Test Adequacy Assessment Using Test-Defect Coverage Analytic Model

Sharifah Mashita Syed-Mohamad[1], Nur Hafizah Haron[1] and Tom McBride[2]
[1]*School of Computer Sciences, Universiti Sains Malaysia, 11800 Gelugor, Pulau Pinang, Malaysia.*
[2]*Faculty of Information Technology, University of Technology, Sydney, 15 Broadway, Ultimo NSW 2007.*
mashita@usm.my

***Abstract*—Software testing is an essential activity in software development process that has been widely used as a means of achieving software reliability and quality. The emergence of incremental development in its various forms required a different approach to determining the readiness of the software for release. This approach needs to determine how reliable the software is likely to be based on planned tests, not defect growth and decline as typically shown in reliability growth models. A combination of information from a number of sources into an easily understood dashboard is expected to provide both qualitative and quantitative analyses of test and defect coverage properties. Hence, Test-Defect Coverage Analytic Model (TDCAM) is proposed which combines test and defect coverage information presented in a dashboard to help deciding whether there are enough tests planned. A case study has been conducted to demonstrate the usage of the proposed model. The visual representations and results gained from the case study show the benefits of TDCAM in assisting practitioners making informed test adequacy-related decisions.**

***Index Terms*—Defect Coverage; Iterative and Incremental Development; Software Analytics; Software Testing.**

I. INTRODUCTION

Software testing is a process by which quality of the software under test can be identified. Information collected during testing such as defect based indicators is used to decide whether a piece of software is ready to be released. An early process was to freeze the software code to prevent further additions to the software functionality and then test it. Any defects revealed by the testing process were then fixed and retested. The rate of detection and fixing of outstanding defects and the overall decline in the number of outstanding defects with respect to time or testing effort indicates the level of reliability and this has led to various software reliability growth models [1-3]. However, these depended on being able to hold the code steady for a fixed period in order to observe the growing reliability. The emergence of incremental development in its various forms required a different approach to determining the readiness of the software for release [4]. This approach needs to predict how reliable the software is likely to be based on planned tests, not defect growth and decline.

Researchers in the field of software testing often focus on defining meaningful test coverage as measures which are frequently used interchangeably with the notion of test adequacy criteria. Test coverage is a measure of how well a test suite tests a program in order to gauge the effectiveness and completeness of testing [5-6]. Many studies suggest that higher test coverage correlates with higher probability of detecting more defects. However, it is often found that defects are not uniformly distributed across modules but typically clustered in large areas. It seems that a method that can furnish several information simultaneously is needed to help one determine the adequacy of tests. In light of this, this paper proposes an integrated model of test adequacy assessment in which results of testing are displayed graphically using bubble charts. And deciding quickly whether there are enough tests planned can be aided by presenting various factors or measure visually in some form of dashboard.

The rest of the paper is organized as follows: Section II describes the related works. Section III presents our proposed work. This is then followed by a case study to validate our proposed model presented in Section IV and Section V concludes the paper.

II. RELATED WORK

Software testing is an important indicator of software reliability and quality. The main problem of testing is to ascertain the adequacy of tests that is typically judged when enough coverage and effective tests are performed. There are two main approaches to deciding whether software is ready for release; test based indicators [7] and defect based indicators [8-10]. Defect based indicators such as reliability growth models and defect occurrence patterns in one circumstance predicting defect occurrence in other circumstances. Growth in reliability usually happens through freezing the functionality at a certain stage during system testing, and then fixing defects as they are detected. A growth in reliability indicates the readiness of the software to be released. However, this is observed during a stabilization phase in which a time period where the software does not have any more features added [4]. Software that is rapidly evolving undergoes continuous changes and modifications [11]. Rapidly evolving software projects (such as agile) might not have a definitive stabilization phase during which reliability can be examined. Therefore, there is a need to predict how reliable the software is likely to be based on planned tests, not defect growth and decline.

Test based indicators such as test coverage indicate the thoroughness of testing. Some common test coverage criteria include statement coverage, branch coverage, condition/decision coverage, and path coverage. Mala et al. [12] for example, used path, state and branch coverage to define software test adequacy. Although some studies argue that the effectiveness of defect detection does not correlate to test coverage [13-14], other studies have presented evidence that higher test coverage correlates with higher probability of detecting more defects. Malaiya et al. [15] showed that the

growth in the number of defects detected was almost linearly correlated with the growth in test coverage. Their finding supports existing view that 80% branch coverage is often adequate [16]. Mockus et al. [16] conducted case studies on two industrial projects and concluded that increase in test coverage is associated with decrease in post-released reported defects.

Cai & Lyu [17] employs test coverage and mutation analysis to investigate the relation between test coverage and defects detecting capability for different testing profiles. He found that test coverage contributes to noticeable amount of defect coverage. However in both cases it is possible that the indicators do not reflect reality or are impractical. Defect based indicators are limited by the extent to which the future mimics the past or, in the case of reliability growth measures, the amount of time available to establish a measure. Test coverage based measures do not, by themselves, draw attention to defect prone code. Several studies show that software defects are not randomly and uniformly distributed throughout the software under test. Rather defects tend to be found in clusters. A positive relationship between code complexity and defects has been shown by a number of studies [18-19]. These and, possibly other factors can be used to predict differing levels of test coverage for different parts of software under test. Therefore, we need a way to combine the two measures that will better indicate the true state of the software. More insights can be derived by combining different kinds of analysis and metrics, summarizing, filtering, modeling and experimenting the value of data and information [20].

A bubble chart can be used to visualize several factors simultaneously. Bubble chart is an extension of scatter chart where instead of having two dimensions, it has the third dimension (of the data point that can represent another set of data by its size). Bubble charts have been used extensively to represent and visualize data in various fields such as health, information technology and finance [21].

Deciding quickly whether there are enough tests planned can be aided by presenting various factors or measures visually in some form of dashboard. Most software analytics approaches focus on quantitative historical analysis, often using dashboards [20]. Dashboards will help managers to consistently monitor and measure certain aspect of the software project such as its quality, reliability, maintainability and complexity.

Clover is a commercial test coverage tool by Atlassian. Features that are supported by Clover includes test coverage by statement, branch, methods and complexity, identification of the riskiest code in the software under test, and calculation of other code metrics such as lines of codes (LOC) and class complexity. Clover uses tree map to represent the relation of code complexity of each component to the test coverage percentage. With this, modules that are not fairly covered will be highlighted in red to indicate to the user to focus on that particular component. Components with more complex code will be drawn using bigger rectangle. Hence, component with more complex code that are poorly tested will be visually clearer to the test managers as the size of the rectangle is bigger and it is colored in red. However, Clover is not free and seems to be for Java only.

eclEmma and SonarQube are examples of open source test coverage tools that use JaCoCo (an open source test coverage API) as its test coverage engine. It integrates test coverage analysis into Eclipse IDE. eclEmma provides features such as source code highlighting (according to coverage percentage) and coverage dashboard. eclEmma coverage dashboard lists out coverage summaries for the software under test, allowing drill-down to method level.

SonarQube employs several other open source tools such as GitHub for revision control and source code management, and FindBugs for bugs prediction. SonarQube covers seven axes of code quality: unit tests, test coverage, complexity of the source code, bugs prediction, coding rules, duplicate code, and complexity of the code. The dashboard is an aggregation of more detailed metrics of the software under test. Metrics such as lines of codes, number of files, test coverage results and technical debt can be analyzed by this tool. However, there seems to be lack of connection and reasoning between test based indicators and defect based indicators.

In summary, the emergence of incremental development in its various forms demands an applicable means to determining the readiness of software for release. We have to predict how many test needed during planning and cannot afford to wait then react to the number of defects one may find during testing. We have knowledge of defect occurrence in software that can predict what level of coverage is needed for different parts of the software. We do not have a readily available easily understood means to combine information from a number of sources into an easily understood dashboard that can help decision making about test coverage. Hence, there is a need to combine the two indicators that will better indicate the true state of the software.

III. Test-Defect Coverage Analytic Model (TDCAM)

Test-Defect Coverage Analytic model (TDCAM) is proposed that integrates test coverage and defect coverage information into an easily understood dashboard to assist decision making about test adequacy. Figure 1 depicts the overview of our proposed model in the form of input, process and output. The processes involved are data extraction, data parsing and filtering, test analytics module and analytics dashboard.

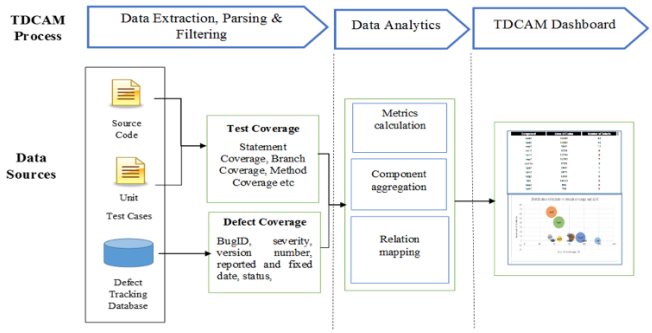

Figure 1: An overview of TDCAM model

The data acquisition/extraction process involves extracting both test coverage and defect coverage datasets from test coverage and defect tracking tools. The test coverage indicator provides related test-based metrics such code coverage. The defect coverage indicator provides defect related metrics such as types and severity of defects. The data parsing and filtering process involves filtering and removing redundant and trivial data of our concerns, and selecting data of our concern from the datasets extracted in the previous

process. In this process, data extracted from both datasets will be aggregated into component level.

Three main steps involved in Data Analytics stage; metrics calculation, data aggregation and data relation evaluation. The first step involves calculating relevant metrics derived from the two coverages. The metrics are aggregated to source code components (units of implementation). Examples of the metrics are shown in Table 1.

Table 1
Suite of Test Adequacy Metrics calculated by data analytics module

Name	Metrics
DAR	Defect Arrival Rate
DFR	Defect Fixed Rate
NDC	Total number of defects per component
NUDC	Number of unresolved defects per component
BCC	Branch coverage percentage per Component
CCC	Code Complexity

In order to support managers in making informed test adequacy decision, we proposed both qualitative and quantitative representations for the analytical view which is in a dashboard form. The quantitative analytics approach highlights high-level data trends thru statistical summaries of various metrics. On the other hand, the qualitative analytics approach emphasizes on the attributes and relationship of a set of software artifacts and metrics of interest. As discussed earlier, the bubble chart type with three dimensions of data is chosen to represent the hybrid indicators of test adequacy criteria. Figure 2 shows an example of the adopted bubble chart in terms of the number of defects, branch coverage measurement and code complexity.

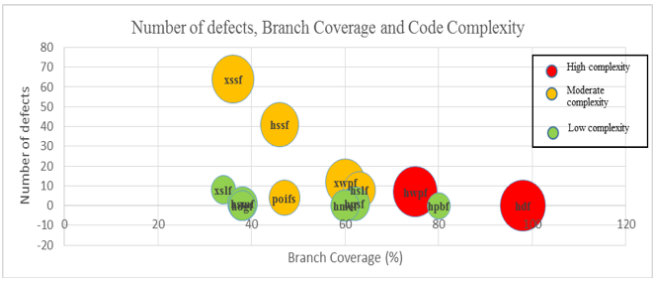

Figure 2: Test-defect coverage information represented in a bubble chart form

The y-axis represents the number of defects and x-axis represents the branch coverage percentage. Another dimension of data is depicted by the bubble size. The bubble size represents a few options of defect-based or test-based metrics such as the severity of defects or other useful related metrics such as lines of code and complexity, as shown in Figure 2. One can reasonably raise his or her concern on the test adequacy of those components, or decide to put less emphasis on components with low number of defects and high branch coverage. To further indicate how complex the code of each component is can be illustrated by the bubble's size and color. The bigger bubble size indicates a higher complexity of a component. The bubble can be color-coded with different colors according to its complexity such as red for high complexity, orange for moderate and green for low complexity code. This highlights which component is the hot spot for test inadequacy.

A. Implementation

This paper serves as a proof of concept meant to demonstrate how the model can be implemented. Apache POI project was selected in which both defect and test coverage data sets were extracted from open source repositories. Defect datasets were retrieved from Bugzilla defect tracking system. The extracted data are saved in xml (eXtensible Markup Language) file format. Test coverage measurement datasets were generated by running JUnit test cases provided in the source code. JaCoCo test coverage library and API are used to execute the JUnit test. JaCoCo provides a lightweight and flexible library for integration with various build and development tools. The API also supports complexity of the code measurement as defined by McCabe. Results of the test coverage are generated in csv (Comma-Separated Value) file format.

In data filtering and analysis process, the Data Filter Module filters and removes unwanted raw data from Data Acquisition Module. This module employs XMLParser and CSVParser API integrated into the eclipse IDE to extract and filter needed data from the raw data extracted. Next is the execution of the calculation module to calculate and analyze metrics. The last process is the visualization of the results in a dashboard.

Eclipse Birt is adopted and integrated into Eclipse IDE as a reporting tool. It has the capability of reporting and charting features much like Microsoft Excel but with capabilities to be extended with user-defined analysis algorithm through Java Objects and Classes. Design Report, Chart and Engine Report API are provided by Eclipse Birt in order to support this functionality.

Software test adequacy metrics calculated by the Data Analytics Module are converted to Java objects through classes and variables. These Java objects are the input for the Eclipse Birt. The TDCAM tool has been implemented as an Eclipse plugin. Figures 3 and Figure 4 depict the overall implementation process mapped against open source tools employed in our model, and the TDCAM drop-down menu for users, respectively.

Users can choose to either extract or display Test Coverage data in TDCAM Test Coverage view, extract and display Defect Coverage data in TDCAM Defect Coverage view, run data analysis or lastly to view data analysis result through a dashboard.

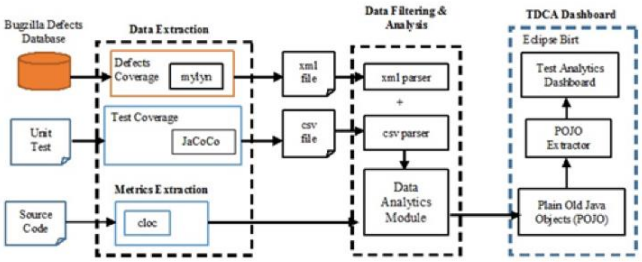

Figure 3: Implementation overview of TDCAM

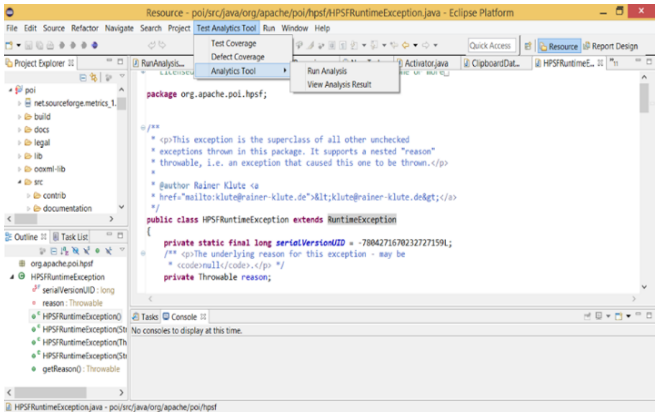

Figure 4: Drop-down menu for TDCAM tool

Mylyn connector was used to connect Eclipse IDE and the defects tracking database. Mylyn is a task-focused interface that can extends Eclipse capabilities to keep track of users' tasks. A task is defined as any unit of work that users want to retrieve or share with others, such as a bug reported by a user. These tasks can be stored locally or in a task repository (such as Bugzilla). Defects dataset extracted from the database are saved into a text-based file format. TDCAM tool further processes this extracted data and displays it in the Defect Coverage view of TDCAM tool. The tool extracts relevant attributes of each defect from the database such as BugID, product name, component name, status, date reported, date fixed, version number, severity and priority of the defects. Results of the analysis are presented in a dashboard view, as shown in Figure 5. TDCAM dashboard can be used to visualize both quantitative and qualitative analyses.

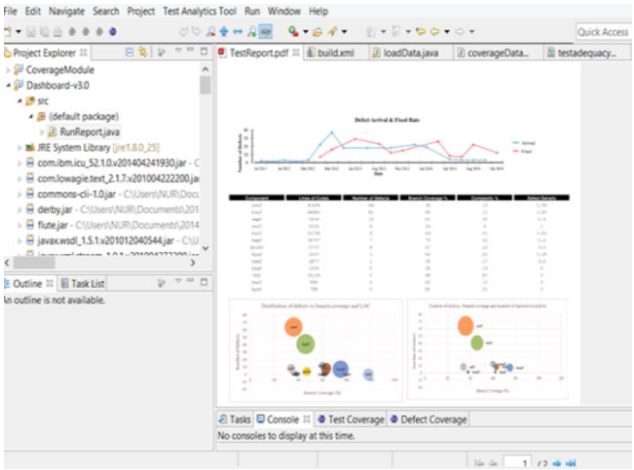

Figure 5: TDCAM dashboard

IV. Case Study

As mentioned earlier, Apache POI, an open source software was selected to demonstrate the proposed work. Apache POI is used to create and maintain Java APIs for manipulating various Microsoft file formats such as Microsoft Words, PowerPoint and Excel. It is chosen as our case study based on the following criteria:

i. A number of literatures adopted Apache POI as their case study. For example, Marian [22] proposed defect prediction model and Inozemtseva & Holmes [13] studied the relationship between test coverage, test suite size and test effectiveness.

ii. Apache POI is actively being developed and supported. Its first version was released in August 2003 and its latest release, version 3.11-beta3 was on Nov 2014.

iii. 7 active developers are currently working on this project. Number of defects reported and fixes can also be an indicator of the collective effort of the team in developing and maintaining the source codes. 179 defects are reported for Apache POI version 3.9 (which is the latest stable version at the time of our implementation).

iv. Sufficient data for valid empirical analysis. Apache POI source code is of reasonable size (>100,000 LOC). It also comes with extensive unit test. Its defects database can also be publicly accessed via Bugzilla. This allows us to run test coverage and extract defect information from the database.

A. *Data Analysis and Results*

We extracted two artifacts which are software defects data from the defects tracking system and test coverage measurement results. At the time of our implementation, version 3.9 is the latest stable release version for Apache POI. Thus, this version was chosen as our basis for the data analysis. We further filtered the defects by its resolution status of "DUPLICATE", "WORKSFORME", "INVALID" and "ENHANCEMENT". Total number of defects remains after filtering the defects is 144. We then categorized the software defects according to its respective software component name. Table 2 provides the information of test and defect coverage metrics used in this study.

Based on the table, there are 4 components that have no reported defects; hpbf, hmef, hdf and hdgf. It is found that these components are legacy code from the earlier version of the software under reviewed. Based on our observation of the project's change requests, the components have stabilized over time.

Table 2
Apache Poi's Test Adequacy Metrics for each component

Components	Code lines	Defect numbers	Defect Density	Branch Cov (%)	Code Complexity
poifs	8703	4	0.5	47	18
hpsf	4500	1	0.25	62	20
hdgf	1209	0	0	38	7
hdf	15119	0	0	98	11
xslf	8224	8	1	34	22
xwpf	5809	12	2.4	60	17
hssf	44880	41	0.93	46	10
hwpf	34757	7	0.2	75	12
hslf	13736	8	0.61	63	10
hsmf	2877	1	0.5	38	6
hmef	996	0	0	60	9
xssf	41636	64	1.56	36	8
hpbf	788	0	0	80	23

In this paper, we plotted 3 bubble charts to visualize several defect-based and test-based metrics simultaneously, as shown in Figures 6, Figure 7 and Figure 8. Based on Figure 6, two components that should be of concern to developers or testers are xssf and hssf components, as the number of defects is relatively high compared to other components. The percentages of branch coverage for those two components are also less than 60% which should alarm them on the thoroughness of tests. The bubble chart also shows a distribution of test-defect metrics against lines of code (LOC) for each component. LOC has been shown to have correlation

with other quality metrics such as number of defects and defect density. Many defect-free components tend to be small in size [23] and our result appears to be consistent with the existing findings.

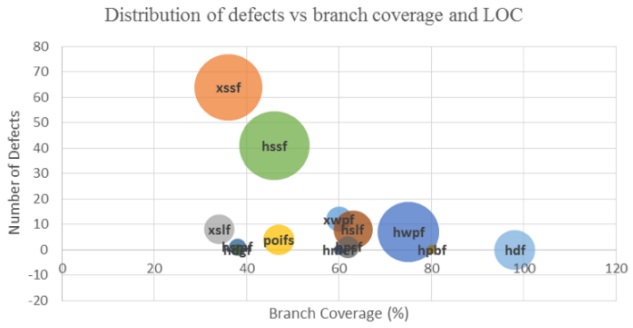

Figure 6: Combination of test-defect coverage and lines of code metrics

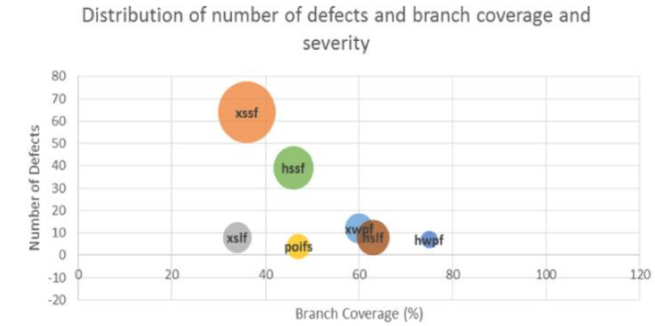

Figure 7: Test-defect coverage metrics mapped against severity of defects

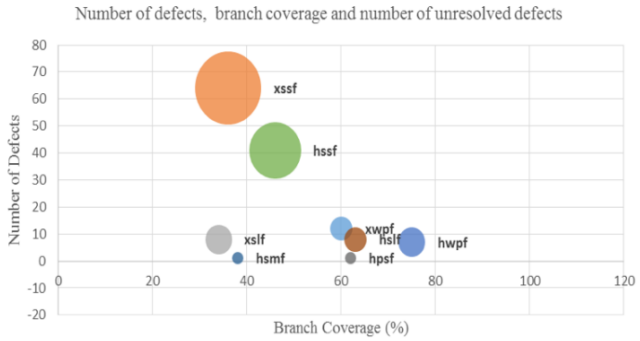

Figure 8: Test-defect coverage metrics mapped against number of unresolved defects

Figure 7 displays the information of test-defect coverage metrics mapped against severity of defects, in which the severity information is represented by the size of the bubble. We determined the bubble size by calculating the ratio of high severity defects for the component over the total number of high severity defects for all components. Thus, the bubble with the biggest size represents the component with the most number of high severity defects compared to other components. The hwpf component has relatively large lines of codes, yet small number of high severity defects. One of the possible reasons to this exception is that this component is thoroughly tested as indicated by the branch coverage percentage. The branch coverage measurement for this component is high (around 80%) compared to other components with large lines of codes (refer to component xssf and hssf). This finding is supported by the works of Mockus et al. [16] and Cai & Lyu [17] in which higher test coverage relates to lower number of reported software defects.

Another criterion to determine if the software has been adequately tested is by observing the number of remaining defects (as shown in Figure 8). Due to certain limitation such as time and cost, testers may decide that it is not necessary to resolve all reported defects. Apart from highlighting which component has the high number of defects but low branch coverage, the chart also visualizes which component with the most number of unresolved defects thru the bubble size. It also shows certain components that are more prone to defects. Hence, more resources and test should be focusing on these defect-prone components.

B. Survey

As part of the evaluation, a short survey was developed to get feedbacks pertaining to the usefulness of the model. Five participants consist of four developers/testers and one manager were asked to rate three questions by using five-point Likert scale (5 – Strongly agree, 4 – Agree, 3 – Normal, 2 – Disagree and 1 – Strongly disagree). All of the participants are actively being involved in development, testing and maintaining various software projects at the Centre for Knowledge, Communication and Technology and School of Computer Sciences, Universiti Sains Malaysia (USM).

We first conducted a demo session on how to use the tool and then distributed the questionnaire to each participant after the demo session. Participants were asked to try the tool and answer the given questions. The first question concerns with the usefulness of the tool in providing insight into software under test based on test-defect coverage metrics. The result shows 60% of the participants strongly agree that the bubble chart is able to provide meaningful information about the status of the software under test.

The participants were also asked to rate how easy for them to use the tool. 80% of the participants agree that the tool is easy to use while another 20% of the participants strongly agree that the tool is indeed easy to use. The third question concerns with whether the tool can be used to assist them in making informed test adequacy-related decisions. 60% of the participants strongly agree that the tool can help them in making such decision, while the rest agree.

C. Discussion

1) Threats to validity.

This section concerns the possible construct and external validity threats related to this study. The theoretical construct is software test adequacy. The main variable we used to assess software test adequacy is based on the number of software defects. This is where the test adequacy and to be exact, reliability of a software system is indicated by the absence of failure, which in turn, is indicated by the presence of defects. All measures used are justified and defined in this paper to remove the risk to construct validity.

Our sampling includes five practitioners from two software development centers. The small number of participants can potentially introduce biases in the survey results. However, the analysis on their responses suggests a similarly positive response.

2) Lessons learned

Based on our case study results, it is observed that a component with low branch coverage percentage tends to have high number of detected defects. In addition, if the

component has low branch coverage and low number of defects detected, the component either has small LOC or low code complexity. It is therefore, advisable for test managers to focus on the large components that are high in code complexity.

Although, five responses are not enough to form any sort of conclusion, but one can reasonably gain the benefits of TDCAM in assisting practitioners making informed test adequacy-related decisions, in particular for on-going projects that are actively being developed. One can continually assess the current state of the software on day-to-day or week-to-week basis as needed. For instance, in agile development projects, where the project cycle is faster and shorter, monitoring the two main metrics of software test adequacy can immediately alert managers on components that are inadequately tested in any iteration.

V. CONCLUSION

The emergence of incremental development in its various forms required a different approach to determining the readiness of software for release. Test-Defect Coverage Analytic Model (TDCAM) has been proposed to predict how reliable the software is likely to be based on planned tests, not defect growth and decline as typically shown in reliability growth models.

TDCM combines two important metrics of test adequacy criteria; test and defect coverage, to better indicate the true state of the software. A bubble chart presented in a form of dashboard is used to visualize several metrics on software defects and test coverage simultaneously. The purpose is to guide decision makers in testing to take informed decisions on test readiness. One of the findings shows that a component with high branch coverage and high number of defects detected indicates that it either has a large number of LOC or high in code complexity. So, practitioners can make better-informed decisions about their tests.

It has been our aim to extend TDCAM to become a suite of business analytics tool for software practitioners by incorporating the elements of artificial intelligence where a human's intervention can be minimized.

ACKNOWLEDGMENT

This research is partly funded by the Short-Term Grant (304/PKOMP/6312090) of the Universiti Sains Malaysia and by Exploratory Research Grant Scheme (203/PKOMP/673140) from the Ministry of Education of Malaysia.

REFERENCES

[1] S. Yamada, J. Hishitasni, and S. Osaki, "Software-reliability growth with a Weibull test-effort: a model and application," *IEEE Transactions on Reliability*, vol. 42, no. 1, pp. 100-106, 1993.

[2] A. Wood, "Software reliability growth models: assumptions vs. reality," in *Proceedings of the Eighth International Symposium On Software Reliability Engineering*, 1997, pp. 136-141.

[3] M. R. Lyu, "Software reliability engineering: a roadmap," in *2007 Future of Software Engineering IEEE Computer Society*, 2007, pp. 153-170.

[4] S. M. Syed-Mohamad, *An Empirical Investigation of Software Reliability Indicators*. University of Technology, Sydney, 2012.

[5] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366-427, 1997.

[6] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini, "Evaluating testing methods by delivered reliability," *IEEE Trans. On Soft. Eng.*, vol. 24, no. 8, pp. 586-601, 1998.

[7] L. Jihyun, S. Kang and D. Lee, "Survey on software testing practices," *IET Soft.*, vol. 6, no. 3, pp. 275-282, 2012.

[8] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. on Soft. Eng.* vol. 26, no. 7, pp. 653–661, 2000.

[9] N. Fenton, and O. Niclas, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans on Soft. Eng.*, vol. 26, no. 8, pp. 797-814, 2000.

[10] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th IEEE International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 284 – 292.

[11] S. M. Syed-Mohamad, and T. McBride, "Open source, agile and reliability measures," in *Proceedings of the 12th International Conference on Quality Engineering in Software Technology* (CONQUEST), Nuremberg, Germany: International Software Quality Institute. 2009, pp. 103-118.

[12] J. D. Mala, V. Mohan and M. Kamalapriya, "Automated software test optimisation framework-an artificial bee colony optimisation-based approach," *IET Software*, vol. 4, no. 5, pp. 334-348, 2010.

[13] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 435-445.

[14] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 147-157.

[15] Y. K. Malaiya, M. N. Li, and J. M. Bieman, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 420-426, 2002.

[16] A. Mockus, N. Nagappan and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *Proceedings of the* 3rd *IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM 2009), 2009, pp. 291-301.

[17] X. Cai, and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," in *Proceedings of the Workshop on Advances in Model-Based Software Testing* (A-MOST), St. Louis, Missouri, 2005, pp. 1-7.

[18] E. J. Weyuker, "An empirical study of the complexity of data flow testing," in *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, 1988, pp. 188-195.

[19] T. M. Khoshgoftaar, and J. C. Munson, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 253-261, 1990.

[20] O. Baysal, R. Holmes, and M. W. Godfrey, "Developer dashboards: the need for qualitative analytics," *IEEE Software*, vol. 30, no. 4, pp. 46-52, 2013.

[21] S. Halliday, B. Karin and V. S. Rossouw, "A business approach to effective information technology risk analysis and management," *Information Management & Computer Security*, vol. 4, no. 1, pp. 19-31, 1996.

[22] J. Marian, "Significance of different software metrics in defect prediction," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 86–95, 2012.

[23] A. Okutan, and O. T. Yildiz, "Software defect prediction using Bayesian networks," *Empirical Software Engineering*. vol. 19, no. 1, pp. 154-181, 2014.