ISSN 1816-6075 (Print), 1818-0523 (Online)
Journal of System and Management Sciences
Vol. 12 (2022) No. 4, pp. 85-100
DOI:10.33168/JSMS.2022.0406

Implementation Approach of Unit and Integration Testing Method Based on Recent Advancements in Functional Software Testing

Zheng Yang Joel Tan, Mohammed Mahedi Hasa, Man Yi Wong, R Kanesaraj Ramasamy

Faculty Computing and Informatics, Multimedia University

kanes87@gmail.com

Abstract. Finding bugs and flaws, detecting invalid or inaccurate functionality, and analyzing and certifying the entire softwareproduct all require software testing. We looked at unit testing and integration testing in this project since they are two fundamental stages of software testing and are significantly associated. For both unit and integration testing, a sufficient number of testing methodologies and approaches have been assessed and contrasted, with each implementation system, algorithm, and technique being thoroughly scrutinized. Some of them are effective in finding as many hidden defects as possible while also reducing testing complexity, time, and expense. In this context, we chose sOrTES, a stochastic scheduling support tool that would be utilized for manual integration test cases. The chosen strategy is the most appropriate since empirical evidence reveals that it can prevent around 40% of testing failures while also increasing requirement coverage by 9.6%.

Keywords: integration testing, unit testing, software testing, class integration test order, functional testing, test optimization, stochastic test scheduling, dependency.

1. Introduction

Software testing is one of the most challenging steps in the whole software development process. Proper testing needs a methodical approach for evaluating and verifying a software product to detect bugs, faults, and gaps which also leads to finding the correctness of behavior and performance of the software. Software testing can be classified into two main aspects, functional testing, and non-functional testing. The scope of this paper is to cover only the first and second stages of functional testing known as unit testing and integration testing. We found many relevant methods for the unit and integration testing. Afterward, we evaluated and compared those methods to select the most suitable one for using in new software.

Unit testing is considered the first stage of software testing levels. In this stage, each individual unit or component of the software is tested to verify if the intended functionality works properly. In test-driven development (TDD), unit tests are created even before the programmers write the actual code for the software. The written code is considered complete when the unit test is passed. We have reviewed a few unit testing methods, algorithms, and methodologies. For example, Zhang et al. (2019) focused on preconditions of individual functions and proposed the rule-directed symbolic execution approach using a tool, CTS-IC (Code Testing System with Implicit Constraint), which was able to obtain good coverage of the function even when preconditions were missing. Menendez et al. (2022) introduced a unit test generation tool named OutGen, which implements an automatic output sampling to provide unit test sets with diversified output. On the other hand, Nassif et al. (2021) proposed a technique called DScribe, which is tool-supported and complies with the co-generation of unit tests and documentation. Alternatively, Cerioli et al. (2021) presented a tool TestWizard, for automatically assessing the unit test method if it is coherent to its specification. An interesting use case for this unit testing method is that it can be used to elevate the performance testing procedures. To minimize the practical obstacles while applying the unit testing for performance testing, Bulej et al. (2017) introduced an algorithm named Stochastic Performance Logic (SPL). Overall, all the analyzed methods and tools proposed by various researchers for unit testing are mostly used targeting automated unit testing.

After unit testing, integration testing is the next step in the software testing process. Software testers use integration tests to assess the performance of individual components as a whole and uncover any issues with the interface between modules and functionalities. Various research papers have been reviewed to find and evaluate integration testing approaches in software testing. Class Integration Test Order (CITO) which decides the order, inwhich the detection of inter-class faults occurs, has been discussed by Zhang et al. (2021) and Jiang et al. (2021). Theirmain objective was to reduce the overall stubbing complexity and the costs in problem space.

Zhang et al. (2021) used identical class dependence (ICD) and symmetric classes (SC) to get the same types of classes and introduced a cycle-breaking algorithm for the integration testing. On the other hand, Jiang et al. (2021) proposed ConCITO, a test order strategy to control coupling that improves existing stubbing cost and complexity. Zhang et al. (2019) introduced the Particle Swarm Optimization (PSO) algorithm in another paper toachieve precision and speedy convergence for creating test orders. Czibula et al. (2018) proposed Reinforcement Learning (RL) to minimize the stubbing effort and reducetime. Other than the CITO approach, some researchers proposed machine learning methods. For instance, Lima et al. (2019) mentioned that in the Continuous Integration (CL) environment, the test case prioritization (TCP) technique finds the appropriate order of test cases whichimproves the overall fault detection. Furthermore, Yang et al. (2021) proposed the Reward with Additional Rewardmethod where Test Occurrence Frequency (TOF) finds the failure effect of a test case. Another unique method was the time-window-based reward function that considers test cases with failure quantities and failure distribution. One of the most remarkable methods proposed by Tahvili et al. (2019), where the researchers introduced a supportive tool for stochastic scheduling ofmanual integration test cases known as sOrTES. sOrTES,the Python-based tool, schedules manual integration test cases in natural language text.

After evaluating the approach of sOrTES, we finally chose sOrTES as the recommended method for this project. During the early phases of testing, sOrTES can help testers better understand the dependencies for the testing requirements. Tahvili et al. (2019) mentioned that about 40% of testing failures could be avoided using the proposed execution method of sOrTES. Moreover, the adopted method should reduce human judgment when deciding which test case to run first. Finally, we can expect a better outcome when applying sOrTES in a newsoftware product.

The objectives of this project are formulated as follows:

1. To study the existing unit and integration testing techniques.
2. To evaluate various methods of the unit and integration testing.
3. To understand how the method or algorithm works,designed and implemented for testing software.
4. To select the most suitable method for the unit or the integration testing by comparing the existing methods.

Selecting the suitable method to conduct the unit or the integration testing is crucial in software testing. Thisproject could help to find the suitable method for the unitor the integration test by comparing many existing nobletest approaches. During the selection of the suitable method, we will check the method in detail to see how the method works, how the algorithm is designed and finally, how we can implement that algorithm for testing.The selected method will be used to test an existing software product where the ultimate target is to reduce the test complexity as well as to detect more functional faults and bugs. Furthermore, choosing a suitable method could reduce the testing time and the cost of either testing.

2. Literature Review

This section summarizes the papers, which have proposed their own methods, into Table I. The advantages, as well as the limitations or challenges of each method, are summarized to provide an overview of the methods. This table is mainly used to show the advantages and limitations of each method clearly. In addition, from Table I, it can be found that the literature review performed focuses on integration testing. As integration testing is normally performed after unit testing, the studies on unit testing are mainly aimed to understand the steps prior to integration testing better.

In Table I, there are several algorithms, methodologies, frameworks, and tools introduced to optimize the testing process. It is clear that each method has its own advantages and limitations, thus the selection of the test method should depend on the system under test (SUT). For example, some methods are not suitable for large systems. Furthermore, even though all methods have been proven to be effective in certain situations, they do not experiment with large amounts of datasets. Some methods are not implementation-friendly as they are too complicated. Therefore, when determining the method that will be used to test the SUT, the limitations and challenges of the method should be taken into consideration.

Table 1: Summary table of the paper reviewed

Method	Testing Type (Integration/Unit)	Advantages	Limitations/Challenges
Combining similar-class withdependency (Zhang et al., 2021)	Integration	This method obtained lower stubbing costs compared to traditional methods. Also, the cycle numbers were minimizedfor similar classes, reducing theproblem space without hampering the performance.	The researchers have not testedtheir method for various programming languages, so the generalizability of this method still needs to be explored.
ConCITO (Control CouplingClass Integration Test Order Generation (Jiang et al., 2021)	Integration	Generated CITO with less overall stub complexity, loweststub cost in shortest execution time.	Does not check the source codethat may affect the authenticityof path conditions.
Multi-level feedback approach,MLFCITO (Zhang et al., 2017)	Integration	Provided shortest execution time, less time consuming, generate test order with less overall stubbing complexity forlarge-scale systems.	Not suitable for use in small systems as execution time is thesame as other algorithms.

Multi-granular flow network(MGFN) model (Wang et al., 2018)	Integration	Able to test higher risk classes in early integration steps and minimize overall complexity ofestablished test stubs.	Only 3 software were selected for testing with 2 other algorithms, resulting in limitedtest results.
Particle swarm optimizationalgorithm (Zhang et al., 2019)	Integration	Great precision and fast convergence speed to generateCITO.	Experiment only available forjava program.
metamorphic testing (Zhang et al., 2021)	Integration	It is the first approach to validate the existing CITO generation systems.	Experiments were only conducted on CITO generationsystems that are not public.
Reinforcement learning approach (Czibula et al., 2018)	Integration	Reduced the test time and stubbing effort.	Experiment does not evaluatelarge software systems.
Historical failure information-based rewards and additional reward method (Yang et al., 2021)	Integration	The sparse reward in TCP of RL-based CI testing is improved and the TCP effect obtained by the reward with additional reward is better than those without additional reward.	Information of the requirementand code is not used to design the rewards.
Historical Reward Strategies(Yang et al., 2020)	Integration	Improved the ability of fault detection of the test order.	The CI cycle and time are limited to running huge amounts of historical data. The size of test cases and execution could affect the effectiveness oftest prioritization history.
Reinforcement learning approach with test suite-baseddynamic sliding window and individual test case-based dynamic sliding window (Yang et al., 2021)	Integration	Effective in improving the testcase prioritization effect that better adopts the CI environment.	The award function might notbe effective in the situation oflow failure rate limitation in industrial datasets. Similarity and correlation among the testcases were not included.
NLP & LSTM (Deep LearningAlgorithms) and search-based approaches (genetic algorithmsand simulated annealing) (Medhat et al., 2020)	Integration	Enhanced the efficiency of Continuous Testing in IoT systems for prioritization and selection purposes.	The accuracy of test prioritization has to be further improved. Only several testingareas are included, which are integration testing and regression testing.

Two test case selection approaches, class-level FEST, and method-level MEST, basedon framework Sapient (Li et al., 2020)	Integration	Reduced test size and test cost,with better fault detection efficiency and cost-effectiveness.	In rare cases, dynamic bindingin MEST may lose some dependencies when a subclassinstance is a parameter of superclass.
Using coverage metrics to classify the tests which then a regression test suite is created to consist only of effective and unique partially redundant tests (Marijan et al., 2019)	Integration	Improves the performance of CIand significantly improves fault-detection.	Regression testing approach isstill not efficient enough to reduce ineffective test redundancy.
Hybrid multi-criteria selectionmethod using Analytic Hierarchy Process (AHP) and Technique for Order Preference by Similarityto Ideal Solution (TOPSIS) (Abdulwareth et al., 2021)	Integration	Accuracy is high as all the toolsare ranked accurately. It is beginner-friendly and has reduced the selection cost.	Experts indicated that the taxonomy is complex and couldbe difficult to use.
Test focus selection for integration testing [19]	Integration	With a small number of developed test cases, this method can detect 80% of integration errors in tested applications.	This method has two limitations, other than the Javasystem this method may not work, and even in Java programs, it may not be applicable for all domains.
Stochastic scheduling and Natural Language Processing(NLP) (Tahvili et al., 2019)	Integration	Achieved a more efficient testing process and better quality of software product. Lesser human work and judgment and more trustable results.	The quality of the result is dependent on human-written SRS and test specifications. Criteria of test cases must be measured before the first execution. Specifications that are written by humans make itharder and more complex for the system.
An integration testing framework and evaluationmetrics for vulnerability detection (Li 2018)	Integration	Provided effective black box vulnerable mining detection.	It tends to look for vulnerable patterns rather than installed libraries. Only a few vulnerable mining methods are

			included.
Bayesian Optimization Algorithm (BOA) with three different structures is GROOVEtoolset (Rafe et al., 2021)	Integration	Better in coverage and speed compared to existing approaches.	To achieve a set of test objectives, explosions of statespace and test cases might happen while exploring paths.
A method for prioritzing integration testing in softwareproduct lines based on featuremodel (Akbari et al., 2017)	Integration	Prioritizing integration testingbased on feature model	The researchers have not yet developed an algorithm to find the test case reduction rate (TCRR) and test efficiency rate (TER), which helps the domain engineer during the test process.
Set of integration test coveragemetrics (Mukherjee et al., 2019)	Integration	This approach considers multiple characteristics of object-oriented programs suchas objects and methods. Compared to other approaches,this approach also uses class relations based on the design ofobjects while ignoring the traditional connected paths of various methods.	The researchers used only the first-order mutants for the Java mutation testing tool, which may affect the result of fault detection.
Integration Testing Rules (ITR)model based on four main models: the reconciled solution model, the data sources models,the transformations model, and the test models (Blanco et al., 2018)	Integration	Effectively find the deficienciesto improve the final results of the ER application.	Less variety of case studies tovalidate the approach.
Time-Constrained Fragment and Compare (TCFC) algorithm (Brkić et al., 2018)	Integration	It is ideal for generating the overview of differences amongtables in the database within a given time frame.	It is possible to obtain a less accurate comparison of tables asa global overview. Large table comparison could block up all the time available and not all differences can be detected.

Rule-directed symbolic execution using CTS-IC (Zhang et al., 2019)	Unit	Implicit constraints assisted CTS-IC to obtain high coverageof the functions with missing preconditions	Manual checking is required toensure the correctness of implicit constraints.
Automatic output sampling foroutput diversity in unit test generation (Menendez et al., 2022)	Unit	Outperformed other generation approaches in output uniqueness, mutation score, andfault detection.	The Z3 solver restricts the program size and thus lowers the tool's scalability. The definition of diversity needs tobe revised to reduce redundancy.
Co-generation of unit tests anddocumentation with template invocations (Nassif et al., 2021)	Unit	Tests generated are found to bemore readable than those by humans or other state-of-the-artautomated techniques.	The results are highly dependent on the templates.
Unit test's quality assessmentusing anti-oracles (Cerioli et al., 2021)	Unit	More accurate than the manualcode review done by multiple students. The accuracy is evenslightly higher than the three senior experts.	This approach requires the testspecification to have a certain level of formality.
Mutation analysis (Trautsch et al., 2020)	Integration, Unit	Able to analyze the capabilitiesof the unit and integration testing.	The evaluation was only done on the java program, others programming language projectswere not tested.
Determine the relationship between the software tester, personality characteristics, and software testing levels using MBTI (Kamangar et al., 2021)	Integration, Unit	Increased the effectiveness andreliability of crowd-based outsourced software testing.	Limited dataset. More testing done by the tester would result in more appropriate results. The results would also be affected by the tester’s experience on using a particular testing level.

3. Method and Results

Based on the literature review, we selected sOrTES (Tahvili et al., 2019) as the recommended method for our study. The reasons why we select sOrTES as our recommendation method are because sOrTES is able to assist testers to gain a better understanding of the dependencies betweenthe requirements during the early stages of testing. Additionally, this method helps to reduce humanjudgment when selecting which test case should be executed first and the reliable result produced by sOrTES that can help organizations achieve greater efficiency andhigher quality of software products during the testing process. With high-quality software testing,organizations save on debugging costs during the testingphase and are able to expedite the release of the final product.

3.1. Explanation of method works and algorithm design

sOrTES is a Python-based system for assisting automated decision-making, consisting of two distinct phases, the extraction phase, and the scheduling phase. The requirement coverage and dependencies of each test case are collected in the extraction phase, and the execution time is abstracted from the external tool ESPRET before entering the scheduling phase to set the ranking of the execution prioritized test cases.

During the extraction phase, test cases are collected from the requirement specification and test specificationfiles as input in excel file format. Then, the data will be generated in a table consisting of dependencies between test cases, requirement coverage, and output columns. The dependency column is collected from the excel file that contains all test cases. The dependencies can be determined when there are two test cases that are related,for example, the input of test case A is related to the output of test case B, which means test case A needs to wait for the complete execution of test case B before it can be executed. The requirement coverage column can be collected in test cases by counting the total number of requirements, while the output column can be generated based on the number of test cases that can be tested aftereach test case is inserted.

During the scheduling phase, the following algorithms are used to generate the scheduling ranking that defines the best execution order. The R in Equation

(1) represents the result of each test case TC_i, which mayresult in 1 as fail or 0 as pass. This result can be tested from the sample function P in Equation (2) by determining the dependencies between the TCs. For example in Equation (2), let's say the TC_j is dependent onTC_i, the results of TC_j, also known as R_j will always be 1if TC_i was never tested before even though the TC_j is passed. Once all the dependencies and requirement coverage are found in all TCs, the F in Equation (3) can perform to schedule ranking of test cases. The F is a feasible set for setting all possible ways of testing withinTC_i according to the precedent P_i. The P_i is the precedentsfor each test case TC_i, to determine which TC_i is directlydependent on, for example, if TC_2 is dependent on TC_1, then the P_2 will have TC_1, $P_2 = \{TC_1\}$, where the P_1 is anempty set.

$$R_i = \begin{cases} 1, & \text{if test case } \mathrm{TC}_i \text{ fails} \\ 0, & \text{if test case } \mathrm{TC}_i \text{ passes.} \end{cases} \tag{1}$$

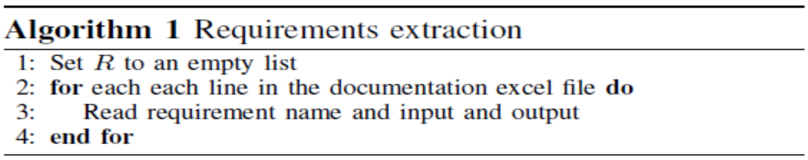
$$\mathbb{P}(\mathrm{TC}_j \text{ passes} | \mathrm{TC}_i \text{ was not tested}) = 0. \tag{2}$$

$$\begin{aligned} \mathcal{F} = \big\{ (\mathrm{TC}_{\cdot,1}, \mathrm{TC}_{\cdot,2}, \cdots, \mathrm{TC}_{\cdot,n}) : P_{\cdot,1} = \emptyset, \\ \forall\, i = 2, \cdots, n,\ P_{\cdot,i} \subseteq \{\mathrm{TC}_{\cdot,1}, \cdots, \mathrm{TC}_{\cdot,i-1}\} \big\}. \end{aligned} \tag{3}$$

3.2. Implementation of the Algorithms

From the previous section, we discussed how thealgorithms have been designed in the previous section inorder to achieve the automation of test cases dependencies detection and test case scheduling to achieve the proposed decision-making system. This information helps us to determine the dependencies detection workflow at Integration Testing. The implementation details are a continuous work which described the necessary packages, libraries, and pseudocodes, thus, we will first review thepython implementation as presented by Tahvili et al., 2018 before further extending the work of the sOrTES.

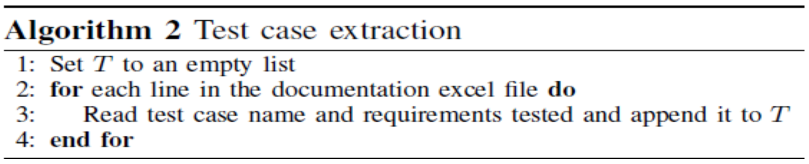
The authors admitted that dependency detection should be started in the early stage. Natural Language Processing techniques are utilized as the two required inputs, namely requirement specification and test specification, are written in natural language. The important information has been extracted from the documents and mapped to each other to examine the interdependencies among the requirements and test cases. The authors presented three steps to complete the dependencies linkage, including test case extractor (Fig. 11), requirements extractor (Fig. 10), and test case and requirement combiner (Fig. 13, Fig. 12). Requirements extraction algorithm defines how the requirements can be extracted from the documents (.xlsx files) by recording the requirement name, and input and output signals; Test case extraction algorithm extracts the important information from the specification by using xlrd2 library package to track the relevant data. In the test case and requirement combiner steps, two algorithms were presented where both of the results from the requirements extractor and test case extractor are combined to create a dependency graph. With the dependencies detected from the specifications, the authors made use of vis.js and javascript libraries to visualize the graph.



Algorithm 1 Requirements extraction

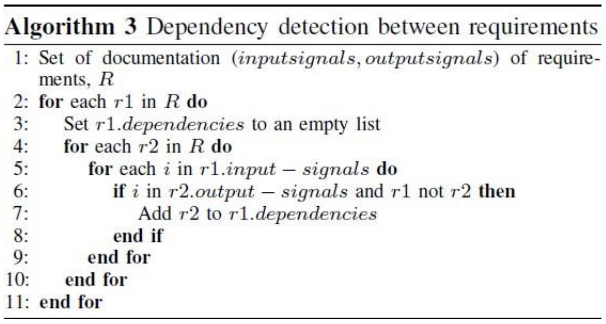
```
1: Set $R$ to an empty list
2: for each each line in the documentation excel file do
3:     Read requirement name and input and output
4: end for
```

Fig. 1: Snapshot of requirements extraction algorithm presented by Tahvili et al., (2018).



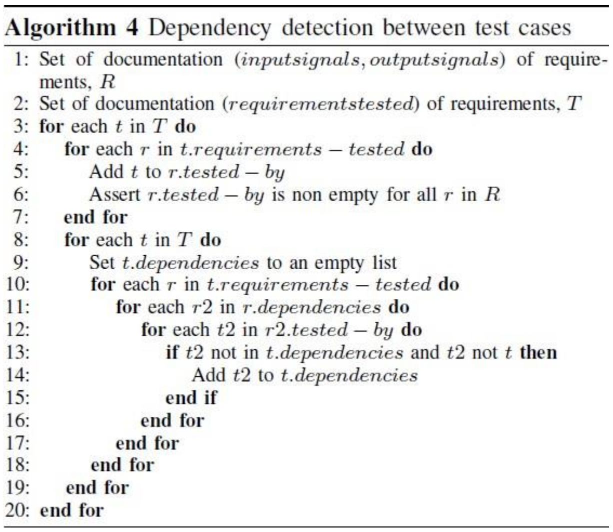
```
Algorithm 2 Test case extraction
Set $T$ to an empty list
for each line in the documentation excel file do
    Read test case name and requirements tested and append it to $T$
end for
```

Fig. 2. Snapshot of test case extraction algorithm presented by Tahvili et al., (2018).



```
Algorithm 3 Dependency detection between requirements
Set of documentation $(inputsignals, outputsignals)$ of requirements, $R$
for each $r1$ in $R$ do
    Set $r1.dependencies$ to an empty list
    for each $r2$ in $R$ do
        for each $i$ in $r1.input - signals$ do
            if $i$ in $r2.output - signals$ and $r1$ not $r2$ then
                Add $r2$ to $r1.dependencies$
            end if
        end for
    end for
end for
```

Fig. 3: Snapshot of dependency detection betweentest cases algorithm presented by Tahvili et al., (2018).



```
Algorithm 4 Dependency detection between test cases
Set of documentation $(inputsignals, outputsignals)$ of requirements, $R$
Set of documentation $(requirementstested)$ of requirements, $T$
for each $t$ in $T$ do
    for each $r$ in $t.requirements - tested$ do
        Add $t$ to $r.tested - by$
        Assert $r.tested - by$ is non empty for all $r$ in $R$
    end for
    for each $t$ in $T$ do
        Set $t.dependencies$ to an empty list
        for each $r$ in $t.requirements - tested$ do
            for each $r2$ in $r.dependencies$ do
                for each $t2$ in $r2.tested - by$ do
                    if $t2$ not in $t.dependencies$ and $t2$ not $t$ then
                        Add $t2$ to $t.dependencies$
                    end if
                end for
            end for
        end for
    end for
end for
```

Figure 4. Snapshot of dependency detection between testcases algorithm presented by Tahvili et al., (2018).

After the dependency analysis, the authors carry on the manual test case scheduling process by utilizing ESPRET (Tahvili et al., 2018), a tool for calculating the test case execution time to better improve the accuracy of ranking the test cases. The approach measures the estimation of execution time according to the actual execution time from historical execution data through NLP techniques, log analysis, and finally, regression models were applied. A summary of the steps taken by ESPRET is demonstrated in Fig. 14. When there is no prior test execution data, the system will parse the new test step by determining if the step is matched with the previous execution. If there is no previous data, a baseline value will be assigned to the test case. After the estimation of test case execution has been done, the polynomial regression models are applied to obtain the actual execution time. Total execution time will then be returned at the end of the process.

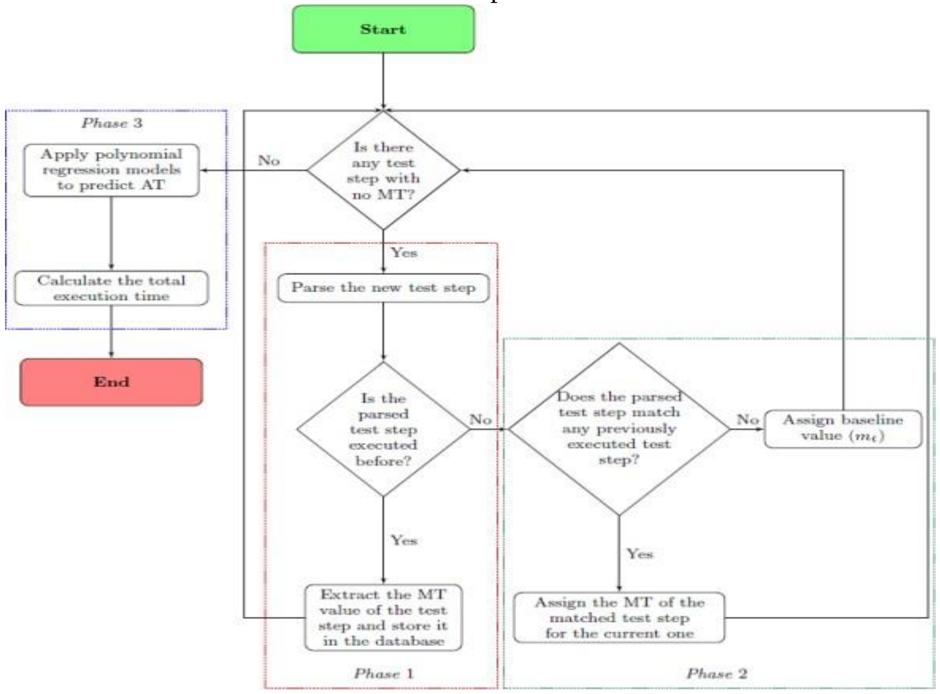

Start
Is there any test step with no MT?
No
Yes
Phase 3
Apply polynomial regression models to predict AT
Calculate the total execution time
End
Parse the new test step
Is the parsed test step executed before?
No
Yes
Extract the MT value of the test step and store it in the database
Phase 1
Does the parsed test step match any previously executed test step?
No
Yes
Assign baseline value (m_t)
Assign the MT of the matched test step for the current one
Phase 2

Fig. 5: ESPRET tool workflow diagram (adaptedfrom Tahvili et al., (2018)).

From all of the information gathered and provided through the steps above-mentioned, the automatic test scheduling can be optimized. The authors have done an industrial case study to verify the usability of the proposed tool for dynamic test case scheduling. Thedependencies among the test cases and requirements have been identified. Then, the requirements coverage is computed and test cases can be ranked accordingly. Redundant test execution can be avoided through the dynamic scheduling of the test case based on the dependencies change and cost is effectively reduced.

4. Conclusion

Software bugs and defects could be deadly as they might cost billions of dollars and countless precioustimes. Therefore in this work, over 30 unit test andintegration test papers with various methods or approaches were studied and analyzed to determine the best functional software testing method. More than half of the papers were researching integration testingmethods where CITO-related approaches, machine learning-related methods, and selection of test element methods were the main focuses. On the other hand, unit test generation, assessment, and performance testing were the main focus for the unit testing papers that were being reviewed. A summary of all the papers was illustrated in Table I with respective advantages andlimitations included. With all the papers being referencedand analyzed, sOrTES has chosen to be ourrecommendation method in view of its ability to enable testers to better understand the requirementsdependencies, reduce human judgment during test case selection, as well as all the exceptional outcomes of producing reliable results. Detailed explanations and ways of implementation were as well discussed. This work has provided a comprehensive view of recent workson the unit and integration testing approaches proposed.

Authors' Contributions

We reviewed unit testing and integration testingin this project as they are highly correlated and are two fundamental levels of software testing. Then, we chose an approach, sOrTES, a supportive tool for stochastic scheduling that will be used for manual integration test cases.

Acknowledgments

We would like to take this opportunity to express our most sincere gratitude to our subject lecturer, Ts. Dr. R. Kanesaraj A/L Ramasamy, who is a lecturer in the Faculty of Computing and Informatics, Multimedia University Cyberjaya, Malaysia. He has spent much time lecturing and guiding us throughout the completion of this work. The work would not have been completed without his clear explanation and constructive feedback. Especially during this Covid-19 pandemic, Dr. R. Kanesaraj tried his best in providing us with both online and physical classes as well as consultation. He is kind enough to sacrifice his time to clear out our doubts and confusion regarding the work. No words can express our thankfulness to him.

References

Abdulwareth, A. J. & Al-Shargabi, A. A. (2021). Towarda multi-criteria framework for selecting software testing tools. *IEEE Access*, 9, 158872- 158891, DOI:10.1109/ACCESS.2021.3128071.

Akbari, Z., Khoshnevis, S., & Mohsenzadeh, M. (2017). A method for prioritizing integration testing in software product linesbased on feature model. *International Journal of Software Engineering and Knowledge Engineering*, 27(4), 575–600.

Banitaan, S., Nygard, K., & Magel, K. (2017). Test focus selection for integration testing. *International Journal of Software Engineering and Knowledge Engineering*, 27(8), 1145–1166.

Blanco, R., Enríquez, J. G., Domínguez- Mayo, F. J., Escalona, M. J., & Tuya, J. (2018). Early Integration testing for entity reconciliation in the context of heterogeneous data sources. *IEEE Transactions on Reliability*, 67(2), 538-556. DOI:10.1109/TR.2018.2809866.

Brkić, L. & Mekterović, I. (2018). A time- constrained algorithm for integration testing ina data warehouse environment. *Information Technology and Control,* 47(1), 5-25.

Bulej, L. (2017). Unit testing performance with stochastic performance logic. *Automated Software Engineering*, 24(1), 139-187.

Cerioli, M., Lagorio, G., Leotta, M., & Ricca, F. (2021). Fight silent horror unit test methods by consulting a TestWizard. *Journal of Software:Evolution and Process*, e2396.

Czibula, G., Czibula, I.G., & Marian, Z. (2018). An effective approach for determining the class integration test order using reinforcement learning. *Applied Soft Computing,* 65, 517-530.

Jiang, S., Zhang, M., Zhang, Y., Wang, R., Yu, Q., & Keung, J. W. (2021). An integration test order strategy to consider control coupling. *IEEE Transactions on Software Engineering*, 47(7),1350-1367. DOI:10.1109/TSE.2019.2921965.

Kamangar, Z. U., Siddiqui, I. F., Arain, Q. A., Kamangar, U. A., & Qureshi, N. M. (2021). Personality characteristic-based enhanced softwaretesting levels for crowd outsourcing environment. *KSII Transactions on Internet and Information Systems (TIIS)*, 15(8), 2974-2992, 2021.

Lima, J. A. P. & Vergilio, S. R. (2020). Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121, 106268.

Li, J. (2018). An integration testing framework and evaluation metric for vulnerability mining methods. *China Communications,* 15(2), 190-208. DOI:10.1109/CC.2018.8300281.

Li, Y., Wang, J., Yang, Y., & Wang, Q. (2020). An extensive study of class-level and method-leveltest case selection for continuous integration. *Journal of Systems and Software*, 167, 110614, 2020.

Marijan, D., Gotlieb, A.. & Liaaen, M. (2019). A learningalgorithm for optimizing continuous integration development and testing practice," Software: Practice and Experience, 49(2), 192-213.

Menendez, H. D., Boreale, M., Gorla, D., & Clark, D. (2022). Output sampling for output diversity in automatic unit test generation. *IEEE Transactions on Software Engineering*, 48(1),295-308. DOI:10.1109/TSE.2020.2987377.

Medhat, N., Moussa, S. M. Badr, N. L., & Tolba, M. F. (2020). A framework for continuous regression and integration testing in iot systems based on deeplearning and search-based techniques. *IEEE Access*, 8, 215716-215726. DOI:10.1109/ACCESS.2020.3039931.

Mukherjee, D. & Mall, R. (2019). An integration test coverage metric for Java programs. *International Journal of System Assurance Engineering and Management*, 10(4) 576–601.

Nassif, M., Hernandez, A., Sridharan, A., & Robillard, M. P. (2021). Generating unit tests for documentation. *IEEE Transactions on Software Engineering*. DOI:10.1109/TSE.2021.3087087.

Rafe, V., Mohammady, S., & Cuevas, E. (2021). Using Bayesian optimization algorithm for model-based integration testing. *Soft Computing*, 1-23.

Tahvili, S. (2018). Functional dependency detection for integration test cases. *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 207-214. DOI:10.1109/QRS-C.2018.00047.

Tahvili, S., Pimentel, R., Afzal, W., Ahlberg, M., Fornander, E., & Bohlin, M. sOrTES: A supportive Tool for stochastic scheduling of manual integration test cases. *IEEE Access*,7, 12928-12946. DOI:10.1109/ACCESS.2019.2893209.

Tahvili, S., Afzal, W., Saadatmand, M., Bohlin, M., & Ameerjan, S. H. (2018). ESPRET: A tool for execution time estimation of manual test cases. *Journal of Systems and Software*, 146, 26-41.

Trautsch, F., Herbold, S., & Grabowski, J. (2020). Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects. *Journal of Systems and Software*, 159, 110421.

Wang, Y., Zhu, Z., Yu, H., & Yang, B. (2018). Risk analysis on multi-granular flow network for software integration testing. *IEEE Transactions on Circuits and Systems II:Express Briefs*, 65(8), 1059-1063. DOI:10.1109/TCSII.2017.2775442.

Yang, Y., Li, Z., Shang, Y., & Li, Q. (2021). Sparse reward for reinforcement learning-based continuous integration testing. *Journal of Software: Evolution and Process*, e2409.

Zhang, M., Keung, J. W., Chen, T. Y., & Xiao, Y. (2021. Validating class integration test order generation systems with Metamorphic Testing. *Information and Software Technology*, 132, 106507.

Zhang, M., Keung, J. W., Xiao, Y., & Kabir, M. A. (2021). Evaluating the effects of similar-class combination on class integration test order generation. *Information and SoftwareTechnology*, 129, 106438.

Zhang, M., Jiang, S., Zhang, Y., Wang, X., & Yu, Q. (2017). A multi-level feedback approach for the class integration and test order problem. *Journal of Systems and Software*, 133, 54-67.

Zhang, M. Z., Gong, Y. Z., Wang, Y. W., & Jin, D. H. (2019). Unit test data generation for c using rule-directed symbolic execution. *Journal of Computer Science and Technology*, 34(3), 670-689.

Zhang, Y., Jiang, S., Wang, X., Chen, R., & Zhang, M. (2019). An optimization algorithm applied to the class integration and test order problem. *Soft Computing*, 23(12), 4239-4253.