A Model-Based Framework to Support Complexity Analysis Service for Regression Testing of Component-Based Software*

Chuanqi Tao[1], Jerry Gao[2], Bixin Li[3]
[1]*School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, Jiangsu, China*
[2]*School of Computer Engineering, San Jose State University, San Jose, CA, USA*
[3]*School of Computer Science and Engineering, Southeast University, Nanjing, Jiangsu, China*

***Abstract*—Today, software components have been widely used in software construction to reduce the cost of project and speed up software development cycle. During software maintenance, various software change approaches can be used to realize specific change requirements of software components. Different change approaches lead to diverse regression testing complexity. Such complexity is one of the key contributors to the cost and effectiveness of software maintenance. However, there is a lack of research work addressing regression testing complexity analysis service for software components. This paper proposes a framework to measure and analyze regression testing complexity based on a set of change and impact complexity models and metrics. The framework can provide services for complexity modeling, complexity factor classification, and regression testing complexity measurements. The initial study results indicate the proposed framework is feasible and effective in measuring the complexity of regression testing for component-based software.**

***Keywords*-testing service; component-based software regression testing; software maintenance; regression testing complexity**

I. INTRODUCTION

Today, component-based software engineering is a widely used approach in software construction. Many modern software and web service systems are constructed based on reusable components, such as third-party components and in-house built components. During software maintenance, when a component is updated or upgraded, it must be retested based on its updated requirements in its reuse contexts. In addition, it needs to be re-integrated into a component-based application system. This refers to regression testing, which is an important phase of software maintenance.

In software maintenance, given a change requirement, component developers and maintainers could adopt various change approaches to realize the update requirements. Different ways to change components could cause diverse impacts on both components and system, thereby result in different retesting complexities and regression costs. Hence, how to measure, evaluate, and predict these complexities and costs becomes a challenging issue for test managers and quality assurance engineers. Complexity measurement is an effective way to address the above issues, and it provides the useful reference information in cost prediction [1, 2]. Therefore, it is necessary to measure the complexity of regression testing. From the perspective of software maintenance, the complexity of regression testing includes two parts. The first refers to program maintenance complexity which relates to software changes and impacts. The other refers to software re-testing complexity which is related to test updates, re-test operations, and test suite refreshment.

The paper has two major technical contributions. The first is its proposed complexity analysis framework based on formal metrics and graph-based models. It is introduced to evaluate the complexity of regression testing of component-based software. And the second is the reported case study experiment, which indicates that the proposed approach is feasible to apply and practical to use. This paper is organized as follows. Section II discusses the complexity measurement framework. Section III reports the results of empirical study. The related work is provided in Section IV. Conclusion and future work are summarized in Section V.

II. COMPLEXITY ANALYSIS SERVICE FOR REGRESSION TESTING

The existing research indicates that complexity can be used to estimate the cost or effort required to design, code, test, and maintain software, as well as predict errors or faults that might be encountered during testing [1, 2]. In addition, complexity measurement provides a guideline and cost indicator for software maintenance. Therefore, regression testing analyzers and performers need a systematic framework service for software regression testing complexity analysis. The testing service type could cover the whole regression testing process, such as change management, change impact analysis, test suite refreshment, testing complexity factor classification, complexity measurements. This section attempts to describe the service provided for regression testing complexity analysis, and then introduces a measurement framework to measure and analyze corresponding complexity in software components.

A. *Classification of Complexity Factors for Regression Testing*

The entire regression testing in software maintenance involves software changes, change impact, and retest. Changes and the corresponding impacts are an indispensable part of software maintenance and evolution. Therefore, the complexity of regression testing could be categorized into change complexity, impact complexity, and retest complexity. Software changes in a component-based system occur at different levels. There are five primary change complexity factors at component level. Internal data and function changes are commonly-used internal component changes. API Function changes are important to component users when new version is released. Port changes involve the changed caller functions which invoke functions from callees. These four changes are adopted as the complexity factors of component changes. Since software changes involve adding, deleting, and changing, these three change types are considered the complexity facets of those

*Supported partially by the National Natural Science Foundation of China under Grant No.61402229 and No.61202003, and partially Supported by Doctoral Fund of Ministry of Education of China under Grant No.20113219120021, and partially by the Postdoctoral Fund of Jiangsu Province under Grant No.1401043B

*Correspondence to: jerry.gao@sjsu.edu

978-1-4799-8356-8/15 $31.00 © 2015 IEEE
DOI 10.1109/SOSE.2015.42

four factors above. For instance, we could have added, deleted, and changed API for a modified component. Correspondingly, component impacts also have four complexity factors, which are affected internal data, internal function, API, and port. Retest complexity relates to a number of factors, such as testing environment setting, test case execution, and test result checking. Without regard to these normal testing complexity factors, retesting complexity is primarily related to test suite refreshment. Thus, we consider it as the primary retest complexity contributors. In addition, different test suite refreshment could result in various complexity. For instance, adding test cases usually brings more complexity than reusing test cases. Therefore, at component level, test suite refreshment basically has four factors, which are added, changed, deleted, and reused component test cases.

At system level, due to component function and data changes, we need to pay attention to composition changes, message changes, and configuration changes. For example, adding a new component could change the composition relation in a system. Component function or data changes could cause message-communication relation changes between components. In addition, configuration relation could be also changed whenever configurable functions, environment or structures are changed. Therefore, composition change, message change, and configuration change can be considered to be the major complexity factors of system changes. Complexity factors of system impacts are those affected elements, such as affected composition and configuration. Similar to component level, system test suite refreshment includes added, changed, deleted, and reused system test cases.

B. A Measurement Framework to Support Regression Testing Complexity Analysis

In this section, we propose a measurement framework for regression testing complexity analysis. The framework consists of two parts: graph-based models and formal metrics.

*1) **A Graph-based Model**:* Regression testing complexity depends on diverse complexity factors for software change, impact, and test suite refreshment, thus it is important to choose an effective approach to support the complexity analysis. This paper presents such analytic models in a graphic format as a base to develop well-defined metrics that enable dynamic visualization, evaluation and comparison for regression testing complexity. Inspired from the well-known *McCall Quality Assurance Model* [3], we propose a regression testing complexity model named **Complexity Graphic Model** (CGM). It is a radar chart-based polygon, in which the link from the origin vertex to an extreme point represents the complexity value of one complexity factor. The area of various polygon indicates the total complexity value.

After determining the primary complexity factors, we can model those factors in the polygon, and calculate the complexity value using the proposed metrics. Based on the calculating algorithm for polygon area in [4], we can derive the formula to formally compute regression testing complexity (RTC) as follows.

$$RTC = \frac{1}{2}\sin\frac{2\pi}{n}\sum_{i=1}^{n} a_i * a_{i+1} \quad (1)$$

Where a_i represent various *complexity factors*. For instance, regarding component changes, the *complexity factors* refers to the changed API functions, changed internal functions, changed internal data, etc. Each complexity factor associates with one link in CGM, and *n* denotes the number of *complexity factors* involved.

The graphic-based models and metrics are specially used for multi-factors. Since the change, impact, and retesting complexity refers to a number of factors respectively, this model could provide a multi-factor complexity in a reasonable manner. Compared to single linear computation, this model could facilitate comparison and visualization of the diverse change and impact complexity of related changed components or system, as well as different regression testing approaches. We can definitely discover which regression testing approach is more effective in terms of complexity comparison in the graphic model. In addition, this model could also provide a guideline for the development of related regression testing complexity analysis tools.

*2) **Complexity Metrics**:* We discuss the complexity metrics for component change, impact, and test suite refreshment respectively. There are two steps to measure those complexities: a) complexity metrics for each change (impact or test suite refreshment) factor, and b) complexity metrics for total change (impact or test suite refreshment) complexity with multi-factors based on the graphic models.

Component Change Complexity

First we consider how to use those factors collected and analyzed to effectively compute the change complexity. Intuitively, the complexity of changes are proportional to the number of changes made to the components or system. In addition, the diverse change factors might result in different change complexity. For instance, API changes usually lead to more complexity than internal function changes do. Thereby API changes should be assigned higher weight than internal function changes.

Moreover, for each change factor, we need to consider the percentage of changes out of totality for corresponding component factor to determine the complexity. The percentage can be used as a balance value for complexity computation. Note that the totality means the number of items for corresponding component factor in new version. Hence, the contributors to the change complexity are the number of changes, the weight, and the percentage of changes. The Function Point (FP) Model [5] was developed originally for the effort estimation of a new software project in the 1970s. Researchers also expanded the method to address the effort estimation in software maintenance [6]. Motivated from the concept of FP, we compute the complexity value through multiplying the number of changes for each factor with weighting factor and the balanced percentage.

From our experience, *adding* code in component is more complicated than *changing* code. Meanwhile, *changing* code is usually more complicated than *deleting*. Thus, we assigned the corresponding weights in terms of their diverse contributions to the total complexity. Also, some changes like API change usually lead to higher change complexity than other changes, such as internal data and function. Thereby we give them greater weight value. Nevertheless, the assigned weight is subjective from our experience in the case studies, thus, researchers or engineers can adjust the weights in practice according to their experience. For any change factor c_i, the complexity metric (denoted as $Complex_{comp}(c_i)$) can be given below.

$$Complex_{comp}(c_i) = |F^a_{comp}(c_i)| * W^a_{comp}(c_i) * P^a_{comp}(c_i) +$$

Table I
A SAMPLE COMPLEXITY METRICS FOR CHANGE FACTORS

Change Factors	Change Type	Number of Changes (value)	Weight (value)	Percentage of Changes (value)	Complexity (value)
$Complex_{comp}(c_1)$	add	2	4	33.33%	2.667
	delete	1	1	16.67%	0.167
$Complex_{comp}(c_2)$	change	2	2	20%	0.800
$Complex_{comp}(c_3)$	add	2	5	22.2%	2.222

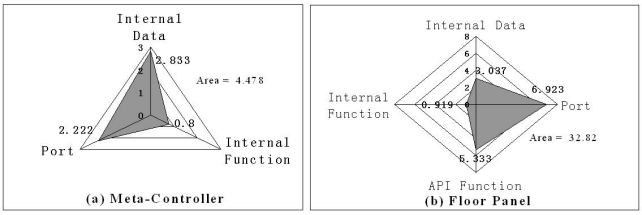

Figure 1. A Sample Change Complexity Models of Meta-Controller and Floor Panel

$$|F^d_{comp}(c_i)| * W^d_{comp}(c_i) * P^d_{comp}(c_i) + |F^c_{comp}(c_i)| * W^c_{comp}(c_i) * P^c_{comp}(c_i) \quad (2)$$

Where

• *a*, *d*, and *c* represents *added*, *deleted*, and *changed* respectively. $|F^a_{comp}(c_i)|$ denotes the number of added items. The right subscript *comp* for F, W, and P means changes made to a component. Similarly for deleted and changed items.

• $W^a_{comp}(c_i)$, $W^d_{comp}(c_i)$ and $W^c_{comp}(c_i)$ denotes the corresponding weight value.

• $P^a_{comp}(c_i)$, $P^d_{comp}(c_i)$, and $P^c_{comp}(c_i)$ represents the percentage of added, deleted, and changed items out of the total items respectively.

Each modified component could have various change factors. Thus, to measure change complexity with multi-factors, we propose a metric for regression testing change complexity (denoted as RTCC(comp)) based on the graphic model CGM. RTCC(comp) indicates change complexity of each modified component *comp*. Assuming there exists *n* change factors $c_i (1 \leqslant i \leqslant n)$ in component *comp*, the metric RTCC(comp) is given below.

$$RTCC(comp) = \frac{1}{2}\sin\frac{2\pi}{n}\sum_{i=1}^{n} Complex_{comp}(c_i) * Complex_{comp}(c_{i+1}) \quad (3)$$

Where $Complex_{comp}(c_i)$ denotes the complexity of any change factor. Note that $c_{n+1} = c_1$.

Figure 1(a) and 1(b) present the change complexity of two components respectively in a graphic model. In the system, the students changed two components *Meta-Controller* and *Floor Panel*. There are three change factors involved in *Meta-Controller*. They are internal data (c_1), internal function (c_2), and port (c_3).

The total change complexity is represented by the area of the triangle in dark color in Figure 1(a). The detailed change complexity computation consists of two steps shown as follows.

Step 1 *complexity metric for each change factor*

$Complex_{comp}(c_1) = (2 * 4 * 33.33\%) + (1 * 1 * 16.67\%) = 2.833$

$Complex_{comp}(c_2) = 2 * 2 * 20\% = 0.800$

$Complex_{comp}(c_3) = 2 * 5 * 22.22\% = 2.222$

Step 2 *total change complexity metrics*

$$\begin{aligned} &RTCC(metacontroller) \\ &= \frac{1}{2}\sin\frac{2\pi}{n}\sum_{i=1}^{n} Complex_{comp}(c_i) * Complex_{comp}(c_{i+1}) \\ &= \frac{1}{2} * (2.833 * 0.8 + 0.8 * 2.222 + 2.222 * 2.833) * sin(\frac{360}{3}) \\ &= 4.478 \end{aligned}$$

In Figure 1(b), four change factors are involved in component *floor panel*. They are internal data, internal function, port, and API. Thus, the change complexity is represented by the area of the quadrangle in dark color.

Component Impact complexity

Whenever any component is changed, its internal data and function, API interface, and port could be affected. Therefore, they must be considered the primary impact factors of complexity. We utilize a similar approach to computing impact complexity for diverse impact factors. Here we adopt three complexity contributors, i.e., the number of impact items, the weight, and the percentage of the impacts.

For any impact factor I_j, the complexity (denoted as $Complex_{comp}(I_j)$) is measured below.

$$Complex_{comp}(I_j) = |F_{comp}(I_j)| * W_{comp}(I_j) * P_{comp}(I_j) \quad (4)$$

Where

• $|F_{comp}(I_j)|$ represents the number of impact items.

• $W_{comp}(I_j)$ denotes the corresponding weight value.

• $P_{comp}(I_j)$ represents the percentage of the impact items out of totality.

Each affected component could have various impact factors. Thus, to measure impact complexity with multi-factors, we propose a metric for regression testing impact complexity (denoted as RTIC(comp)) based on the graphic model CGM. RTIC(comp) indicates the impact complexity of each affected component *comp*. Assuming there exists *n* impact factors involved due to components changes, the computation of RTIC(comp) is given below.

$$RTIC(comp) = \frac{1}{2}\sin\frac{2\pi}{n}\sum_{j=1}^{n} Complex_{comp}(I_j) * Complex_{comp}(I_{j+1}) \quad (5)$$

The Complexity of Component Test Suite Refreshment

As we mentioned above, the retesting factors primarily depends on added, deleted, reused, and changed test cases in the test suite. In [7], testing complexity is measured in terms of the number of test data required for demonstrating program correctness. We also take the number of test cases as an important complexity contributor. In previous work like [8], the authors made assumption that each test case has uniform cost, in order to simplify their problem to support cost model analysis. Here, to simplify the computation, we also assume that all the test cases in the suite are independent from each other, and each single test case has uniform complexity. Therefore, the retesting complexity is proportional to the number of test cases. In addition, different factors may contribute to complexities differently. For instance, according to our experience in case studies, adding test cases is usually more complicated than reusing test cases. Thus, we need to consider a weight parameter in complexity computation. The weight value is

usually determined by engineers subjectively based on their project experience. Besides, we need to consider the percentage of affected test cases out of the total size in the test suite. The weight values can be derived based on empirical data in the case studies.

Hence, for any factor T_k, its complexity for added, deleted, changed, and reused test cases (denoted as $Complex^a_c(T_k)$, $Complex^d_c(T_k)$, $Complex^c_c(T_k)$, $Complex^r_c(T_k)$ respectively) is measured below.

$$\begin{aligned} Complex^a_{comp}(T_k) &= |F^a_{comp}(T_k)| * W^a_{comp}(T_k) * P^a_{comp}(T_k) \\ Complex^d_{comp}(T_k) &= |F^d_{comp}(T_k)| * W^d_{comp}(T_k) * P^d_{comp}(T_k) \\ Complex^c_{comp}(T_k) &= |F^c_{comp}(T_k)| * W^c_{comp}(T_k) * P^c_{comp}(T_k) \\ Complex^r_{comp}(T_k) &= |F^r_{comp}(T_k)| * W^r_{comp}(T_k) * P^r_{comp}(T_k) \end{aligned} \quad (6)$$

Where $|F^a_{comp}(T_k)|$, $|F^d_{comp}(T_k)|$, $|F^c_{comp}(T_k)|$, and $|F^r_{comp}(T_k)|$ represents the number of test cases added, deleted, changed, and reused for factor T_k.

Intuitively, the complexity of test suite refreshment depends on all these factors. Thus, we propose a complexity metric (denoted as RTTC(*comp*)) with multi-factors based on our graph-based model. Note that no more than four factors could be involved in our test suite refreshment complexity metric computation. Therefore,

When three factors involved in *comp*, the computation is below.

$$RTTC(comp) = \frac{1}{2}\sin\frac{2\pi}{3}\sum_{k=1}^{3} Complex_{comp}(T_k) * Complex_{comp}(T_{k+1}) \quad (7)$$

At system level, the complexity metrics approach is similar to that at component level. The corresponding measurement formulas can also be expressed through the proposed graphic-based models and metrics. Thus, we do not discuss the measurement realization in details here.

III. Empirical Studies

A. Study Subjects and Objectives

We report our case study by applying the proposed complexity analysis service into a component-based elevator simulation system. We have used two software testing classes and three master project teams to conduct the controlled experiments in San Jose State University, California, USA. Now we investigate if the proposed measurement framework is feasible and potential to measure the regression testing complexity.

In the new version of the elevator system, some change requirements were made: (a) adding a 'floor indicator' function in the existing component to show the car id, car type and floor number, and (b) enhancing the protocol, to support the odd and even floor service. The students utilized the approach in our previous work [9] to conduct regression testing. Then they applied the proposed models and metrics in this paper to perform complexity computation and analysis. The students have two deliverables. The first is the updated design document, in which they record the component and system change and impact, and measure the change and impact complexity using the proposed metrics. The second is the retest document, in which they record the added, deleted, changed, and reused test cases, and measure the complexity of test suite refreshment. We have three groups of students participated in the case study. Each group made their own changes to the component or system in order to meet the change requirements. Next, we will report the results of the case study and provide some discussion.

Table II
Change Complexity Record (Group 1, 2 and 3)

Group No.	UserPanel	FloorPanel	Algorithm	Car	Car Controller	Admin Console	Meta controller	System
G1	0.78	5.82	-	3.10	0.46	6.23	4.48	10.00
G2	2.24	3.10	4.67	2.59	3.50	1.18	-	6.99
G3	1.36	67.03	1.98	1.86	-	3.11	1.67	8.93

Table III
Impact Complexity Record (Group 1, 2 and 3)

Group No.	User Panel	Floor Panel	Algorithm	Car	Car Controller	Admin Console	Meta controller	UserPanel Queue	System
G1	0.154	-	-	0.256	0.091	0.082	0.084	0.941	12
G2	0.712	0.286	-	0.042	1	1.653	1.333	-	1.5
G3	1.82	0.088	8.66	1.133	-	0.27	0.653	-	9.75

B. Study Results and Discussion

Using the metrics proposed in this paper, we measured the complexity of changes, impacts, and test suite refreshment for both components and system. Table II presents the results of change complexity value for each changed component and system. G1 has the greatest complexity value for component *car* (3.10), *adminconsole* (6.23), and *metacontroller* (4.48), while G2 gets the greatest complexity value for component *userpanel* (2.24) and *algorithm*(4.67). G3 has a distinct high complexity (67.03) from the other groups. This is because G3 accounted the change complexity of new component *floorindicator* in *floorpanel*. At system level, G1 has the greatest value of system complexity (10).

Table III shows the results of impact complexity value. At component level, G1 has the lowest impact complexity for most of the affected components, such as userpanel (0.154), adminconsole (0.082), and metacontroller (0.084). G2 has the highest complexity value for *floorpanel*, *carcontroller*, *admincosole*, *metacontroller*. In addition, G1 has the highest system impact complexity value (12) while G2 has the lowest (1.5). The results of test suite refreshment complexity is shown in Table IV.

C. Empirical Validation

We need to perform a validation to indicate the effectiveness of the proposed complexity analysis approach. Complexity is a primary maintenance and testing cost driver. Therefore, we investigate the effectiveness through an initial comparison between the

Table IV
Test Complexity Record (Group 1, 2 and 3)

Group	UserPanel	Floorpanel	Algorithm	System
G1	21.14	82.30	-	146.60
G2	15.03	19.05	53.10	122.80
G3	13.20	136.20	29.10	71.50

Table V
Complexity and Actual Effort Record

	CC	CE	IC	IE	TC	TE
G1-UserPanel	0.78	7	0.15	3	21.14	17
G1-FloorPanel	5.82	16	-	-	82.30	20
G1-Algorithm	-	-	-	-	-	-
G1-System	10	18	12	20	46.60	35
G2-UserPanel	2.24	22	0.17	2	15.03	12
G2-FloorPanel	3.10	25	0.29	3	19.05	15
G2-Algorithm	4.67	28	-	-	53.10	32
G2-System	6.99	37	1.50	6	122.80	46
G3-UserPanel	1.36	15	1.82	21	13.20	11
G3-FloorPanel	67.03	52	0.09	1	136.20	45
G3-Algorithm	1.98	17	8.66	21	29.10	31
G3-System	8.93	49	9.75	42	71.50	37

(Note: CC: component complexity; CE: component effort; IC: impact complexity; IE: impact effort; TC: test complexity; TE: test complexity.)

complexity and the actual effort.

In the empirical studies, we recorded the actual effort of the entire regression testing. The actual effort includes change, impact, and test refreshment cost. The three study groups reported their actual effort. Actual efforts were collected from the empirical study results and measured in person-hour. The change effort is measured by the time spent on change identification. We take the time spent on modifying the code after changes as the actual impact effort. Testing effort is measured in terms of test suite refreshment, which includes the time for test execution, test coverage analysis, and test result checking. Table V shows the sample for the record of the complexity value and actual effort in three modified components and the system.

From Table V, we discover that more complexity usually leads to more effort. The effort made to regression testing is generally proportional to the complexity. For instance, G3 has the greatest change complexity value for component *floorpanel* among those three groups, and it also has the biggest effort recorded. G1 makes the least change complexity in component *userpanel* and its effort is also the lowest. However, the testing effort seems not always proportional to the complexity. For example, Group 3 recorded more test complexity than Group 1 at system level. But those two groups reported the similar testing effort: 35 and 37. Due to the complex external testing environment, the precision of complexity value could be affected in practical use. Thus, we need to refine the testing complexity analysis and metrics in future work to enhance the precision of the proposed complexity model.

Our effort record and comparison indicates the complexity metrics is an important indicator for the regression testing cost. Through the effort comparison, we can initially conclude that the proposed complexity measure framework is effective to support complexity analysis and metrics. Note that cost analysis and indicator is not the research presented in this paper. We only try to verify the effectiveness of our approach through actual effort. To study the correlation between complexity and effort, further statistical analysis such as regression analysis could be performed. In addition, well-defined cost indicator can be developed to check the effectiveness of the complexity measurement approach.

D. *Observations and Lessons Learned*

This section summarizes our observations and lessons learned from the case study. Based on the same change requirement, those groups made diverse changes, which cause different impacts and test suite refreshment. This leads to the distinguish regression testing complexities.

Observation 1- For a changed component, its test suite refreshment complexity is not always proportional to change and impact complexity.

In Table II, the component change complexity value for component *algorithm* reported by G2 and G3 is 4.67 (27% of total change complexity) and 1.98 (3% of total change complexity). In Table III, the component impact complexity reported by G3 is 8.66 (69% of total impact complexity), but G2 does not report any impact complexity. Thus, G3 reports more total change and impact complexity than G2. Nevertheless, we find that the test suite refreshment complexity value reported by G2 (53.10) in component *algorithm* is greater than G3 (29.10) shown in Table IV. We analyze that they used different testing methods to generate new test cases. G3 adopted basis path-based testing, while G2 used branch-based testing. Via checking the internal code of *algorithm*, we find that the internal changes involve more branches. This also gives us the implication that different testing methods might result in different retesting complexities.

Observation 2- Some changes made to components might cause less impacts at component level, but cause more impacts on the entire system.

For example, a total of change complexity for components and system for G1 is 20.87 and a total of impact complexity is 1.61. Those values are 17.28 and 5.03 for G2. However, G1 reported more system change and impact complexity than G2. This is the case that component changes cause less complexity on components, but cause more on the system. The results also indicate that component changes not only bring impacts on components, but also affect the entire system.

Observation 3- Change factors play an important role in impact complexity at component level.

For instance, the total component change complexity for G1 and G2 is 20.87 and 17.28, but the corresponding impact complexity is 1.61 and 5.03 respectively. Thus G1 reported more component change complexity than G2, but reported less component impact complexity. Based on our analysis, G2 added and changed more *APIs* for different components, thereby results in high impact complexity. Hence, different change factors and types could cause various impacts.

Observation 4- At component level or system level, the complexity of test suite refreshment is proportional to the sum of change and impact complexity.

For example, at component level, the sum of change and impact complexity is 22.49, 22.31 and 89.63 for those three groups respectively. As a result, the complexity of test suite refreshment is 103.44, 87.18 and 178.5 respectively. At system level, the sum of change and impact complexity value from each group is 22, 8.49, and 18.68. The complexity of test suite refreshment is 146.4, 122.8, and 131.5 respectively. In summary, the complexity of test refreshment is proportional to both change and impact complexity.

From the case study, we conclude that various changes cause diverse impacts and test suite refreshment, thus leading to different regression testing complexity. For example Group 3 added a new component *floorindicator*, which results in a high change complexity. The component change complexity (77.01) is three times higher than Group 2 (17.28). Therefore, to reduce the regression testing complexity, we should avoid to create new components when it is not necessary. Overall, change factors play an significant role in change and impact complexity. Some factors cause more impacts, and some cause less. The specific changed components also affect impacts. That indicates different component change could cause various impacts. In general, core components or complex components changes could bring more impacts. Component changes can bring impacts at both component level and system level. For instance, API changes, message changes, and port changes affect system significantly. Test suite refreshment complexity is related to change and impact complexity. As we mentioned, the more changes and impacts brought to the system usually cause more test complexity at both component level and system level.

IV. Related Work

There are a variety of software complexity measurement, such as McCabe's Cyclomatic complexity, McClure's Control Flow Metric, function-oriented metrics, lines of code, etc [10]. Most of the existing research work focuses on complexity measurement in software maintenance, evolution as well as faults and effort prediction. For instance, Kemerer et al. summarized the application of complexity measurement in software maintenance [11] in early research. They attempted to explore the relationships among software complexity and various important aspects of software maintenance. They concluded that software maintenance complexity measurement correlates with changes, effort and errors. Hassan discussed complexity metrics that are based on the code change process, which includes fault repairing modifications, general maintenance modifications, and feature introduction modifications [2]. The empirical study shows the change complexity metrics are better predictors of fault potential than other predictors. Kafura and Reddy applied a total of seven code metrics and structure metrics to the experience of maintenance activities performed on a system with successive versions [12]. The study results indicate that these proposed metrics were able to identify the complexity of maintenance. Nikora and Munson proposed an approach to measuring software evolution [1]. They utilized several metrics such as the number of nodes and edges in control flow graph. They indicated that structural measurements of a systems structural evolution can serve as useful predictors of the number of faults inserted into a system during its development. They also found that various changes result in the different introduction of faults into the system.

So far, we have not discovered a research paper regarding complexity measurements of regression testing for component-based software. According to our survey, a related topic to complexity is cost-effectiveness. Several papers have addressed this issue. For instance, Leung and White et al proposed a well-known regression testing cost model [13]. Malishevsky et al. enhanced Leung and White's model through considering the cost of omission of faults and rate of fault detection [8]. Those cost models for regression testing above are primarily used for evaluation of various regression testing strategies. Researchers also proposed some prediction models for cost-effectiveness of regression testing. For example, Rosenblum et al. and Harrold et al. proposed a prediction for cost-effectiveness [14, 15].

V. Conclusions and Future Work

This paper has presented an approach to complexity measurement for regression testing of component-based software. We proposed a graphic model and several metrics for the complexity measurement, which consists of both maintenance and retesting complexity. A case study is conducted to compare the complexity of regression testing using the data from three independent groups. The case study results indicate the approach is feasible, and can visually compare the regression testing complexity. Currently, we are working on a cost model for regression testing of component-based software. The model will associate with the complexity analysis. We attempt to predict regression testing cost using the models through empirical studies. In addition, we are developing an automated tool for regression testing complexity and cost analysis.

Acknowledgement

We thank the students of SJSU's CMPE 287 course who participated in our study, and the support of Computer Engineering Department in San Jose State University of California.

References

[1] A. P. Nikora and J. C. Munson. An approach to the measurement of software evolution. *Journal of Sosftware Maintenance And evolution: Research and Practice*, 17(1):65–91, 2005.

[2] A. E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering*, pages 78–88, 2009.

[3] J. A. McCall. Factors in software quality. *Springfield, VA: National Techniacal Information Service*, 1-3, 1977.

[4] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3(2):153C174, 1990.

[5] A. J. Albrecht. Measuring application development productivity. In *Joint SHARE/GUIDE and IBM Application Development Symposium*, pages 83–92, 1979.

[6] Y. Ahn, J. Suh, S. Kim, and H. Kim. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance And Evolution: Research And Practice*, 15(2):71–85, 2003.

[7] K. C. Tai. Program testing complexity and test criteria. *IEEE Transactions On Software Engineering*, 6(6):531–538, 1980.

[8] A. G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*, pages 204–213, 2002.

[9] C. Q. Tao, B. X. Li, and J. Gao. A model-based approach to regression testing of component-based software. In *International Conference on Software Engineering and Knowledge Engineering*, pages 230–237, 2011.

[10] R. S. Pressman. Software engineering: A practitioner's approach. *Mc Graw Hill*, 2008.

[11] C. F. Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, 1995.

[12] D. Kafura and G. R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions On Software Engineering*, 13(3):335–343, 1987.

[13] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of International Conference on Software Maintenance*, pages 201–208, 1991.

[14] D. Rosenblum and E. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engieering*, 23(3):146–156, 1997.

[15] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engieering*, 27(3):248–262, 2001.