International Journal on New Computer Architectures and Their Applications (IJNCAA) 1(1): 74-83
The Society of Digital Information and Wireless Communications, 2011 (ISSN 2220-9085)

Evaluating Software Maintenance Testing Approaches to Support Test Case Evolution

Othman Mohd Yusop and Suhaimi Ibrahim
Advanced Informatics School
Universiti Teknologi Malaysia, *International Campus*
54100 Jalan Semarak, Kuala Lumpur, Malaysia
{othmanyusop, suhaimiibrahim}@utm.my

ABSTRACT

Software Maintenance Testing is essential during software testing phase. All defects found during testing must undergo a re-test process in order to eliminate the flaws. By doing so, test cases are absolutely needed to evolve and change accordingly. In this paper, several maintenance testing approaches namely regression test suite approach, heuristic based approach, keyword based approach, GUI based approach and model based approach are evaluated based on software evolution taxonomy framework. Some of the discussed approaches support changes of test cases. Out of the review study, a couple of results are postulated and highlighted including the limitation of the existing approaches.

KEYWORDS

Maintenance Testing, Test Case, Test Suite, Software Change, Software Evolution.

1 INTRODUCTION

Maintenance testing as defined in ISTQB glossary terms (standard glossary terms ver2.0) "*testing the changes to an operational system or the impact of a changed environment to an operational system*". There are two type of maintenance testing that relates to changes in artefacts during the maintenance phase: **confirmation testing** and **regression testing**. Maintenance testing phase happens after the deployment of the system. Over time, the system is often changed, updated, deleted, extended, etc during software evolution. The artefacts that support the system need to be updated concurrently to avoid being outdated as compared to the source codes. [1] shows 80% of overall testing budget went to retesting the software and 50% of total software maintenance is consumed by retest alone.

Confirmation Testing can briefly define as a re-testing. Defects found during testing will be corrected and another test execution will take place to re-confirm the failure does not exist. During the retest, originality of test environment, data and inputs have to be exactly identical as it was tested in the first time. If the confirmation testing has passed, it does not guarantee the defect has been corrected. It might introduce defects somewhere else, hence **regression testing** is required. In order to ensure the defect does not propagate to other functionalities, **regression testing** has to be carried out.
More specifically, the purpose of **regression testing** is to verify that modifications in the software or the environment have not caused unintended

adverse side effects and that the system still meets its requirements.

Regression testing is a technique to validate the unchanged system features whether it is remained intact and not impacted by the recent changes made to the system or otherwise. Whenever there is a new version of software being produced, all the test cases in a regression test suite would be executed. This technique is ideal candidate for automation. However, **regression testing** will become slower to be executed when newly added test cases increase a number of existing test cases. Therefore to reduce the cost of **regression testing**, **test case selection** is required.

In this paper some of the maintenance testing approaches are evaluated to find the commonalities among the approaches and to what extent these approaches provide support within entire spectrum of software development activities specifically during maintenance testing.

This paper is organized as follows: Section 2 presents an overview of software maintenance approaches. Section 3 elaborates software evolution framework perspective while section 4 discusses two results obtained from the study. Section 5 talks about threats to validity and finally followed by the conclusion section 6.

2 OVERVIEW OF SOFTWARE MAINTENANCE TESTING APPROACHES

In this section several maintenance testing approaches namely Regression Test Suite approach, Heuristic-Based Framework approach, Keyword-Based approach, Graphical User Interface (GUIs) Regression Testing approach and Model-Based approach will be evaluated into several subsections respectively.

2.1 Regression Test Suite Approach (RTS)

When changes applied in to a system, impacted artefacts i.e. test suites have to be changed accordingly during maintenance phase. **[2]** came up with a case study how test suite maintenance can be done during system evolution that caused by changes made to the system during maintenance phase. This case study make used of reusable test environments and program. **[2]** investigated issues in software maintenance through exploratory study and follow-up study on change strategies in their studies and this study was inspired by **[3]**.

Four phases involved during the case study process: *environment setup*, *build configuration*, *execute unit test and execute functional test* as shown in Fig. 1 below:

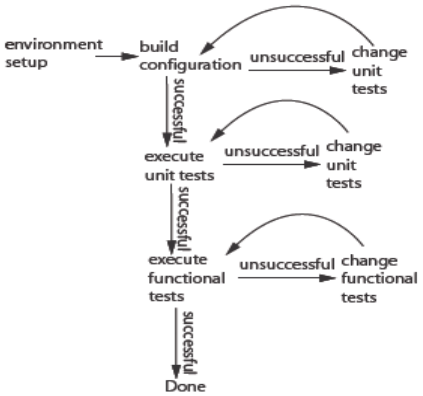

Figure 1. Case Study Process [2].

Several steps were involved during the test. First step involves installation of

baseline version and follows by executing the baseline to verify its executable and usable. Change propagation version will follow the suite i.e. installation and execution processes. The adapting task will be logged and recorded. The results will undergo a comparative study with different version of the test cases. Validation of this work is done via a model called Padgett [4]. The result than postulated into three classifications: *reactivity*, *researcher bias* and *respondent bias*.

2.2 Heuristic-Based Framework Approach (HBF)

Graphic User Interfaces (GUIs) are believed to make up a large portion of source code [5]. Testing GUI is different compared to traditional testing which involve implementation level of source code/programming. This is a true when it comes to test case maintenance especially during regression testing. Minor changes in GUI, could impact malfunction of test cases. [5] has created an approach which is based on heuristic model to solve GUI test case maintenance.

This approach make used of two techniques: *Capture/Replay* and *Elements and Actions*. *Capture/Replay* technique requires end-user gestures such as mouse maneuverability usage and keystrokes. These activities are recorded and played back. The advantage of this approach, it does not require a good programming skill. The issue of this approach raise when there is change in the interface and consequently causing test cases to be malfunctions and broken. Whenever this situation happens, manual effort is often required to repair some test cases in test suite. *Elements and Actions* approach models a GUI test case as a sequence of actions. Examples are shown as in Fig. 2 and Table 1. Fig. 2 shows a sample of GUI as captured test cases whereas Table 1 shows the elements and actions captured during the *capture/replay* technique

Figure 2. Find Dialog Box [5].

Table 1. Sample of Test Case for Find Dialog Box [5].

GUI Elements	*Actions*
FindTextBox	setText('GUI')
CaseSensitiveCheckBox	'click'
FindButton	'click'
CancelButton	'click'

In Fig. 3 shows the captured new elements and actions. While in table 2 shows the sample heuristics table.

Figure 3. Modified Find Dialog Box [5].

Table 2. Sample Heuristic Result [5].

Heuristic Name	*Description*
SameLabel	Element has the same label.
SamePreviousSibling	Element has the same previous sibling element.
SameNextSibling	Element has the same next sibling element.
SameType	Element is of the same type (button, checkbox, etc.).

Changes are applied to system and some of new elements in Find Dialog box have been updated as Fig. 3. Some of these changes causing test case as Table 1 invalid unless the test case need to be updated as well. **HBF** approach has to be produced so automated test case can be formed accordingly to the changes made as shown in table 2.

2.3 Keyword-Based Approach (KB)

[6] used keyword-based approach for software testing automation and maintenance. They categorised test automation into 5 categories namely: test management, unit test, test data generation, performance test and functional/system/regression test. The authors picked up test execution for test automation. Basic principle of this keyword based approach is test engineering tasks are separated into specific roles. Identified roles for test engineering composed of *test designer*, *automation engineer* and *test executor*.

As *test designer*, the person needs to form test cases using keywords and documented in using spreadsheet. The *automation engineer* will code up the keywords scripts using scripting tool and language. Finally the *test executor* will run the tests directly from the spreadsheet. The approach is said to improve Capture/Playback technique through reduction the amount of test script. The approach is as shown in Fig. 4 below and follows by the test result (Fig. 5) of the research study by the authors.

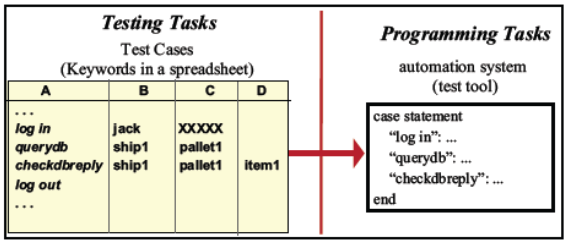

Figure 4. Keyword Based Approach [6].

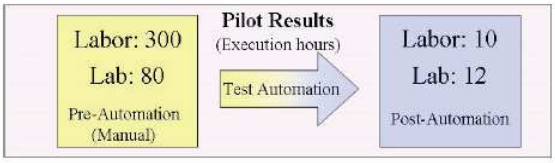

Figure 5. Result as Return of Investment [6].

2.4 Graphical User Interfaces (GUIs) Regression Testing Approach (GUIs-RT)

[7] claims the test case maintenance using GUI is approachable and not many research have been done on it. This approach basically provides some useful insight information on test suite maintenance through GUI maintenance. This approach is called GUI regression testing and it determines the usability of test suites after changes are imposed on the system GUIs.

This technique consists of two parts: *a checker* and *a repairer*. *A checker* is responsible to categorise test cases into usable and un-usable. If the test case is siding to the latter, *the repairer* will try to repair the un-usable test cases. Once done, the repaired test cases are labeled and stored as repairable test case. Details of the Fig.6 as shown below:

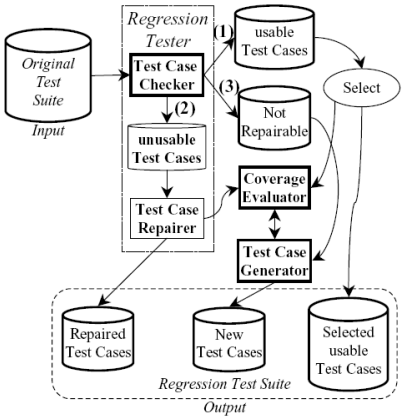

Figure 6. Regression Testers' Components [7].

[7] Repaired test case is an effective way to reduce cost of creating new test cases. Each component details have been explained by the authors in their research studies and will not recurring into this sub-section. Results are compared through several case studies and Bit-vector and Graph Matching Checker execution time were taken as shown in Table 3 and Fig. 7 respectively below:

Table 3. Time Taken at a Glance [7].

Step	Subject Application	Time
Manual Generation	Reader	7.59 hrs.
	WordPad	5.47 hrs.
Checker	Reader	6.5 sec.
	WordPad	6.15 sec.
Repairer	Reader	17.76 sec.
	WordPad	18.01 sec.

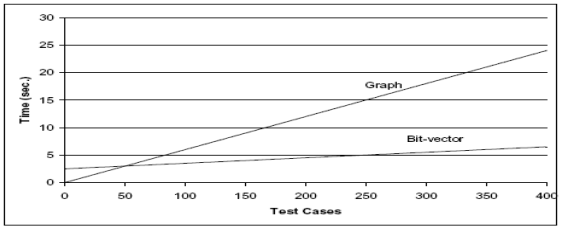

Figure 7. Comparing Bit-Vector with Graph Matching Checker [7].

2.5 Model-Based Approach (MB)

[8] proposed a model-based approach for maintenance testing. Models are the main source for this approach and tools that support models and source codes are presumably established i.e. support auto generation between models and source code. UML classes and sequence diagrams are two input factors for test case generation. Whilst generating test cases out of models, an infrastructure composing of test related model and fine-grained traceability are created. The infrastructure or the approach as depicted in Fig. 8 below:

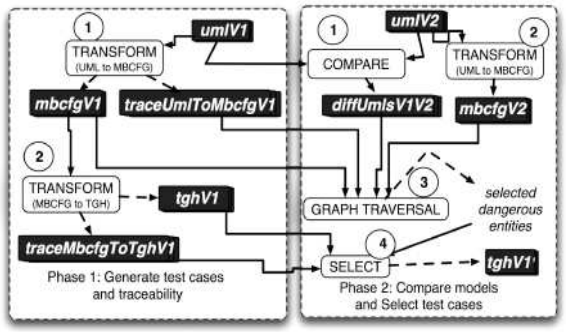

Figure 8. The Approach Overview [8].

This approach is divided into two phases. *First phase* is the creation of models and traceability, and the *second phase* utilises created models and traceability from the first phase as well as the modified UML models. During *first phase*, two steps are executed: (1) sequence diagram into model-based control flow graph (*mbcfg*) transformation and (2) then converting *mgcfg* information into test generation hierarchy and keeps safe the traceability model. Abstract test cases are produced and during further transformation the abstract test cases will turn into concrete test script skeletons.

The second phase involves four activities: (1) comparing the models to find differences hence differencing model, (2) converting sequence diagram into modified *mbcfg*, (3) *mbcfg* and differencing model will used during pair wise graph traversal between original

and modified *mbcfg*, and finally (4) test cases are classify through selected dangerous entities identification. This approach support modification at model level i.e. classes and sequence diagrams.

3 EVALUATION OF SOFTWARE EVOLUTION FRAMEWORK

[9] taxonomy provides a framework for software evolution. This framework is not focusing on *why* the changes took place or *who* involves with the changes, instead it answers other non-trivial aspect of changes such as mechanisms of change and influence factors through investigating of *how*, *when*, *what* and *where*, software has changed. The framework is a conceptual ground that supports various evolution mechanisms and tools that emerged due to changes in software system.

The taxonomy is composed of features of consideration called dimensions. These dimensions are determined and classified into either characterising mechanisms or influencing factors. Each of these dimensions will be placed under four types of logical groups: *temporal properties*, *object of change*, *system properties* and *change support*. These logical subjects and dimensions stated aspects of software changes in term of *when*, *where*, *what* and *how* [9].

Temporal properties group contains time aspect of ***when*** evolution changes happened and its frequent occurrence. Under this group, *time change* dimension has been specified into three different phases of change: static, load-time and dynamic; *change history* refers to history of changes made to the software both parallel or sequential history and it has to be supported by versioning tool; *change frequency* property states changes interval into periodically, continuously or arbitrarily; and *anticipation* describes a foreseen changes at early stage of development i.e. requirement phase [9]. Anticipation change helps reduce effort of implementing changes compared to unanticipated change [10].

Object of change describes the location of ***where*** changes are made. Supporting mechanisms that are needed: *artefacts*, it could be static artefacts or dynamic artefacts; *granularity* of artefacts, from fine-grained, medium and coarse; *impact of change* determines range of impacted artefacts; and *change propagation*, following up the changes if they span to non-local artefacts or different level of abstraction [9].

System properties indicates the ***what*** part and it composed of; *availability* that indicates either the software system is permanently or occasionally available; *activeness* state the system either it is reactively or proactively evolved; *openness* indicates how open and closed the system to new extensions; and finally the *safety* is a feature to distinguish between static safety and dynamic safety[9].

The *change support* describes the ***how*** part. Some features involve in this logical theme are; *degree of automation*, is a feature to differentiate among fully automated, partially automated or manually change; *degree of formality* indicates the degree of formalism used during the change process and formal methods [11] may be used during the process; *change type* characterises

between structural and semantic changes.

4 RESULT REVIEWS OF MAINTENANCE TESTING APPROACHES

Evaluation of the existing maintenance testing approaches will be done by benchmarking against [9] software evolution framework. [9] defined their framework criteria based on characterising factors of change mechanism and its influence factors.

For this particular review, research candidate has done cross checking against one of the dimensions namely *Object of Change* (*where*). Other dimensions are not going to be included in this review. This is due to research candidate nature research is based on scopes such as change of artefacts, granularities, impact and change propagation.

4.1 Result Review I

[9] classified the location of changes or answering *where* changes happened, as the second logical of his taxonomy. Within this dimension, there are four influencing factors had been highlighted namely *artefacts*, g*ranularity*, *impact* and *change propagation*. *Artefacts* are sub-categorised into static evolution and dynamic evolution. Level of artefacts abstraction will be divided into three sub-categories *granularity*; coarse, medium and fine. The *impact* of the changes indicates whether the changes influencing at local or global/system-wide level of abstraction. Changes made can spread out to other entities; this influencing factor is called *change propagation*.

Table 4. Object of Change for Maintenance Testing Approaches – Result Review I.

Object of Change	RTS	HBF	KB	GUIs-RT	MB
Artefacts	Test Suites	Test Cases	Test Cases	Test Cases	Classes and sequence
• *Static Evolution*	☑	☑	☑	☑	diagrams ☑
Granularity					
• *Coarse*	☑		☑	☑	
• *Medium*					☑
• *Fine*		☑			
Impact					
• *Locally*	☑	☑	☑	☑	☑
• *System Wide*	☑	☑	☒	☑	☒
• *Level of abstraction*	Test Cases	Graphic User Interface	Testing Script	Test Suite	Model
Change Propagation	☑	☒	☒	☒	☑

Based on table 4, all maintenance testing approaches are checked against the object of change dimension. For influencing factor artefacts, **RTS** caters set of test cases or a test suite. The granularity of the artefacts is still coarse or at file level. Though the impact can span from locally impacted or system wide impacted. **RTS** does cover the change propagation by having change propagation cycle in its approach.

HBF has similar set of artefacts when compared to **RTS**. **HBF** goes a little detail on level of artefacts which is test case instead of test suite. Since **HBF** covering element of artefacts based on graphical user interface, each of the field inside the window is treated as a single possible test case, thus **HBF** provides finer granularity as opposed to **RTS**. The impact can be traced out locally or widely despite no change propagation tracing was provided.

KB uses keywords to build up test cases. The level of test case is still coarse in term of its granularity. All the keywords are typed out manually into a test script and will be used to assist a programming task during test automation. It has locally impact and does not support change propagation.

GUIs-RT is another approach which took graphical elements as its test cases artefacts. The focus is more towards maintaining user interface therefore all test cases are sourced by graphical elements. Level of granularity is still at coarse level and the impact is able to trace out locally and widely except with no change propagation.

Finally **MB** approach make used of model as input artefacts. Specifically classes and sequence diagram will be used as the sources. Two UML elements are considered as medium granularity and the impact can be traced out locally with change propagation support.

4.2 Result Review II

Besides evaluating the approaches based on [9] taxonomy perspective, research candidate did some comparative study among the maintenance testing approaches. Some generic features or commonality namely contribution, limitation, level of abstraction, traceability support, version control support, tool support, result of research and validity to threat were included and tabulated as the following table 5:

Table 5. Maintenance Testing Approaches Commonality Features – Result Review II.

Commonality Features	RTS	HBF	KB	GUIs-RT	MB
Contribution	Test Suite Automation	Support GUI test cases	Support Test Automation through Keywords	Support Test Suite maintenance during GUI regression testing method	Model based test case generation and maintenance
Limitation	Manually execution for encapsulated function	Larger GUI size causes adverse effect to its accuracy	Scalability Issues	Obsolete test cases still can occured	If more modified operations were called, adverse in precision

Level of Abstration	Test Suite	GUI elements i.e. buttons, textbox	Words	Test Suite	Models, classes, source code
Traceability Support	×	×	×	×	✓
Version Control	×	×	×	×	×
Tool Support	×	GUIAnalyzer	×	GUITAR	RTStool & DejaVOO
Validity to Threat	Padgett model developed by [4] to validate the approach	Not stated	Capture/Playback was comparative case study	Subject application, Performance and Effectiveness, comparative algorithms	Using these criteria: efficiency, precision and safety [12]
Result of Research	*Role splitting* strategy is favourable over the others	Accuracy can be sought using multiple heuristic sets	Comparatively Keyword is better that Capture/Playback approach which is very costlier and manually done	Efficient and effective new regression GUI test approach	Combinational between model-based test generation and regression selection test

5 THREATS TO VALIDITY

The sources used to evaluate the approaches are based on accepted published international journals and conference papers. Justifications are based on the software evolution taxonomy mentioned in [9]. Whereas another evaluation was based on common criteria such as limitation, contribution, etc. Though without any formal methods, the results were based on our understanding and experiences in the field before such results were postulated.

6 CONCLUSION

In this paper we presented the evaluation results for maintenance testing approaches. These inputs can be used for future references in case research candidate wishes to further the research study.

7 REFERENCES

1. Harrold, M.J.: Reduce, reuse, recycle, recover: Techniques for improved regression testing. 2009 IEEE International Conference on Software Maintenance. pp. 5-5. , Edmonton, AB, Canada (2009).
2. Skoglund, M., Runeson, P.: A case study on regression test suite maintenance in system evolution. 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. pp. 438-442. , Chicago, IL, USA (2004).
3. Rajlich, Gosavi: A Case Study of Unanticipated Incremental Change. Proceedings of the International Conference on Software Maintenance (ICSM'02). p. 442. IEEE Computer Society (2002).
4. Robson, C.: Real World Research: A Resource for Social Scientists and Practitioner-Researchers (Regional Surveys of the World). {Blackwell Publishing Limited} (2002).
5. McMaster, S., Memon, A.M.: An Extensible Heuristic-Based Framework for GUI Test Case Maintenance. Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops. pp. 251-254. IEEE Computer Society (2009).
6. Wissink, T., Amaro, C.: Successful Test Automation for Software Maintenance. 2006 22nd IEEE International Conference on Software Maintenance. pp. 265-266. , Philadelphia, PA, USA (2006).
7. Memon, A.M., Soffa, M.L.: Regression testing of GUIs. Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on

Foundations of software engineering. pp. 118-127. ACM, Helsinki, Finland (2003).
8. Naslavsky, L., Ziv, H., Richardson, D.J.: A model-based regression test selection technique. 2009 IEEE International Conference on Software Maintenance. pp. 515-518. , Edmonton, AB, Canada (2009).
9. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change: Research Articles. J. Softw. Maint. Evol. 17, 309–332 (2005).
10. Kniesel, G., Noppen, J., Mens, T., Buckley, J.: Unanticipated Software Evolution. Object-Oriented Technology ECOOP 2002 Workshop Reader. pp. 92-106-106. Springer Berlin / Heidelberg (2002).
11. Goguen, J.A.: Formal methods: Promises and problems. IEEE SOFTWARE. 14, 73--85 (1997).
12. Rothermel, G., Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Trans. Softw. Eng. 22, 529-551 (1996).