boundary values with length of array.

Once the categorized coverage item like character 4 is achieved, the test coverage can draw a conclusion that the said array now supports mix of boundary values with length of array[17].

4.2. Test Approaches

In this section, we review some known test approaches that achieves the test coverages. These approaches are known as the software testing practices in the industry.

4.2.1. Unit Testing Practices

Unit Testing Practices is a test approach focusing testing procedures/modules at the smallest unit possible[19][20] objectively[20]. Unit testing produces fast feedback through their independent test execution[20]. Therefore, It is highly suitable for regression testing, especially for continuous integration and delivery[20].

The downside however, is that unit test can be challenging when it comes to graphical user interface testing, test scripts reusability, lack of documentation, developers' competency, and cost over value[20]. Some developers or organization views unit testing is a costly process as far as taking 50% developers' extra time from development timeframe[21].

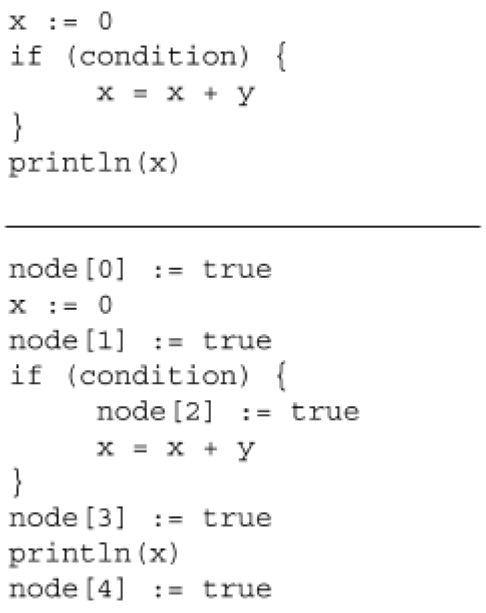
Developer usually produces the unit test codes alongside writing or debugging the product[21]. There are various techniques such as stubbing, mutation analysis, and mocking used to produce the unit test[21].

Some example like in object-oriented classes, the unit test for said class is called class testing[19].

4.2.2. Source-to-Source Transformation

Source-to-Source transformation is an approach to modify the source code for catering code coverage analysis[1]. It injects the source code with the statement node based on the test coverage for later test coverage analysis[1].

Diagram 4.2.1-1 illustrates an injection example: the top section is the original code; the bottom is the node injected code. Based on the example, as the test is running, these executed node is set to "true" which means it went through this path. After the test execution, the system can analyze the executed nodes for the designated coverage points.



```
x := 0
if (condition) {
      x = x + y
}
println(x)
```

```
node[0] := true
x := 0
node[1] := true
if (condition) {
      node[2] := true
      x = x + y
}
node[3] := true
println(x)
node[4] := true
```

Diagram 4.2.1-1 - A source-to-source transformation

4.2.3. Industrial-Strength Transformation System

Industrial-Strength Transformation System is an approach of using industrial commodity test tools for analyzing code coverage[1]. These engineering tools have specialized source-to-source rewrite rules and facility to confidently inject the probe and analyze on-the-fly. Normally, these tools are specific to programming language, especially mainstream languages like C, C++, Ada, FORTRAN and etc[1].

Among the known commercial tools are REFINE from Reasoning Systems, XT from Program Transformation Organization, and DMS from Semantic Designs[1]. Many compiler now offers ad-hoc parser modifications feature, allowing developers to perform source-to-source transformation configuration easily[1].

4.2.4. Execution Log Tracing

Execution log tracing is an approach that parses the execution traces for test coverage analysis[5]. Certain programming languages like Ruby, Python, and Shell do output their respective line-to-line execution traces[5] for facilitating software testing.

There are some known softwares such as shcov or kcov which already using this implementation method[5].

4.2.5. Static Analysis

Static analysis is an approach focuses on analyzing the code/object itself for defects[22]. Depending on the language, this approach scans for security vulnerability such as buffer overrun, unvalidated input, memory referencing like null dereferencing and uninitialized data, resources leak like memory or OS, API violation, exception handling, encapsulation and race conditions[22].

Static analysis is frequently deployed not only in conventional software application but critical industrial systems like A380 fly-by-wire control system[23]. This type of tool uses different abstraction interpretation like AbsInt's worst-case execution time analysis, CEA's Fluctuat, Astr´ee etc[23].

Static analysis is implementable via various methods like abstraction, AST walker, type analysis, lattice analysis, etc[22][24]. However, it has its limitation such as frequent false reporting, poor test exception handling, and modularity dependency[22]. Therefore, developer must use other approaches to complement said limitations.

5. Analysis - Test Tools

In this section, we review each of the software testing tools available across different system and languages. We look into the test tool implementation, learn their test coverages, and understand their test approaches.

Then, we review the gaps we identified through our analysis and compare them with our problem statement.

5.1. Test Tools

In this section, we review some currently deployed test tools from its purpose, supported language, supported test coverage, and its test approaches.

5.1.1. Valgrind

Valgrind is a dynamic analysis tool primarily focusing on C/C++ program[25]. Valgrind is able to monitor various memory usage through its memcheck, cachegrind, callgrind, and massif[25].

Valgrind uses the industrial-strength transformation system approach by adding itself as a service layer between the program and the operating system[25], shown in Diagram 5.1.1-1. It employs statement and condition CFG coverage throughout its own stepping. As for edge and boundary value analysis, Valgrind requires user to provide the correct arguments in order to test it.

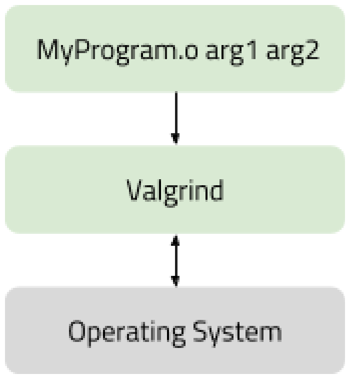


Diagram 5.1.1-1 Valgrind Interaction

Since Valgrind is working in between the program, it has the ability to inject memory creation call functionalities to analyze the memory usage[25]. This is the main reason why program run under Valgrind takes 20~30 times longer to complete[25]. Valgrind is able to achieve this mainly because the C/C++ language itself permits such interfaces.

5.1.2. Shellcheck

Shellcheck is a static analysis linter created using Haskell language for bash and shell scripts[26]. It uses static analysis approach to analyze faulty codes. Hence, it relies heavily on line-by-line abstraction for its analytic work.

Unlike other tools, shellcheck only checks on coding practices and reports potential defective codes[26]. Therefore, it is common that shellcheck reports numerous false positive since it doesn't execute the program on its own.

5.1.3. Rubocop

Rubocop is a static analysis linter for ruby programming language[27]. Similar to shellcheck, Rubocop uses static analysis approach to read the source codes itself and reports potential defects and coding style or formatting irregularity[27].

Due to the high flexibility in ruby language, Rubocop is able to perform keywords abstraction analysis upon the source code[27]. Then from there, it compares each line of codes against its huge set of linting libraries[27].

5.1.4. MinUnit

MinUnit is a unit-test framework for C language similar to CPPUnit[28]. MinUnit uses the unit-testing practices approach by providing 3 lines header to illustrate the simplicity for implementing unit testing[28]. It relies on the C language static/dynamic linking capability to perform white-box testing against the main program.

Since this is a unit-testing practices approach, it allows test developers to perform all the test coverages depending on their competency. Optionally, developer can use industrial-strength transformation system approach for the unit-testing.

The idea is to have the main program's functions packed as a library and then have the unit test program runs like the main program. Upon compilation, developer will just need to run the unit test program once and capture the output.

MinUnit successfully demonstrates the unit test only requires a minimum of 3 statements:

1. The assert statement
2. The run test statement for running all tests
3. Test return values

User can then expand the framework based on needs.

5.1.5. Rspec

Rspec is a unit testing framework for Ruby language[29]. Hence, it employs unit testing practices approach for developer to test their program. Rspec executes on an independent ruby test script that import the main script. The test script contains the test cases written using test statements style, similar to MinUnit[29].

As a framework, Rspec relies heavily on Ruby's built-in modules, class functionalities[29], and external libraries such as simplecov to analyze the test coverage[30].

5.1.6. Simplecov and Ruby Coverage Module

Simplecov is a code coverage reporting and interfacing feature for Ruby built-in Coverage module[31]. Hence, it uses industrial-strength transformation system approach to facilitate code coverage analysis.

Ruby built-in Coverage module also uses execution log tracing approach for test coverage analysis[32][33]. The module is able to measure the path, conditions and edge CFG coverages[32][33].

5.1.7. Unittest

Unittest module is a built-in python based unit testing framework[34]. Therefore, it uses unit testing practices approach to facilitate testing.

It leverages python object-oriented class to package a test suite with its test cases[34]. After that, it has the test script self-executable using the unittest class's main function.

5.1.8. Coverage.py

Coverage.py is a Python code coverage test framework[35]. It uses its industrial-strength transformation system approach like Python inherited line number table and execution log tracing approach for test coverage analysis[35]. Also, Coverage.py uses static analysis approach to analyze codes for defects. Comparing to other tools, Coverage.py is a very robust test tool.

Coverage.py also specified that using it can significantly slows down the execution time[35]. To compensate this, part of the analytic tools is written using C language[35].

5.1.9. Go Test Tool

Go test tool is an all-in-one test tool for Go programming languages. It relies heavily on its industrial-strength transformation system approach such as its go interface features for test injection[45]. It also uses execution log tracing approach to test coverage analysis[45]. Go test tool also uses unit testing practices approach to facilitate the conventional test executions.

One thing special about go test tool is that it facilitates performance testing alongside its unit testing feature[44][45]. This allows the user to do performance testing down to the functional level. It uses statistical average calculations by running the test across many repeats, as high as 1 million[44]. Therefore, the produced results are confidently consumable. However, this performance testing is subjected to noises such as CPU temperature throttling which might not produce accurate results[44].

5.2. Gap Identified

Throughout our analysis, we observed our addressed problems are reflecting across all the tools: they are great testing tools in their programming language/operating system environment, but not portable across one another.

For example, Valgrind, being able to analyze memory management capabilities for binary execution is only specific to C/C++ binary program. It can't be directly used against softwares from other programming languages due to design limitation.

Similarly, Shellcheck and Rubocop are static analysis test tools but can't be used across each other's language domains. This made sense since static analysis test tool needs to scan the codes directly for defects. Hence, they are not portable and must maintain its programming language specific nature. However, they share common processes like "check, clean, parse, analyze, report" for each code analysis.

As for test coverage, Simplecov and Ruby Coverage module, and Coverage.py all exhibit the same language specific gap. Coverage.py is not usable for Ruby language and Simplecov is not usable for Python language. Their internal test coverage processes share the same pattern: both are using log tracing approach and industrial-strength transformation approach to execute their test.

Looking at unit testing test tools, MinUnit, Unittest, and Rspec are the 3 main tools. They are all specific to their own programming languages and not portable across one another. However, they follow the same implementation processes: parse the source code, setup the test environment, execute fragment of unit test codes, process the results, and repeat them again until all fragment of unit test codes are ran.

Another observation is that all tools are grouped in a purpose-specific manner instead of one test tool for one language. As shown above, Valgrind holds its own purpose of testing; Shellcheck and Rubocop are grouped as static analysis test tools; Go Test Tool, Simplecov, and Coverage.py are grouped as a code coverage test analysis tools; Go Test Tool, MinUnit, Rspec, and Unittest are grouped as unit testing tool; and Go Test Tool is grouped for performance testing.

One exception observation is that go test tool packed all the purpose-specifics test functionalities into 1 single tool. This is a good trait as the user would only use a single tool to run all the appropriate tests.

From the above observations, we can see our problem statement reflected on all tools. Since we can't use the existing test tools for other programming languages, we can extract the common processes and document it as the software testing algorithms.

During the course of the research, there is no detailed studies related to power management of the test system. This is different from the power management performance metric in performance testing algorithm.

The reason is that all the test executions are assumed running on a single operating system process, complying to the operating system default power management settings. This works for application level testing but can be troublesome for low-level testing such as kernel and hardware.

Also, there is no study related to the test tool's power management affecting the test results. This also means that any test executions studied in this paper should be carried out in a non-interruptive power-management operating system.

Therefore, a further study is advisable for this specific power management topic.

6. Discussion

In this section, we proceed to discuss our identified algorithm extracted from the analyzed test tools. This is the section where we consolidate the work patterns into a common software testing algorithm.

Then, we discuss about new findings like performance testing, power management, parsing capabilities etc. These findings are not detailed in this paper.

6.1. Extracting Method

Since software testing is not something new, we can perform reverse engineering on existing industrial-ready test tools and learn their testing algorithms. Then, we compare their processes with one another. If one tool is performing in an usual way, we will study the reason behind it and adjust accordingly. Once a common algorithms is visible, we then extract it.

This method of extraction is preferred mainly because:

1. Each tools' algorithms can validate one another during comparison.
2. Not to remove their development insights over the years
3. Not re-inventing the test methodologies.

Through this method of extraction, it assures our derived algorithm is not an reinvention and it is safer to deploy compared to creating algorithms from scratch.

6.2. Derived Software Testing Algorithms

By analyzing all the test tools, we observed a similar pattern for all industrial test tools. There are 5 recognizable process stages:

1. Prepare
2. Parse-and-Inject
3. Execute
4. Analyze
5. Report

The details for each stages are in the subsections.

6.2.1. Prepare

This stage prepares the testing environment. It is responsible for setting up the test directory, assemble the test files and test scripts accordingly, read user's configurations such as test exceptions and test parameters, and setup the framework data storage. Diagram 6.2.1-1 illustrates the process flow for this stage.

The first step is to process the test command arguments. Since the full test is a time consuming process especially test coverage, the algorithm should bail out for any improper arguments and exit with error.

The next step is optional. Taking consideration in some test tool for using a source-to-source transformation approach, this stage can optionally prepare an isolated test directory if necessary to prevent permanent modification effect against the original source codes.

The following step is to prepare or update its test data storage. This includes preparing a clean storage and logs location for storing the captured test data. If the test tool supports artifact caching, this is the step to restore the artifacts to the right location.

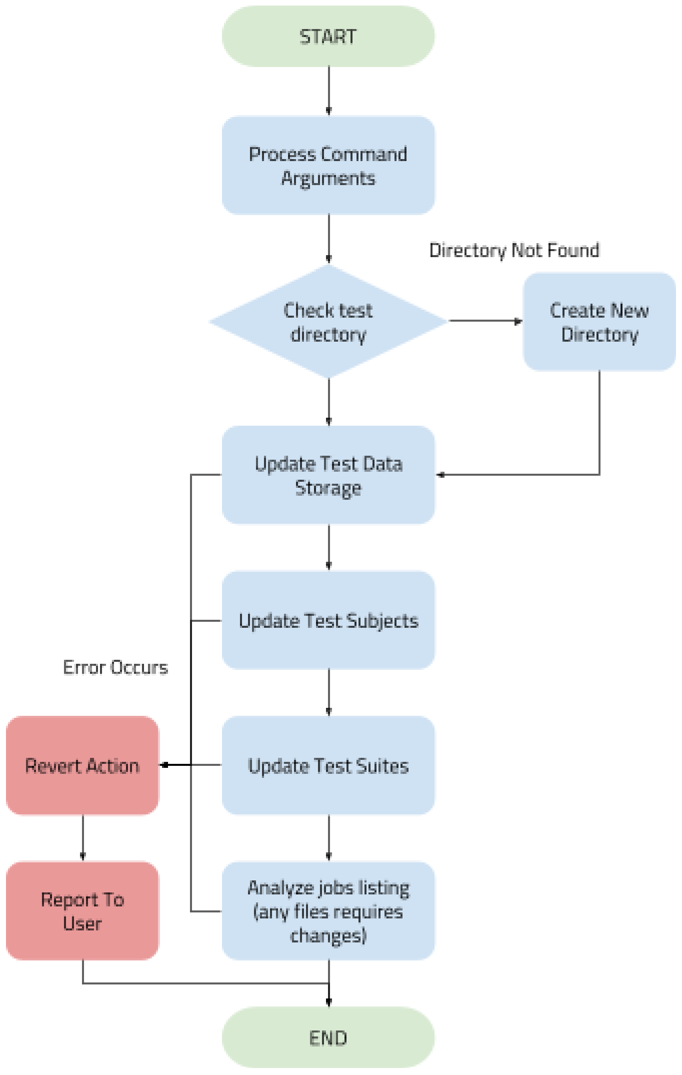


Diagram 6.2.1-1 - Process Flow for Prepare Stage

If the test directory is isolated, the next step is to update the test subjects and test suites by copying into it, reassembling a clone software repository for test execution. Otherwise, this step is skippable.

Lastly, this step is for optimization purposes. It is optional depending on the test tool requirement. The test tool can analyze changes for each files, identifying what has altered and what can be skipped (for saving time), making the test execution transactional. This saves time and resources.

If there are any error occurs at this stage, the entire test execution should comes to a halt and request user attention for solution. The is because the test environment is not prepared properly which can disrupt the test results.

6.2.2. Parse-and-Inject

This stage processes the test subjects for implementing test coverage. It is responsible for parsing and injecting statement nodes into the test subject. Diagram 6.2.2-1 illustrates the process flow for parse-and-inject stage.

This stage is time and resources consuming. Since some test tool warned that the overall slowdown can be as much as 20~30 times[25], the focus is to execute efficiently but prioritize doing the right thing.

The first and second steps are to find the possibility of skipping a test coverage analysis. This includes reviewing user's explicit instruction for test coverage or detecting negative parameters like no test cases scenarios. This way, the entire stage is skippable.

The next step is to check whether the code or language has an industrial-ready transformation facility for parsing and injecting the statement nodes. Example, for Ruby programming language, there is a built-in Coverage module. If such facility is available, the test tool should use such facilities instead of reinvention. Otherwise, the tool needs to perform source-to-source transformation approach for facilitate test coverage.

Source-to-source transformation approach alters the source codes by injecting executable "nodes" into each identified statements based on the CFG coverage. Therefore, the approach starts by duplicating the test subject. If the repository is duplicated in the preparation stage, this step is skippable.

Next, the approach parses the code line-by-line, analyzes its meaning, and then prepare the node number incrementally. The node itself is an executable function only reports to test coverage analytic tools; it must not alter test subject and test scripts original execution intention. As a best practice, this injection should be invisible and doesn't requires developer to inject manually in each test script.

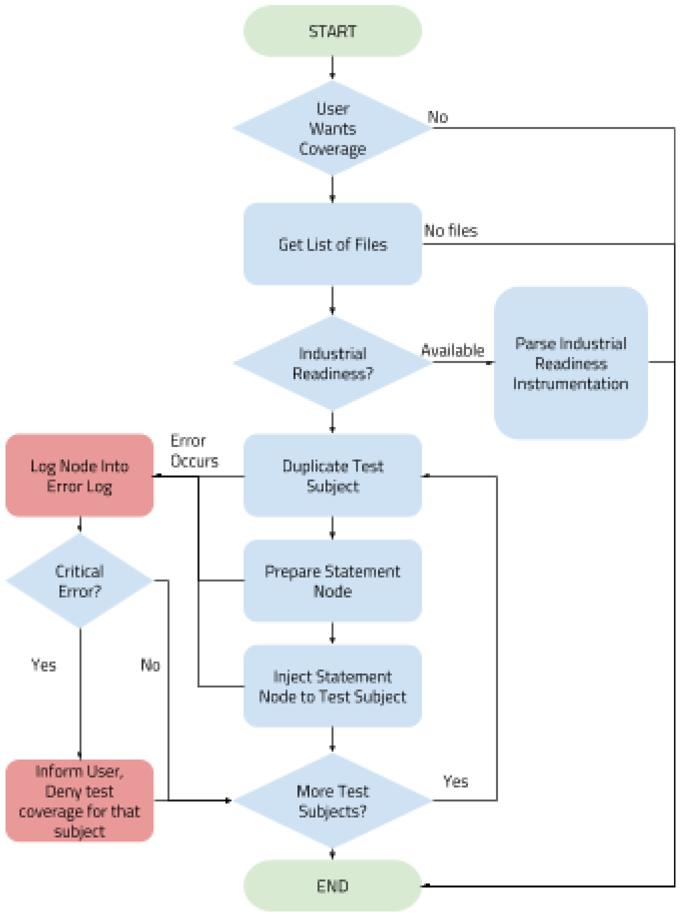


Diagram 6.2.2-1 - Process Flow for Parse-and-Inject Stage

During the line-by-line parsing, the test tool can perform numerous test coverages at a time. Since the tool is injecting statement modes, Statement Node CFG coverage is automatically implemented. Additionally, the test tool can perform path CFG coverage, condition CFG coverage, edge CFG coverage, static analysis, and boundary value analysis coverage simultaneously, depending on the test tool design requirements.

By simultaneously analyzing each line at a time, this saves time and resources since we are running the parsing execution for multiple coverages once compared to multiple tools running sequentially.

However, we have to keep in mind that the machine running the test tool must be able to facilitate such processing power for multiple simultaneous analysis.

Once the node is ready, the next step is to inject the node into the source code: next line of the analyzed line. This line-by-line parse and inject analysis is repeated for all the line of codes in a test subject.

Lastly, the transformation repeats itself for other test subjects until there is no test subject left. This signal that the stage is completed.

If any error or question occurs during this stage, the error is recorded into the error log. Then, the test tool developer needs to decide its severity and handling. For example, say the error is critical syntax error for 1 of the source codes, the test tool can deny running the test for that source code file and inform user later on for amendment via the error log. Then, the test cool continues to evaluate other source codes files. Doing this allows user to gain a broad view of errors across multiple source codes in a single run.

Contrary, if the test tool detects a best practice syntax writing issue but the execution is fine, the test tool developer can decide to record it as information log. Since it is a non-blocking issue, the tool can proceed to run test coverage for that source code.

As we study the test tools and derived the algorithms, we notice that one of the critical component for developing a new test tool is parsing capabilities. These capabilities must be thorough and smart enough to provide input for the test tool to generate necessary nodes.

Not only the parsing capabilities must understand a human-written codes, it must be able to analyze and to understand the code statements in order to determine the insights within it. Using these insights, the tool then is able to generate the coverage nodes or provide verdict for the code statement in static analysis.

Without a good parsing capabilities, it would be very tough to implement test coverage. However, individual test cases testing is still executable since they are not involved for statement node processing.

6.2.3. Execute

This stage is executing the test scripts There are various way to execute the tests. The common implementation is unit testing. Diagram 6.2.3-1 illustrates the process flow for this stage.

The idea is to execute each test case based on their individual test scripts. If the industrial-strength transformation approach tool is available, like Rspec, MinTest or Unittest, the test tool can always utilize them. Otherwise, the tool will need to execute each test scripts manually.

For manual testing, the test tool begins with gathering and sanitizing the list of test scripts. Optionally, if there is a denied list generated from the previous stage, this step can cross check filter the source codes out. If the final list is empty, this stage is then concluded passed with no available test.

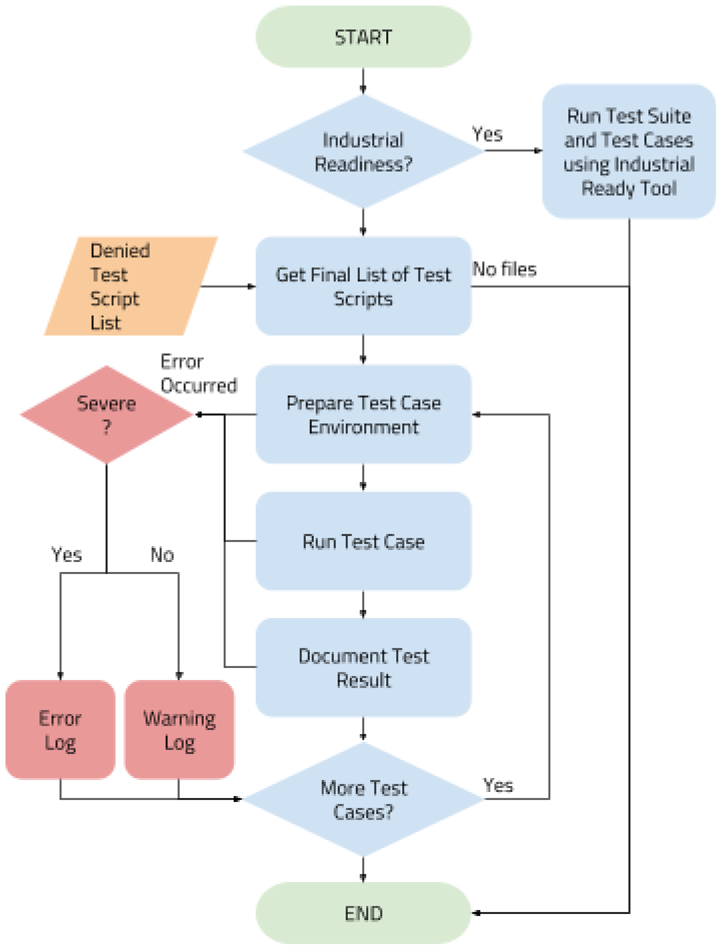


Diagram 6.2.3-1 - Process Flow for Execute Stage

The next step is to prepare the testing environment. This means that the test tool must be able to prepare an isolated test case environment before running the test case. This includes:

1. Deleting previous test case environment regardlessly.
2. Clear and release all previous variables and resources.
3. Create a new test case environment.
4. Assemble and configure the test case environment per instruction.
5. Calibrate result parsing functionalities.

Once the preparation is completed, the next step is to run the test case. For easiness, each test case should always produce result in a constant pattern. This simplifies the later result parsing capabilities. The next step is capturing the result data and record it into the test data storage. The process repeats itself until all test cases are executed.

If there is any error occurs in this stage, depending on severity, it should be recorded into the error log or warning log without disrupting the test executions. This is because the execution should be independently run, even they can have dependencies with one another. The test tool can facilitates "pass by skipping" ability for test scripts containing failed dependencies that which denied proper test execution.

6.2.4. Analyze

This stage is analyzing test results and data logs obtained from all the previous stages. Primarily, its job is to read all the data and generate test insights. Diagram 6.2.4-1 illustrates the process flow for Analyze stage.

This stage can be independently executed on an already tested environment. This allows user to reuse the raw data results for different analytic processing without needing to re-run all previous stages from scratch.

This stage is resources and time consuming. Hence, the first step is to check user explicit input on whether there is a code coverage running. If it is an explicit no, the test tool can skip this entire stage.

Since the nodes and the test raw data are vital raw materials for analytics, the next step is to check their existence. Without them, there is no point running this stage.

The next step is analyzing the test data. A test tool can decide whether to use the industrial-strength ready facility such as built-in analytic modules or running its own analytic algorithm. Either way, they both must be able to process the nodes data into insights. This analytic facility is what gives the test tool its uniqueness. Hence, many proprietary test tools opted to use its own algorithm.

If the choice is not to use industrial-strength ready facility, this paper illustrates the basic example of building one. The idea is to loop through each identified nodes and cross-check it against the result log. If the node is not found, that means that node is not executed and should file it as negative. If the node is found, depending on the coverage intensity like having multiple coverages at a time, the tool should document it accordingly.

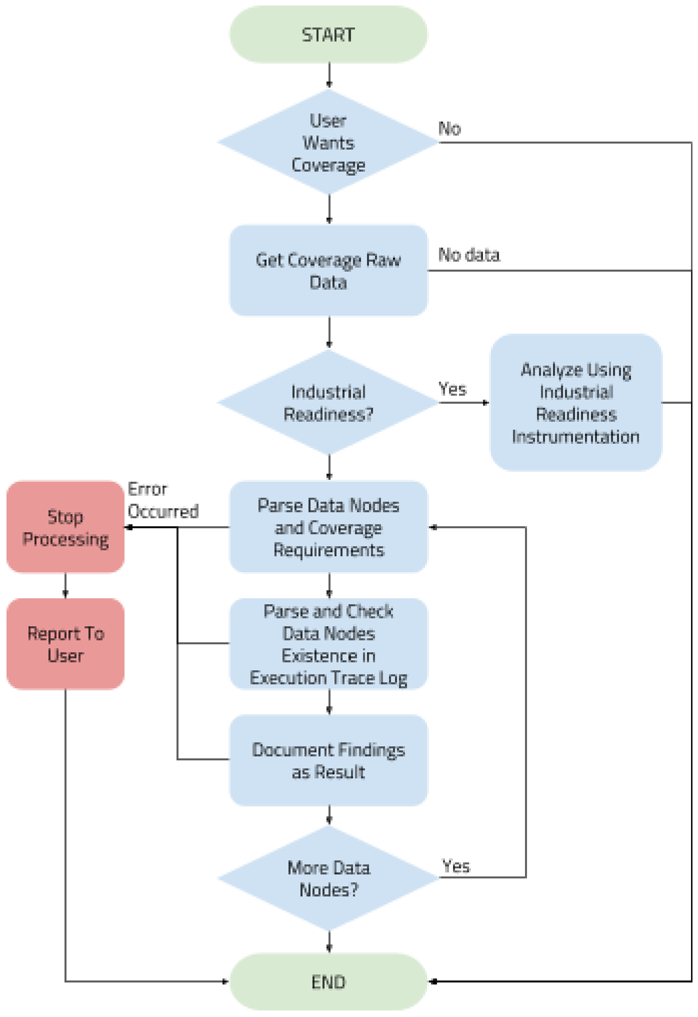

START
User Wants Coverage
No
Get Coverage Raw Data
No data
Industrial Readiness?
Yes
Analyze Using Industrial Readiness Instrumentation
Stop Processing
Error Occurred
Parse Data Nodes and Coverage Requirements
Report To User
Parse and Check Data Nodes Existence in Execution Trace Log
Document Findings as Result
More Data Nodes?
Yes
END

Diagram 6.2.4-1 - Process Flow for Analyze Stage

The coverage report is usually having the minimum of outputs:

1. Total number of nodes
2. Total number nodes executed

3. Total number nodes not executed
4. Difference of executed nodes against total number of nodes
5. Percentage representation of the differences in item 4.

If the test tool did ran a static analysis back in "Parse-and-Inject" stage, this stage also processes its output. Static analysis report is usually consumed in a way that developer will go through the logs and amend the result. Usually, static analysis reports consists of the following items:

1. List of errors identified
2. List of warnings identified
3. List of informational issue identified

The differences between each items is that:

1. an error will cause blocking in execution (e.g. syntax),
2. a warning is issue that can potentially cause blocking in execution or performance degradation (e.g. variable not used),
3. an informational issue with advice (e.g. long variable name)

If any error occurs in this stage, it should be reported directly to the user. This allows user to amend the problem and re-run this stage.

6.2.5. Report

This is the final stage: reporting the analyzed results in an user requested format. User has different requests for reporting the results depending on needs: Some prefers HTML web reports, some prefers on-screen terminal reporting, some prefers files reporting etc.

Although it can be executed independently from other stages, this stage is still go hands-in-hands with analytic stage due to data interpretation dependency.

The stage starts by checking user request for coverage report. The process ends successfully if the request is no. Otherwise, it proceeds to learn the user requested format like HTML, UNIX terminal, JSON, etc. If none is provided, the test tool goes with a default format.

The next step is to get the analyzed data produced from Analyze stage. If the data are missing, this stage ends with error and reports directly to user for attention. Otherwise, the test tool then parse the data, prepare the designated format via template, and then merge the data with the requested report format. The stage ends by presenting the final report to the user.

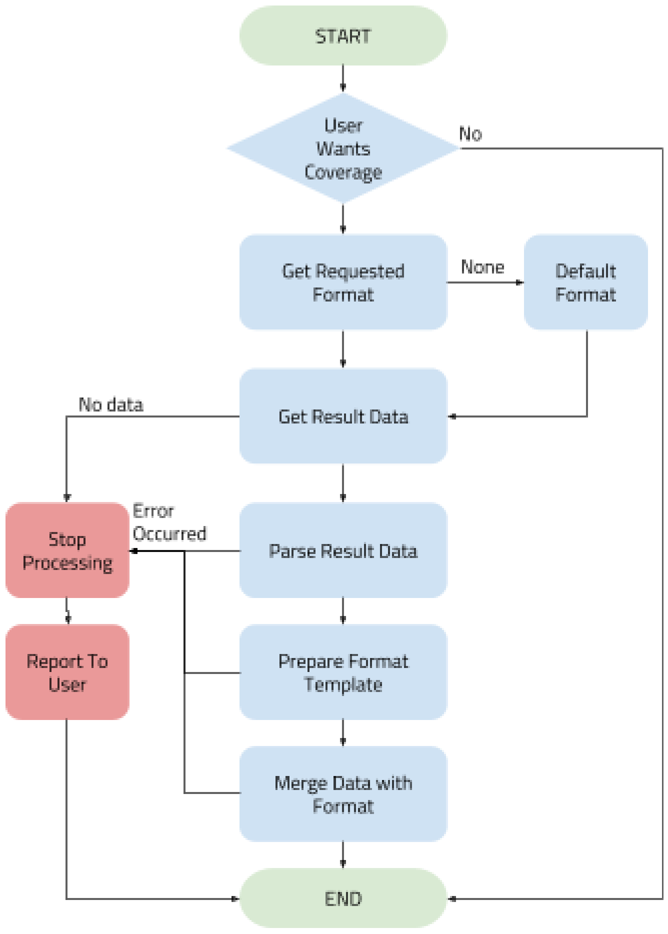


Diagram 6.2.5-1 - Process Flow for Result Stage

If any error occurs, this stage should immediately halt the process and report to user. These errors are typically related to faulty data, parsing issues, or formatting issues that requires user or tool developer's attention.

6.3. Derived Performance Testing Algorithms

Performance testing is a measurement of metrics for a program running. They are usually executed before any release to get a statistical overview of the product's capabilities.. These metrics are available in the definition of quality and are chosen based on the software requirements. As an example, these are the common metrics used in software testing:

1. Execution speed (Time Behavior)
2. Loads limits (Capacity)

3. CPU Loads (Resource Utilization)
4. Memory Loads (Resource Utilization)
5. Storage Loads (Resource Utilization)
6. Power consumption (Resource Utilization)
7. Scalability (Reusability)
8. Backup (Recoverability)

Performance testing should be executed either in the absence of code coverage analytics or as an independent efforts. This is due to the "parse-and-inject" stage in code coverage analytics creates a modification against the original codes. Such modification alters the performance result, yielding an inaccurate performance data.

Performance testing feature can be consolidated inside a single test tool instead of splitting into multiple tools like Valgrind[25]. There are tools like "Go test" for Go programming language that had successfully implemented static analysis, unit testing, code coverage, and performance testing under a single "Go test" feature[44]. Also, "Go test" has proven that performance testing can be scripted like unit test scripts[44].

The performance testing starts by preparing its own test environment. Normally, a performance test runs in repetition and an average value is counted as result. Therefore, the environment must be durable for repetition testing.

The next step is to parse the repetition counts from the user. This is a facility for user to limit the repetition at will. Otherwise, the test tool can default to 1 million repetition count.

Since different metrics has different probe to sense and to capture the data, the next step is to prepare such probe. Example, for power consumption metric, this step is calibrate the power measuring tools. Otherwise, if this step is not needed, it is skippable.

The next few steps are running the repetitive testing. It starts by recording the probe data for "before execution" into a database or data table. Then, it executes the program. After that, it records the probe value for after execution into the database or data table. These steps are repeated based on the determined repetition count.

After the test, the last step is to calculate the performance data and get the average value as a metric. Once done, the performance testing execution ends by reporting the results.

Diagram 6.3-1 illustrates the process flow for executing a performance testing.

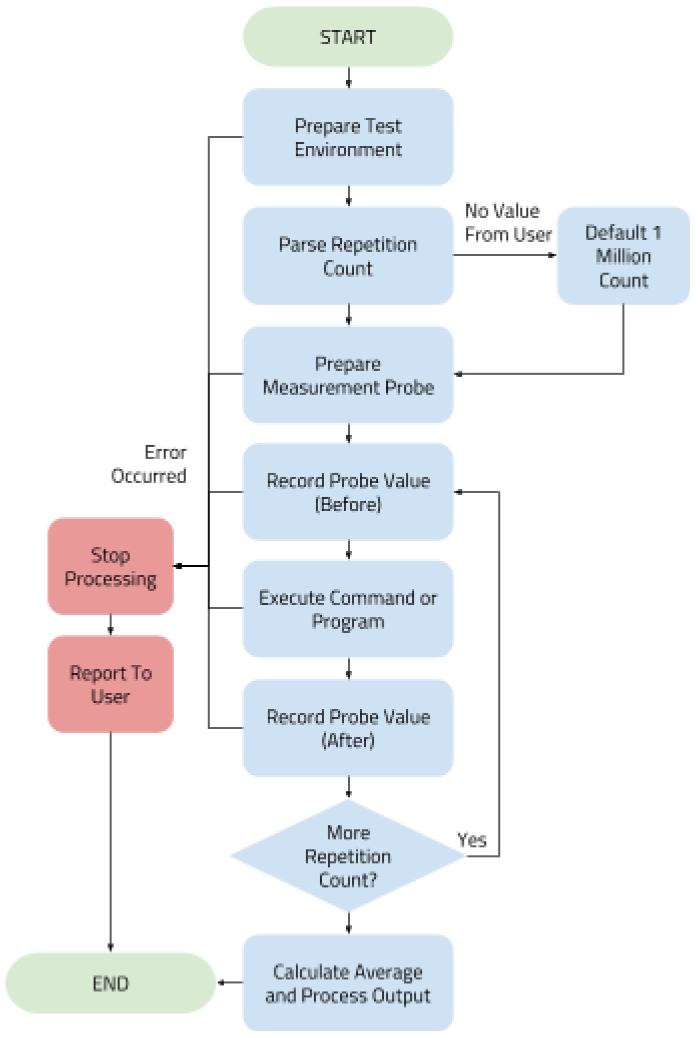


Diagram 6.3-1 - Process Flow for Performance Testing

7. Conclusion

Software test tools are available and robust since the beginning of software development. However, due to their software domain specificities like specific to programming language and operating system, it is hard to deploy across different domains. If we want to develop these tools from scratch, we need a common algorithm extract from the existing tools.

The extraction method in this paper is to learn and compare each tools' test algorithm. This validates one another and make a common pattern creation easier. Also, "quality" should be defined according to the requirements and with reference with ISO 25000 standards.

There are various test coverages and test approaches to choose and implement. The most important consideration in decision is choosing the one that has can have the largest test coverage possible.

With the common algorithms proposed in this paper for both test coverage and performance testing, one can now create the test tools from scratch easily. Also, keep in mind that if a test tool uses source-to-source transformation approach in the test coverage algorithm, the test tool must isolate performance testing algorithm from test coverage. This is due to source codes alteration which affects the performance outcome.

Lastly, new findings like the study of parsing capabilities and power management influencing test operations are found but not researched in detailed. They are good pointers for further research. Also, since this paper does not includes experiment data from the algorithms due to finding a suitable language candidates, a separate research is encouraged for testing them.

8. Acknowledgement

We would like to thank Lim Lee Booi for her continuous constructive criticism of the manuscript despite the hardships she is going through. We wish you a Merry Christmas, speed recovery for your child, and blessing to your family!

9. References

[1] IRA. D. BAXTER, 2002, “Branch Coverage for Arbitrary Languages Made Easy”, *DMS Technical Related Articles*, Semantic Designs, Incorporated, viewed October 2, 2018, available at: http://www.semdesigns.com/Company/Publications/TestCoverage.pdf

[2] ISO25000.com, 2018, “ISO/IEC 25010”, *ISO 25000 Software Product Quality*, ISO/IEC 25000 System and Software Quality Requirements and Evaluation (SQuaRE) series of standards through ISO2500.com, viewed October 2, 2018, available at: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0

[3] ANTÓNIO SILVA, 2017, “How to Write Meaningful Quality Attributes for Software Development”, *Codementor Blog* - www.codementor.io, viewed October 2, 2018, available at https://www.codementor.io/antoniopfesilva/how-to-write-meaningful-quality-attributes-for-software-development-ez8y90wyo

[4] PATRIK BERANDER, LARS-OLA DAMM, JEANETTE ERIKSSON, TONY GORSCHEK, KENNET HENNINGSSON, PER JÖNSSON, SIMON KÅGSTRÖM, DRAZEN MILICIC, FRANS MÅRTENSSON, KARI RÖNKKÖ, PIOTR TOMASZEWSKI, 2005, “Software Quality Attributes and Trade-offs”, *INF5180*, Blekinge Institute of Technology, Universitetet i Oslo, viewed October 2, 2018, available at https://www.uio.no/studier/emner/matnat/ifi/nedlagte-emner/INF5180/v09/undervisningsmateriale/reading-materials/p10/Software_quality_attributes.pdf

[5] MIKHAIL YAKSHIN (GREYCAT), BENJAMIN W., 2011-2017, “Code Coverage Tools for Validating the Scripts”, Stackoverflow.com, viewed October 2, 2018, available at https://stackoverflow.com/questions/7188081/code-coverage-tools-for-validating-the-scripts

[6] CHRIS TOZZI, 2016, “Quality Assurance and Software Testing: A Brief History”, Sauce Labs, viewed October 2, 2018, available at: https://saucelabs.com/blog/quality-assurance-and-software-testing-a-brief-history

[7] EXTREME SOFTWARE TESTING, 2009, “Software Testing History”, eXtreme software testing, viewed October 2, 2018, available at: http://extremesoftwaretesting.com/Info/SoftwareTestingHistory.html

[8] ISO25000.com, 2018, “ISO/IEC 25012”, *ISO 25000 Software Product Quality*, ISO/IEC 25000 System and Software Quality Requirements and Evaluation (SQuaRE) series of standards through ISO2500.com, viewed October 2, 2018, available at: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0

[9] QIAN YANG, J. JENNY LI, DAVID M. WEISS, 2006-2009, “A Survey of Coverage-Based Testing Tools”, *The Computer Journal*, Vol. 52 No. 5, Oxford University Press on behalf of The British Computer Society, DOI: 10.1093/comjnl/bxm021, viewed October 3, 2018, available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.298.3700&rep=rep1&type=pdf

[10] PCI SECURITY STANDARDS COUNCIL, LLC., “Who has to comply with the PCI standards?” *PCI Security Council - FAQ*, Article Number 1436, PCI Security Standards Council LLC., viewed October 3 2018, available at: https://www.pcisecuritystandards.org/faqs

[11] YAN TING WONG TIKY, n.d., “Software Development Life Cycle”, *Master of Science in Information Technology (MSc(IT)) Individual Student Projects*, The Hong Kong University of Science and Technology, viewed October 4 2018, available at: https://www.cse.ust.hk/~rossiter/independent_studies_projects/software_development/software_development_report.pdf

[12] DR. MICHAEL EICHBERG, 2015, “Software Process Models”, *Introduction to Software Engineering*, Software Technology Group - Department of Computer Science - Technische Universität Darmstadt, viewed October 4 2018, available at:

https://stg-tud.github.io/eise/WS11-EiSE-12-Software_Process_Models.pdf

[13] WALT SCACCHI, 2001, “Process Models in Software Engineering”, *Encyclopedia of Software Engineering, 2nd Edition*, Institute for Software Research, University of California, Irvine via John Wiley and Sons, Inc, New York, viewed October 4 2018, available at: https://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf

[14] NAYAN RUPARELIA,2010, “Software development lifecycle models”, *ACM SIGSOFT Software Engineering Notes*, DOI: 10.1145/1764810.1764814, Hewlett Packard Enterprise via ResearchGate, viewed October 4 2018, available at: https://www.researchgate.net/publication/220631422_Software_development_lifecycle_models

[15] PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING, 2018, “Software Development Lifecycle - The Power Of Process”, CSE403: Software Engineering, Department of Computer Science and Engineering, University of Washington, viewed October 4 2018, available at: https://courses.cs.washington.edu/courses/cse403/16sp/lectures/lecture-03-software-lifecycle.pdf

[16] REMOTEONLY.ORG, n.d., “Remote Only”, remoteonly.org, viewed October 4 2018, available at: https://www.remoteonly.org/

[17] LIONEL BRIAND, 2009, “Software Testing Techniques”, Simula Research Laboratory, Oslo Norway, viewed October 8 2018, available at: https://www.uio.no/studier/emner/matnat/ifi/INF1050/v09/undervisningsmateriale/testingteknikk2009.pdf

[18] REDSTONE SOFTWARE, 2008, “Black-box vs. White-box Testing: Choosing the Right Approach to Deliver Quality Applications”, *Computer Security - IT666 Reading List*, Redstone Software Inc. via University of New Hampshire, viewed October 8 2018, available at: http://www.cs.unh.edu/~it666/reading_list/Defense/blackbox_vs_whitebox_testing.pdf

[19] BARBARA G. RYDER, 2006, “Testing2”, *198:431 Software Engineering - Fall 2006*, Virginia Tech - Department of Computer Science, viewed October 9 2018, available at: http://people.cs.vt.edu/ryder/431/f06/lectures/Testing2-11New.pdf

[20] PER RUNESON, 2006, “A Survey of Unit Test Practices”, *IEEE Software*, 23(4), IEEE Computer Society via ResearchGate, viewed October 11 2018, available at: https://www.researchgate.net/publication/27298529_A_Survey_of_Unit_Testing_Practices

[21] ERMIRA DAKA, GORDON FRASER, 2014, ,“A Survey on Unit Testing Practices and Problems”, *2014 IEEE 25th International Symposium on Software Reliability Engineering*, DOI:10.1109/ISSRE.2014.11, viewed October 11, 2018, available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=61C56CC4FE14D115E725F949F60AD2DE?doi=10.1.1.679.3835&rep=rep1&type=pdf

[22] JONATHAN ALDRICH,2011, “Static Analysis”, *15-214-11fa - Principles of Software System Construction*, Carnegie Mellon University - School of Computer Science, viewed October 12, 2018, available at: https://www.cs.cmu.edu/~aldrich/courses/15-214-11fa/slides/static-analysis.pdf

[23] DAVID MONNIAUX, 2009, “Static analysis: from theory to practice”, *CNRS / VERIMAG*, Universit´e Joseph Fourier (Grenoble) and Grenoble-INP, viewed October 12, 2018, available at: https://tel.archives-ouvertes.fr/tel-00397108/file/HDR_Monniaux_slides.pdf

[24] ANDERS MØLLER AND MICHAEL I. SCHWARTZBACH, 2018, “Static Program Analysis”, Aarhus University - Department of Computer Science, viewed October 12, 2018, available at: https://cs.au.dk/~amoeller/spa/spa.pdf

[25] VALGRIND™ DEVELOPERS, 2017, “Valgrind”, valgrind.org, viewed October 15, 2018, available at: http://valgrind.org/

[26] VIDAR HOLEN, 2018, “Shellcheck”, github.com, viewed October 15, 2018, available at: https://github.com/koalaman/shellcheck

[27] BOZHIDAR BATSOV AND RUBOCOP CONTRIBUTORS, 2018, “Rubocop”, github.com, viewed October 15, 2018, available at: https://docs.rubocop.org/en/latest/

[28] JERA DESIGN, n.d., “JTN002 - MinUnit -- a minimal unit testing framework for C”, jera.com, viewed October 15, 2018, available at: http://www.jera.com

[29] STEVEN BAKER, DAVE ASTELS, DAVID CHELIMSKY, ASLAK HELLESØY, CHAD HUMPHRIES, JUSTIN KO, ANDY LINDEMAN, PAT MADDOX, LUKE REDPATH, BRIAN TAKITA, MYRON MARSTON, JON ROWE, SAM PHIPPEN, XAVIER SHAY, AARON KROMER, YUJI NAKAYAMA, BRADLEY SCHAEFER, 2018, “Rspec- Behaviour Driven Development for Ruby. Making TDD Productive and Fun.”, rspec.info, viewed October 16, 2018, available at: http://rspec.info/

[30] TIRDADC, 2014, “How to check rspec code coverage”, Stack Exchange Inc., viewed October 16, 2018, available at: https://stackoverflow.com/questions/22893907/how-to-check-rspec-code-coverage

[31] CHRISTOPH OLSZOWKA, 2017, “SimpleCov”, github.com, viewed October 16, 2017, available at: https://github.com/colszowka/simplecov

[32] JAMES BRITT, NEUROGAMI, 2018, “Coverage”, rubydoc.org - version 2.4.0, viewed October 16, 2017, available at: http://ruby-doc.org/stdlib-2.4.0/libdoc/coverage/rdoc/Coverage.html

[33] YUSUKE ENDOH, YUKIHIRO MATSUMOTO, 2008, “ruby/ext/coverage/coverage.c”, github.com, viewed Octboer 16, 2017, available at: https://github.com/ruby/ruby/blob/trunk/ext/coverage/coverage.c

[34] PYTHON SOFTWARE FOUNDATION, 2018, “unittest - Unit Testing Framework”, Python Software Foundation, viewed Octboer 16, 2017, available at: https://docs.python.org/3/library/unittest.html

[35] NED BATCHELDER, 2018, “Coverage.py”, coverage.readthedocs.io - version 4.5.X, viewed October 16, 2017, available at: https://coverage.readthedocs.io/en/v4.5.x/index.html

[36] NICHOLAS CARLSON, 2013, “Ex-Yahoos Confess: Marissa Mayer Is Right To Ban Working From Home”, *Business Insider*, Insider Inc, viewed November 28, 2018, available at: https://www.businessinsider.com/ex-yahoos-confess-marissa-mayer-is-right-to-ban-working-from-home-2013-2/?IR=T

[37] LARRY ALTON, 2017, “Are Remote Workers More Productive Than In-Office Workers?”, Forbes, Forbes Media LLC, viewed November 28, 2018, available at: https://www.forbes.com/sites/larryalton/2017/03/07/are-remote-workers-more-productive-than-in-office-workers/#11cd379331f6

[38] FORBES TECHNOLOGY COUNCIL, 2017, “13 Pros And Cons Of Having A Distributed Workforce”, Forbes, Forbes Media LLC, viewed November 28, 2018, available at: https://www.forbes.com/sites/forbestechcouncil/2017/08/03/13-pros-and-cons-of-having-a-distributed-workforce/#33bb4b5913d9

[39] GERRY CLAPS, “The Difference Between Remote and Distributed Teams in Startups”, *Blossom IO Blog*, Blossom IO Inc., viewed November 28, 2018, available at https://www.blossom.co/blog/remote-versus-distributed-teams

[40] LAUREEN MILES BRUNELLI, “Use This Work-at-Home Company Directory to Find Your Dream Job”, The Balance Careers, viewed November 28, 2018, available at: https://www.thebalancecareers.com/work-at-home-jobs-company-directory-3542836

[41] MARCIA RAMOS, 2016, “GitLab Workflow: An Overview”, GitLab Inc., viewed December 5, 2018, available at: https://about.gitlab.com/2016/10/25/gitlab-workflow-an-overview/

[42] DIGITALOCEAN, 2018, “DigitalOcean CI-CD”, DigitalOcean Inc., viewed December 5, 2018, available at: https://www.digitalocean.com/community/tags/ci-cd?type=tutorials

[43] CIRCLECI, 2018, “CircleCI - How It Works”, Circle CI Internet Services Inc., viewed December 5, 2018, available at: https://circleci.com/product/#how-it-works

[44] DAVE CHENEY, 2013, “How to Write Benchmarks in Go”, dave.cheney.net, viewed December 16, 2018, available at: https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go

[45] GO PROGRAMMING LANGUAGE, 2018, “Source file src/testing/testing.go”, golang.org, viewed December 24, 2018, available at: https://golang.org/src/testing/testing.go