PRIORITIZED BLACK BOX TESTING USING GENETIC SOFTWARE ENGINEERING METHODS

Riham Hassan‡1, Hicham G. Elmongui† ‡2, Yasmine Ibrahim*3

‡Computer Science, Virginia Tech, Blacksburg, VA, USA
§Computer and Systems Engineering, Alexandria University, Alexandria, Egypt

*Arab Academy for Science and Technology, Cairo, Egypt

1 rhabdel@vt.edu, 2 elmongui@alexu.edu.eg , 3ysalem@cairo.aast.edu

ABSTRACT
Software regression testing is a critical and intensive phase in the software development life- cycle. In this paper, we propose, RECAP a testing technique that derives regression test cases systematically from semi-formal requirements. RECAP provides means to ensure test coverage of requirements. Moreover, it prioritizes the test cases according to the requirements priorities in order to maximize the customer satisfaction and minimize the cost of regression testing without reducing the quality of test. RECAP also provides sufficient information to trace each test case to its requirements, which reduces the error-proneness of the test cases while enhancing the testing coverage. We demonstrate the effectiveness of RECAP using a case study and an experimental study. The results show better test case coverage of requirements and fault detection for requirements with high priority compared to classical testing techniques.

KEY WORDS
Test Case Derivation, Test Case Prioritization, Genetic Software Engineering, Black Box Testing

1. Introduction

With software prevailing different aspects of our life, the need for stronger verification techniques assuring the software quality ascends to a top priority in the software engineering life- cycle. Performing extensive testing on a product raises the confidence in the software quality; nevertheless impacts the schedule and budget of the software.
Developing effective test scenarios is a key factor for producing a system that satisfies the user requirements. However, developing test cases with complete and consistent coverage of the system requirements has always been a challenge [14]. A better approach to raise the quality of the test cases is to derive (instead of develop) them against software requirements. Specifically, test cases might be automatically generated from a requirements model or a design model [14]. This approach would enhance the requirements coverage.

Further, the derived test cases would be testing the represented requirements while minimizing the risk of misinterpretation or missing requirements, as the derivation process is not dependent on the tester skills [10].
Deriving test cases from informal requirement representation is the major source of incomplete and inconsistent test coverage as informal requirement representation could lead to missing or misinterpreted requirements [1, 10, 16]. The solution is to systematically derive the test cases from more rigorous requirements models.
Test case prioritization has been suggested in the literature as a solution to reduce the testing time and promotes the software quality (e.g., [2, 6, 9, 18]). Prioritizing test cases based on prioritized requirements ensures customer satisfaction while reducing the software testing time [15].
In this paper, we propose the Requirements-based tEst Case generAtion and Prioritization algorithm (RECAP) to systematically derive prioritized regression test cases from a set of functional requirements modeled with the Genetic Software Engineering (GSE) method [7]. RECAP addresses the system and regression testing phases with the objective to raise the confidence the coverage of the derived test cases to the customer requirements. Modeling requirements with GSE addresses various requirements problems including requirements inconsistency, incompleteness, and the high cost of change. In this paper, we propose the Requirements-based tEst Case generAtion and Prioritization algorithm (RECAP) to systematically derive prioritized regression test cases from a set of functional requirements modeled with the Genetic Software Engineering (GSE) method [7]. RECAP addresses the system and regression testing phases with the objective to raise the confidence the coverage of

the derived test cases to the customer requirements. Modeling requirements with GSE addresses various requirements problems including requirements inconsistency, incompleteness, and the high cost of change. RECAP derives effective regression test cases through providing test cases that promises reasonable requirements coverage, as they are derived from requirements represented by a semi-formal model. Due to test case prioritization in cooperated in RECAP, it helps reducing the testing time and cost by decreasing the numbers of tests to be made. It provides sufficient traceability information to trace each test case to its requirements, which enhances the requirements coverage. Further, it promotes customer satisfaction as it propagates requirements priorities (specified by the different stakeholders) to regression tests.

The rest of the paper is organized as follows. Section 2 reviews the current state of the art in test case generation and prioritization. Section 3 gives an overview on the Genetic Software Engineering (GSE) approach, We depict and demonstrate the RECAP technique using a case study in Section 4. Section, validates the RECAP technique through an experimental study. Finally, Section 6 concludes the paper.

2. Related Work

Test case derivation and prioritization techniques have been proposed in the literature and can be classified to three categories. The first category includes techniques that only derive test cases like [4, 5, 8, 10, 16, 17]. The second category includes techniques that only prioritize test cases like [12, 15]. The third category that we focus on in this section both derives and prioritizes test cases. The proposed RECAP technique falls in this category of solutions.

In the three categories, the generation of test cases is typically based on UML, classification trees or genetic algorithm, whereas, prioritization techniques rely on the weight given to the requirements represented in semi-formal methods based on certain criteria.

Marini proposed a generation and prioritization technique for COTS (Commercial, off- the-shelf) components in which components are monitored for their interactions followed by an automatic synthesis of the behavior model using BCT technology (Behaviour, Capture and Test) [6]. Rajappa generates and prioritizes test cases based on a genetic algorithm combined with graph theory. However, the high complexity and time consumption of the technique makes it more suitable for high integrity systems [11]. The Cow-Suite tool introduced in [2] derives and prioritizes test cases based on UML models. The UML models are analyzed to explicitly define associations and relations among the developed use cases and the involved actors to form a graph representing the design models. The work proposed in [18] derives and prioritizes test cases using enhanced classification trees to guide the tester towards the determination of test cases. Sapna et al. generate and prioritize test cases based on control and data flow from UML state diagrams [12]. The inconsistence and misinterpretation of requirements may be detected better due to different people building models for development and testing. The disadvantage in this case is the effort involved in developing two different models

RECAP falls in the third category as it derives and prioritizes test cases. Similar to several mentioned test case derivation techniques RECAP derive test cases from requirements represented in a semi-formal model. In addition, RECAP provides traceable information between each test case and its requirements. Furthermore, RECAP prioritization; likewise other prioritization techniques address the early detection of faults. However, RECAP is concerned with achieving customer satisfaction through propagating requirements priorities (specified by customer needs) to regression tests.

3. Genetic Software Engineering (GSE)

In this section, we briefly illustrate the Genetic Software Engineering (GSE) method. We employ GSE in RECAP to represent the requirements model from which the regression test cases are derived. GSE addresses the challenge of developing software systems that meet their functional requirements and constraints [7]. GSE adopts the genetic engineering principles in building the system out of its requirements, whereas conventional software engineering builds the system to satisfy its requirements. A system built out of its requirements enables satisfying the weaker goal of conventional software engineering, which is, “will such a design satisfy its requirements?” [7].

In GSE, each functional requirement, expressed in natural language, is represented formally as a Requirement Behaviour tree (RBT). GSE adopts the behavior tree notation as a solution to a fundamental problem of going from a set of functional requirements to a design that satisfies those requirements since it provides a clear, simple, constructive and systematic path for this transition [7].

RBT is a formal, tree-like graphical form that represents behavior of individual or networks of entities. Such entities could realize or change states, make decisions, cause/respond to events, and interact by exchanging information and/or passing control.

An RBT provides a direct and clearly traceable relationship between a requirement expressed in natural language and its formal specification. Translation is carried out on a sentence-by-sentence basis. For example, in the Microwave Oven System (MOS) [7], requirement 1 (R1) is expressed with the sentence “when the door is closed, the button is enabled” is translated to the behavior tree in Figure 1(a).

Requirement 2 (R2) is expressed with the sentence" Closing the door turns off the light. This is the normal idle state prior to cooking when the user has placed the food in the oven". R2 is translated to the RBT in Figure 1(b).

RBTs of individual functional requirements might be composed, one at a time, to create an integrated Design Behaviour Tree (DBT). GSE defines two axioms namely the precondition axiom and the interaction axiom to

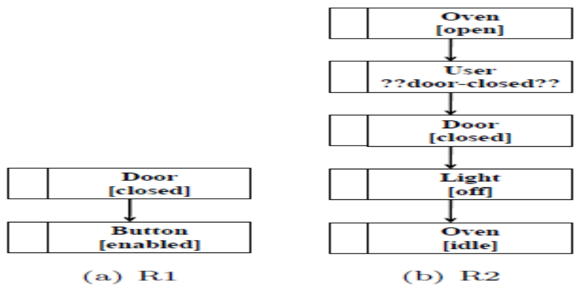

Figure 1 RBTs Example

delineate the relationship between the individual RBTs during the integration process. According to the precondition axiom, every individual functional requirement, expressed as a behaviour tree, has a precondition. The requirement precondition has to be satisfied in order for the behavior encapsulated in the functional requirement to be applicable. According to the interaction axiom, the precondition of every requirement has to be established by at least one other

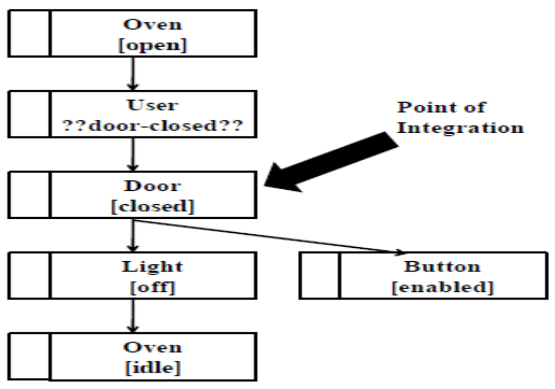

Figure 2: RBTs Integration

By using the precondition and interaction axioms, GSE has the ability to detect missing or inconsistent requirements during the integration of individual RBTs. The root node of each RBT is checked to occur in another RBT. If it does not, it is one of four possibilities: 1) it is the root node of the whole system, 2) it is missing a precondition, 3) the set of requirements is incomplete, or 4) there is a behaviour missing from or implied by the requirement it needs to integrate with. Providing such integration checks raises the confidence in the completeness and consistency properties of the resulting requirements model.

4. RECAP Overview

In this paper, we propose RECAP that derives prioritized regression test cases systematically from a DBT requirements model constructed with the GSE method. Deriving test cases systematically from the DBT formal requirements model strengthens the evidence of requirements coverage by the resulting test cases. Further, the systematic derivation of test cases from requirements allows for a uniform distribution of test cases over requirements. Every requirement is tested by one or more test cases while every test case involves one or more requirement.

The prioritization of the resulting test cases according to the requirements priorities enables RECAP to increase the rate of fault detection and, therefore, decrease software testing time. Further, propagating requirements priorities to the testing phase raises customers' satisfaction as they could ensure that their needs are well accommodated in the testing process.

The RECAP test case derivation algorithm provides sufficient lineage information to trace each test case to its requirements. Lineage information, which enables software developers to trace defects to their requirements, reduces the cost of locating the defect origin significantly. Furthermore, it maps the change of requirements to the test cases.

We have developed an automation framework for RECAP to ensure testing coverage of requirements. The quality of the regression test cases produced from classic testing techniques rely heavily on the expertise of the test case developer as the process is purely manual. RECAP avoids such quality variation through its automation framework. Further, automating the test case derivation decreases software testing time and eliminates human errors.

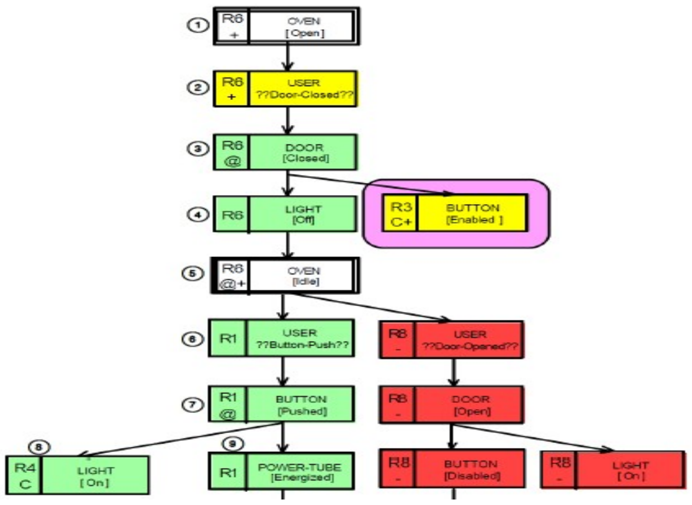

Figure 3 DBT Model of Microwave Oven

4.1 RECAP Workflow

RECAP is primarily composed of three major steps to derive prioritized regression test cases:1) locating a test case in the DBT, 2) storing that test case in the test suite and 3) prioritizing the identified test cases within the suite.

We have demonstrated the effectiveness of RECAP on a number of case studies. Our rationale for the selection of those case studies is that they are of reasonable size, large enough to be convincing and small enough to be manageable. Further, these cases studies have been examined before to demonstrate software engineering approaches, which increases their credibility and reliability. Due to space limitation, we show the multiple RECAP steps on one case study, namely the MOS case study [7].

Shlaer and Mellor introduced the MOS case study that describes the operation of an automated microwave oven system [13]. MOS has also been employed to demonstrate the effectiveness of GSE [7]. Figure 3 represent the MOS DBT model as constructed by Dromey in [7].

4.1.1 Locating a Test Case

RECAP derives a suite of regression test cases through traversing the DBT model. The DBT model is composed of a set of nodes that hold either a state or an event. These nodes are rooted at the node representing the initial state of the system. These nodes confine all the possible system states and the events that trigger a system change from one state to the other. Therefore, system behavior could be extracted from the DBT model in the form of scenarios. Each scenario is rooted at a sub-tree of the DBT model detaining its precondition state, its different steps in the form of events, and its post condition state. The post condition state is the state the system will move to upon the execution of the scenario. Such scenarios would directly serve as regression test cases derived directly from the requirements model. Precondition node(s) are those that hold a state and don't follow an event node directly as they represent the system state prior to a change due to some events. Post condition node(s) are those that hold a state and follow an event node directly as they represent the system state that results from some events.

The algorithm traverses the DBT to locate the test cases and add them to the test suite. The parsing algorithm visits each node and checks for three conditions, which are: 1) it is not the root node, 2) it holds a state and 3) an event has been encountered, which sets a variable isEncounteredEvent to true. The algorithm starts at the root node of the tree and checks the three conditions since it is a root node. The algorithm then moves to the second node that is of type event indicated by the "??Door-Closed??", so the isEncounteredEvent is set to true. The third node holds a state "Door (Closed)", and isEncounteredEvent is true, so the algorithm keeps searching for a node that holds an event to finalize the test case. The sixth node holds the first encountered event "??Button- Push??", so the algorithm cuts the tree right before the sixth node, at the fifth node that holds the state "Oven (Idle)". The fifth node becomes the post condition node of the current test case. The next path to be parsed by the depth first search algorithm is the one that node 1, Oven (Open), to accommodate the other branch of node 4, Button (Enabled). By doing so, the parsing algorithm exhausts all the possible paths of the DBT.

4.1.2 Storing a Test Case in the Test Suite

The sub-tree representing the test case is composed of three major elements namely the precondition, scenario and post condition. After locating a test case, we add it to the test suite in the form of a test case structure. This structure contains the three elements confined in the test case sub-tree. Additionally, the test case structure contains an ID given to each test case and the requirements numbers covered by this test case. The requirements numbers are obtained from the nodes of the test case sub-tree that store the requirements numbers in their tags.

In MOS the first node of the sub-tree being a root node is stored as the test case precondition. The content of the following node is saved as the scenario as it holds an event and the isEncounteredEvent flag is true. The third node holds a state and the isEncounteredEvent is true, so the content of the node is saved as the test case post condition. The fourth and fifth nodes hold states, so they are appended to the post condition. At this point, the test case structure is filled with the content of the test case sub-tree. In this test case, R6 is the requirement associated with three nodes of the post condition. The requirement(s) number(s) stored with each test case is used to prioritize the test case and trace it back to the requirement(s) it tests.

4.1.3 Prioritizing the Test Suite

Once the parsing algorithm finishes identifying all the test cases by visiting all the nodes in the DBT, the final step in RECAP is to prioritize the test cases stored in the test suite so far. RECAP prioritizes the test cases based on the software requirements priorities, which are typically specified by the stakeholders. Each requirement is given a weight by each stakeholder based on its importance from the stakeholder's perspective on a normalized scale from 0 to 1. The final priority value of each requirement is calculated based on a weighted average function of all the weights given by the different stakeholders. We adopted a weighted average function to allow for giving different weights to the perspective of some stakeholders. For example, the customer perspective of the requirement priorities could be higher than the project manager perspective. We highly recommend involving multiple stakeholders' viewpoints in the requirements prioritization calculation process. Requirements with higher total weights are given higher priorities. Propagating the perspective priorities of multiple stakeholders to regression testing increases its reliability. The weighted requirement priority (WRP) function is

$$\sum_{r=1}^{n} WRP = (WCP * (i)) + (WPMP * (i)) + (WDP * (i$$

The (WRP) for every requirement r till n, is the sum of the weighted priority given by the customer (WCP), the project manager (WPMP), and the developer (WDP). The different perspective of the stakeholder is measured by i,

MOS requirements were not prioritized in the literature, so we have calculated the priorities of MOS requirements using weighted average from developer, manager and customers to serve our RECAP demonstration purposes . Further, we assume the perspectives of the three stakeholders are weighted equal. Therefore, the priority of any given requirement is the average of the three stakeholders' priorities. The average weighted requirement priority AWRP is calculated as follows, where z is the number of available stakeholders.

$$AWARP = \frac{WARP}{z}$$

In MOS case study test case 1 that we have shown earlier is associated with R6 that has the priority of 0.8. Therefore, test case 1 is assigned the priority value 0.8. In case multiple requirements are associated with a single test case, the highest requirement priority is assigned to the test case priority. Assigning the highest priority to the test case allows RECAP to promote testing of the higher priority requirements prior to the lower priority requirements

4.2 The RECAP Automation Framework

We developed the RECAP automation framework to automate the test case derivation process from the DBT model. The automation process contributes to the reduction of the software testing time along with the prioritization of test cases. Further, automating the derivation process eliminates human errors. Figure 4 depicts the automation framework of RECAP.
We created an XSD schema for the DBT model and another XSD for the resulting test cases. The XML of the DBT is the input to the automation framework while the XML of the test cases is the output.
The automation framework is composed of three phases. The first phase parses the DBT model in a depth first traversal and stores the nodes in an array structure. The second phase locates the test cases and adds them to the test suite. The test suite output is written to an XML file. The third phase prioritizes the test suite based on prioritized requirements. The test case prioritization logic relies on the priorities of the requirements associated with each test case.

5. Validation Experiments

Experimental approaches provide an attainable opportunity to validate novel software engineering approaches. However finding adequate systems for pilot studies to evaluate software approaches like RECAP is a complicated process. The candidate system should provide its artifacts including programs, requirements document, and fault data. Obtaining such material is a nontrivial task [3]. Free software is accessible, but it doesn't provide requirements document or defects data.

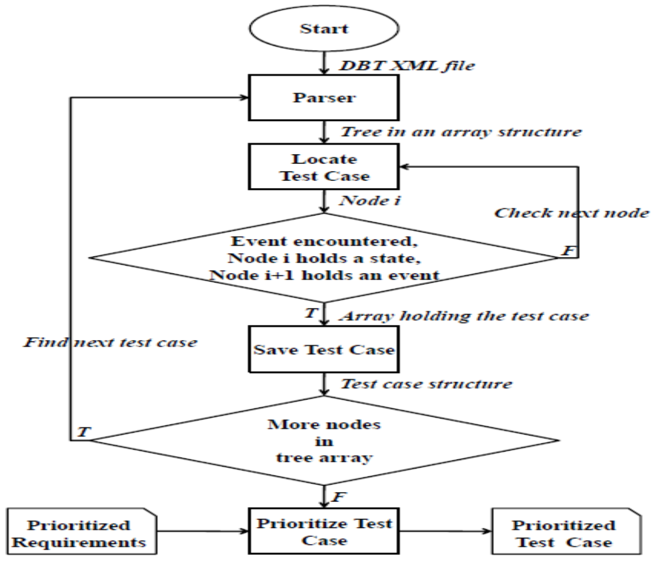

Figure 4 RECAP Automation Framework

establish requirements documents, are often reluctant to release their requirements, source code, test suits, and defects data to researches [3]. A reasonable and manageable experimental validation could be obtained through a study on a student context project that has defects data and existing requirements [3]. The results of such study should highlight the strength and potential problems in RECAP. Moreover, using a student context project for validation RECAP can enable a future commercial study, because of the validated strength can increase the credibility of RECAP.

We have utilized an existing system namely the Sprint Tool System (STS), which includes the system requirements, test cases and test results. STS has been developed by undergraduate students over a full-semester period. Such study has the advantage of reducing some investigation cost due to availability of development elements.

5.1 The Sprint Tool System (STS) Description

STS is a student context project developed by 140 students. The project aims at developing a tool that facilitates the agile development activities following the SCRUM method. Scrum is a project management framework for developing complex products and systems. Scrum employs a lean iterative and incremental approach with empirical process control [20]. The tool provides means for developers to log the user stories in a product backlog from which a SCRUM sprint is chosen. Requirements could not be modified after documenting them in a sprint backlog. When a change occurs in a requirement, it is logged with the user stories and taken into consideration for the next sprint. The tool assumes each sprint lasts for 24 hours to 30 days. The tool also supports task assignment, meeting and project artifacts management.

5.2 Experiment Design

5.2.1 Hypothesis

We hypothesize that when prioritized regression test cases are derived from prioritized requirements represented in a semi-formal method, the derived test cases would provide better requirements coverage and higher rate of fault detection for the requirements with higher priority than the test cases developed using the requirement based classical test case generation technique.

Table 1 STS Prioritized Requirements

	Story Description	Priorit
R1	Guest can register on the system by entering his data, an email is sent to the	1
R2	Any registered user can request to create a project, admin could accept or decline user request. if accepted user is granted project	1
R3	Project owner can create, edit and delete task	0..8
R4	Project owner can invite registered users to a project, user can accept or decline the invitation, if he accepts the user is granted a project member role by default,	0.8
R5	A project member, can request to be a reviewer on a specific type of task, project owner can accept or decline request	0.2
R6	Project owner can create, edit and delete sprint in a project workspace.	0.8
R7	Project owner can create, edit and delete component in a project workspace	0.4
R8	Project owner can create, edit and delete meeting in a project workspace. sprint.	0.6
R9	Project owner can set which tasks will be assigned to which sprint. If the sprint is under implementation task cannot be	0.8
R10	Project owner/developer can assign a specific task to a component	0.4

5.2.2 Experimental Variables

The independent variable is the approach being applied for testing (RECAP and the requirements-based classic test case design technique).The dependent variables are the percentage of requirements coverage, the percentage of fault detection, and the unit of fault severity detected per unit of test-case cost.

5.2.3 Experiment Design

We developed a set of prioritized release test cases manually from the sprint backlog as the students did not develop release test cases for the tool. They only performed unit testing and system integration testing. The test cases were developed by a professional software tester with 5 years of experience in testing a wide variety of software systems in an international software house. Further, the professional tester received two-day training on the tool from the developers to become acquainted enough with the tool features prior to developing the manual test cases. We assigned priorities to the release test cases we developed manually based on the priorities given by the development team to each user story, as shown in Table 1.
We also constructed a DBT model representing the STS requirements from the given user stories. The DBT model was constructed by a professional tester who studied the GSE method. The DBT model and the prioritized set of requirements were given as input to RECAP to automatically produce prioritized release test cases.
We tested STS by running the two sets of release test cases, the one developed manually and the one derived using RECAP.

5.2.4 Evaluation Metrics

We employ a number of evaluation metrics to assess the effectiveness of RECAP against the classic technique as follows:

• Requirements Coverage (RC): This metric assesses the completeness of the test cases in covering the given requirements. We measure RC by the median count of test cases per requirement. The greater the median, the stronger the evidence is for the requirements coverage.

• Average Percentage of Fault Detection (APFD): This standard metric measures the average cumulative percentage of faults detected over the course of executing the test suite in a given order of its test case. APFD has been used to quantify and compare the rates of fault detection of test suites [19]. The higher the APFD, the better the rate of fault detection is for the specified order of test cases.

• Average Percentage of Fault Detection - Cognizant (APFDc): is a cost-cognizant test case prioritization measure. That helps prioritizing and evaluating test case orderings in a manner that considers the varying costs that often occur in testing real-world software systems. The higher APFDc, the better the rate of fault detection is for this order of test cases [19]. APFDc does not assume that the test cases and fault costs are uniform as does the APFD. We measure the cost of the test case by the time taken by the software tester to execute the test case on the system. We do not consider the test case development time in our cost calculation as regression test cases are developed once and run multiple times with every system build. Therefore, the test case development time is inconsiderable compared to the test case running time. Srikanth has proved in [14] that the severity of a fault is proportional to the test case priority in which this fault has been detected.

5.3 Results Analysis and Discussion

The results of the experiments are summarized in Table 2. RECAP produced 33 test cases versus 27 test cases developed manually using the classic technique. The number of test cases produced in any testing approach is not meaningful by itself as the uniform distribution of the test cases over the requirements is a more significant factor. The requirements traceability matrices of both the classic technique and RECAP show a uniform distribution of test cases over requirements.

Every requirement in both techniques has been tested by more than one test case while each test case involved at least one requirement. However, the number of test cases and their distribution over requirements in the classic technique rely heavily on the expertise of the test case developer as the process is purely manual.

Figure 5 depicts the requirements percentile versus the number of test cases per requirement using the requirements traceability matrices. At 50 requirement percentile, the median is 3 for RECAP and 2 for the classic technique. It means that half of the requirements were covered by at most 3 test cases in RECAP and 2 test cases in the classic technique.

Table 2 Summary of Experimental Results

	RC	APFD	APFDc
RECAP	33 Test Cases	54.5%	17.2%
Classic	27 Test Cases	33.3%	16.4%

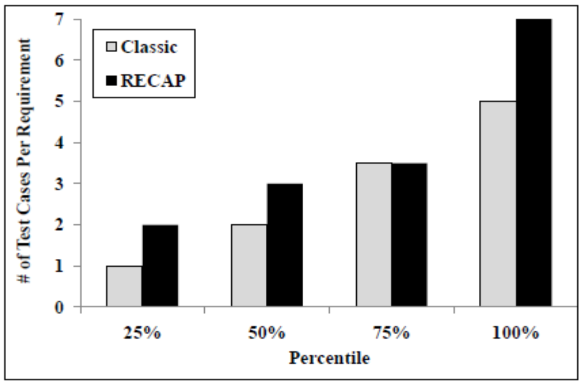

Figure 5 Coverage

The 100 requirement percentile shows that for all the requirements in RECAP, each requirement is covered by at most 7 test cases while it is 5 in the classic technique. The percentile results show a better coverage of requirements in RECAP over the classic technique. Based on these results, we expect the requirements coverage gap between RECAP and the classic technique to widen with larger project sizes.

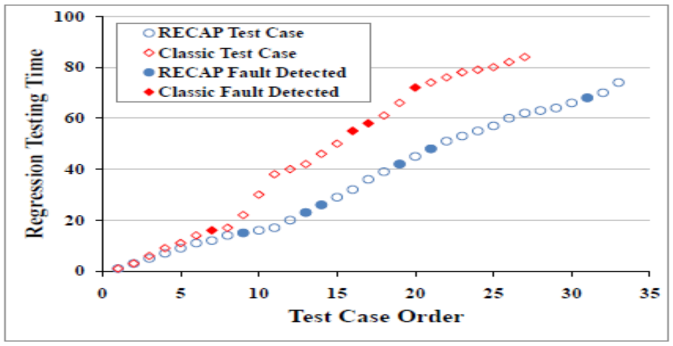

Figure 6 Test Cases and Fault Detection

RECAP test cases detected 6 faults while the classic technique test cases detected 4 faults as illustrated in Figure 6. The figure shows the cumulative time taken to execute the test cases when they are run in their priority order from highest to lowest. RECAP was not only capable of detecting more faults, but it detected such faults earlier than the classic technique. Though the number of test cases produced by RECAP is larger than those produced by the classic technique, the RECAP test suite is executed in less amount of time than the classic technique. Our justification to this result is that RECAP test cases are simpler requiring less amount of time to execute than the manually developed test cases of the classic technique. The higher APFD value of RECAP (54.5%) indicates that RECAP is more effective in fault detection compared to the classic technique (APFD = 33.3%). For APFDc, RECAP scores 17.2% whereas the classic technique scores 16.4%. This is an indication that RECAP detected more faults for requirements with higher priority. Detecting more faults in requirements with higher priority helps reducing the software testing time and raises the software quality [19]. The demonstration case studies and the validation results enabled us to scrutinize some of the strengths and limitations of RECAP.

Our analysis depicts two major strengths of RECAP. First, it reduces the testing time while maintaining a higher software quality than the classic testing techniques. The higher APFD and APFDc values of RECAP over the classic technique provides an evidence that RECAP is more capable of detecting faults in general and for higher priority requirements in particular. Detecting faults in higher priority requirements faster reduces the testing time and provides stronger evidence of the better software quality [19].

We expect RECAP to be even more effective in detecting faults than the classic technique for larger project sizes. As a result of deriving test cases from a semi-formal model rather than an informal requirements representation. Further, the automation process allows for reducing the testing time.
Second, RECAP increases the customers' satisfaction of the product features. Detecting faults in requirements with higher priorities allows the testing process to reveal the more significant faults that concern the customer first. Further, the better requirements coverage of RECAP along with automating the production process provides stronger evidences for customers to entrust the product quality, which increases their satisfaction.

Our analysis reveals two limitations of RECAP. First, RECAP is sensitive to any incompleteness or inconsistency in the GSE requirements model. Any such incompleteness or inconsistency would propagate to the test case derivation process in RECAP. However, RECAP has achieved a step forward by enhancing the requirement coverage compared to the classic technique as RECAP guarantees coverage of the requirements included in the GSE model. The depth first search algorithm used in RECAP to derive the test cases from the DBT model ensures coverage of all the DBT paths. Second, RECAP derives the regression test cases from GSE, which models functional requirements. Additional test cases are required to accommodate testing non-functional requirements such as the goal-oriented techniques or the Non-Functional Requirements (NFR) framework.

6. Conclusion

The proposed technique RECAP systematically derives regression test cases from a genetic requirements model, which ensures better regression testing completeness. Consequently, RECAP raises the customer confidence in the software verification process as it reduces its error-proneness. The generated regression test cases are prioritized according to the customer needs, which enables the reduction of the test suite size without reducing the quality of the tests. The proposed RECAP approach auto- mates the generation of the test case suite, and therefore, reduces human errors as well as the cost and effort of the regression test suite generation process. The regression test suite is derived using a systematic procedure, which allows for the flow of sufficient traceability information that relates test cases to requirements. Traceability information enables the construction of more structured traceability testing matrices. The RECAP approach is validated using credible case studies and an experimental software project. Our results illustrate the effectiveness of RECAP in providing more complete test coverage than the classic developing methods. Further, the results show that faults related to high priority requirements are detected more reliably than the classic testing techniques

References

[1] Sommerville Ian, *Software Engineering* (England, Addison Wesley, seventh edition 2004).
[2] F. Basanieri, A. Bertolino and E. Marchetti. The Cow Suite Approach to Planning and Deriving Test Suites in UML Projects. *Springer-Verlag Berlin Heidelberg 2002*

[3] S.G. Elbaum, A.G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Softw. Eng., 28(2), 2002.*
[4] S. Gnesi, D. Latella, and M. Massink. Formal Test Case Generation for UML State charts. *In ICECCS, 2004.*
[5] A. Hessel, K.G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Real-Time TestCase Generation Using Uppaal. *In FATES, 2003.*
[6] L. Mariani, S. Papagiannakis, and M. Pezz`e. Compatibility and Regression Testing of COTS-Component-Based Software. *In ICSE, 2007.*
[7] T. Myers and R.G. Dromey. From Requirements to Embedded Software - Formalizing the Key Steps. *In ASWEC, 2009.*
[8] M. Prasanna and K.R. Chandran. Automatic Test Case Generation for UML Object diagrams using Genetic Algorithm. *IJSCA, 1(1), 2009.*
[9] B. Qu, C. Nie, B. Xu, and X. Zhang. Test Case Prioritization for Black Box Testing. *In COMPSAC, 2007.*

[10] V. Rajappa, A. Biradar, and S.Panda. Efficient Software Test Case Generation Using Genetic Algorithm Based Graph Theory. *In ICETET, 2008*

[11] S. Rayadurgam and M.P.E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. *In ECBS, 2001..*
[12] P.G. Sapna and H. Mohanty. Prioritiza tion of Scenarios Based on UML Activi ty Diagrams. *In CICSYN, 2009.*
[13] S. Shlaer and S.J. Mellor. Object Life cycles Modeling the World in States. *Yourdon Press*.
[14] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. *In ISESE, 2005*
[15] P.R. Srivastava, K. Kumar, and G. Raghurama. Test Case Prioritization Based on Requireme nts and Risk Factors. *ACM SIGSOFT SEN, 33(4), 2008.*
[16] L.H. Tahat, A. Bader, B. Vaysburg, and B. Korel. Requirement-Based Automated Black-Box Test Generation. *In COMP- SAC, 2001.*
[17] J. Tretmans and A. Belinfante. Automatic Testing with Formal Methods. *Technical Report TR-CTIT-99-17, University of Twentie t h Centre for Telematics and Information Technology, December 99*
[18] Y.Y. Yu, S.P. Ng, and E.Y.K. Chan. Generating, Selecting and Prioritizing Test Cases from Specifications with Tool Support. *In QSIC, 2003.*
[19] Malishevsky G. Alex, Ruthruff R. Joseph, Rothermel Gregg and Elbaum Sebastian, Cost-cognizant Test Case Prioritization. *Technical Report, University of Nebraska 2006 VOL. 17, NO. 5, MAY 1991*
[20] Rising, L.; Janoff, N.S.; AG The SCRUM SoftwareDevelopment for Small Teams. *IEEE 2000*