A Complexity Reliability Model

Norm Schneidewind[1] and Mike Hinchey[2]
[1]*Naval Postgraduate School, Monterey, CA, USA*
[2]*Lero—the Irish Software Engineering Research Centre, University of Limerick, Ireland*
ieeelife@yahoo.com, mike.hinchey@lero.ie

Abstract

A model of software complexity and reliability is developed. It uses an evolutionary process to transition from one software system to the next, while complexity metrics are used to predict the reliability for each system. Our approach is experimental, using data pertinent to the NASA satellite systems application environment. We do not use sophisticated mathematical models that may have little relevance for the application environment. Rather, we tailor our approach to the software characteristics of the software to yield important defect-related predictors of quality. Systems are tested until the software passes defect presence criteria and is released. Testing criteria are based on defect count, defect density, and testing efficiency predictions exceeding specified thresholds. In addition, another type of testing efficiency—a directed graph representing the complexity of the software and defects embedded in the code—is used to evaluate the efficiency of defect detection in NASA satellite system software. Complexity metrics were found to be good predictors of defects and testing efficiency in this evolutionary process.

1. Introduction

Software development can be thought of as the evolution of abstract requirements into a concrete software system. Development, achieved through a successive series of transformations, is inherently an evolutionary process. Software evolution is often sub-optimal, because requisite information, such as reliability and complexity, may be missing during the transformations. While some understanding of software may be reasonably clear at a given time, future dependencies may not be fully understood or accessible. The clarifications obtained over time make the system more concretely understood, but there may be loss of relevant information. Some information may be lost due to failure to be fully acquainted with dependencies among various software artifacts [5].

As pointed out by [11], as systems change through successive builds, the complexity characteristics of the individual modules that make up the system also change. Changes to software systems are measured on these attribute domains to provide prime indicators of potential problems introduced by the changes. Establishing a measurement baseline permits the comparison of a sequence of successive software builds. We adopt this concept by using the first system in a series of software systems as the baseline for relating complexity to reliability, and from this baseline, evolve successive releases with decreasing complexity and increasing reliability. Complexity decreases and reliability increases across releases because even though there may be added functionality in subsequent releases, this is counterbalanced by evolutionary improvements in software development concurrent with increasing fault removal.

As Lehman points out [9], it is beneficial to determine the number of distinct additions and changes to systems and constituent modules of the system per release in order to assess system volatility. This can assist evolution release planning in a number of ways, for example by pointing to system areas that are ripe for restructuring because of high defect rates. Some authors have suggested that if the IT industry used standardized and interchangeable software components, the problem of unreliable systems would largely disappear [4]. Unfortunately, for one-of-a-kind space systems that are addressed for solving unique research problems, the COTS solution will not work. In space systems, reliability and complexity across systems will show considerable variation. Therefore, our analysis cannot be limited to a single system; it must address multiple systems, as we describe later.

Some researchers have found a Pareto effect when analyzing software defects: a small number of modules account for a large number of defects [1]. We did not

978-0-7695-3878-5 2009

U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/ISSRE.2009.10

IEEE computer society

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

find this characteristic in the NASA JM1 satellite system software we examined. This software has no pattern of defects across modules. Thus, we cannot focus on a few modules to detect and remove defects. Instead, the approach to be described involves conducting experiments to see whether a mapping can be developed between changes in software complexity and changes in reliability. If this mapping can be achieved with the desired degree of statistical significance, the approach would be judged a success [13].

This research builds on previous work in software reliability process [14]. However, we now focus on *product* reliability predictors in contrast to previous work that emphasized the *process* of prediction.

2. Complexity Reliability Model

Software complexity is the degree to which software is difficult to analyze, understand, or explain [2]. If software lacks structure, it will be difficult to understand, test, and operate, and therefore, has a direct bearing on reliability. In order to identify beneficial software evolutionary steps, as they relate to reliability and software complexity, we develop our complexity reliability model with the aim of reducing complexity and thereby increasing reliability. Software reliability of a program is hypothesized as a multivariate function of a complexity scalar at test or operational time *t.*

Why study complexity in relation to reliability? The answer is that complexity breeds bugs. The more complex the software, the harder it is to make it reliable [16]. Thus, building statistical models for estimating failure-proneness of systems can help software organizations make early decisions on the quality of their systems. Such early estimates can be used to help inform decisions on testing, refactoring, code inspections, design rework, etc. This has been demonstrated by the efficacy of building failure-proneness models, based on code complexity metrics, across the Microsoft Windows operating system [2]. The ability of such models to estimate failure-proneness and provide feedback on complexity metrics helps guide the evolution of the software to higher-and- higher plateaus of reliability.

Some software projects, such as the NASA JM1 satellite system, have not recorded failure data, but have recorded defect count as a function of time. Thus, we model the defect detection process by letting $D(t)$ be defect count at time t, in equation (0.1):

$$D(t) = f(C, t)), \quad (0.1)$$

where C is a scalar of metrics $M = m_1, \ldots, m_i, \ldots, m_n$, and m_i = cyclomatic complexity (CC) and edge count (EC) of the directed graph of the program. The defect and metrics data were obtained from software modules in several JM1 satellite systems.

Based on some researchers' results, one may question why we use cyclomatic complexity. According to [6], their research validates previously raised concerns about the metric, using a new data set. A simple transformation of the metric was investigated whereby cyclomatic complexity divided by the size of the system in source statements produced a complexity–density ratio. This ratio was demonstrated to be a useful predictor of software maintenance productivity on a small sample of maintenance projects. While this may be true for maintenance productivity on a small sample, we will show that for three safety critical software systems, comprised of 88 modules, cyclomatic complexity, along with edge count, provides accurate predictions of reliability. Furthermore, we do not think it wise to compute a complexity density ratio because cyclomatic complexity may be highly correlated with number of source statements (e.g., 0.8630 for the NASA data).

McCabe's cyclomatic complexity measure [10] provides an indication of how difficult it is to understand and test a program by calculating the number of linearly independent paths through a program. The measure also represents the total number of decision points in a program plus one. McCabe proposed that modules with a value exceeding 10 may be problematic by being difficult to understand [7]. Therefore, in Figure 1 we show that cyclomatic complexity, with mean value much larger than 10, and large mean edge count, provide important independent variables in predicting defect count. The practical effect of this result is that a reliability engineer could develop predictor equations for other software and data, using this approach, to predict defect count.

We also tried a prediction function that included source lines of code (sloc) count but doing this produced a Mean Relative Error, MRE = 0.2700 compared to MRE = .2631 when sloc was not included. The reason for this result is that sloc has high correlation of 0.8630 and 0.9352 with CC and EC, respectively. The reader might wonder why both cyclomatic complexity and edge count are included in the regression equation when cyclomatic complexity is a function of edge count in the directed graph of a program. We wondered about this too and developed a regression equation of defect count solely as a function of edge count. Still, the MRE = 0.2765 of this equation

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

is larger than the MRE = .2631, when both cyclomatic complexity and edge count are included. The reason is that defect count is dependent on cyclomatic complexity. Therefore, cyclomatic complexity must be included when predicting defect count. Finally, we investigated node count as the sole predictor of defect count. Since node count, *NC*, is not available in the data, we computed it in equation (0.2) based on the definition of cyclomatic dependency:

$$NC = (EC - CC) + 1 \quad (0.2)$$

This approach did not work because using an equation with just *NC* as the independent variable produced the highest error (MRE = 0.3744) of any of the regression equations. Thus, we settled on the regression equation that included cyclomatic complexity and edge count.

3. Product Line Concept Applied to Complexity Reliability Model

When dealing with complex systems, and in particular systems exhibiting any form of autonomy or autonomic properties, it is unrealistic to assume that the system will be static. Complex systems evolve over time, and the architecture of an evolving system will change even at run time, as the system implements self-configuration, self-adaptation, and meets the challenges of its environment.

An evolving system can be viewed as multiple versions of the same system. That is, as the system evolves it represents multiple instances of the same system, each with its own variations and specific changes. That is to say, an evolving system may be viewed as a product line of systems, where the core architecture of the product line is fixed (i.e., the substantial part of the system that does not change), and each version of the evolving system may be viewed as a particular product from the product line [12]. This concept can be applied to our complexity reliability model by considering the core system, as the first in a series of systems, comprised of a set of modules, each system with evolving sets of defects and complexity attributes. Our objective, then, is to model reliability across the product line, providing a holistic view of reliability that we find is rarely achieved in practice.

We consider the evolution of systems as progressing to the point where a system has met the defect count goal at a specified test time and can be released for operational usage. As long as this goal is not satisfied, the software systems continue to evolve, as the result of continuing testing. Complexity metrics are used to assess whether defect count is decreasing.

Figure 2 provides an overview of the evolution of NASA's JM1 satellite software as it progresses through defect detection, testing, and defect removal. As the systems that comprise this software transition through these process steps, they are subjected to defect density, defect count, and testing efficiency checks that will be described in detail later.

4. Using Defect Density as a Metric of Software Evolution

Defect density *DD (t)*, that is number of defects divided by program size, present at time *t*, is considered the de facto measure of software quality (and reliability) [3]. While this is true, we consider it to be also a measure of software evolution because, as the software evolves based on the removal of defects, defect density should asymptotically approach zero. For the NASA software that we evaluated, we will consider .02 as the defect density goal (i.e., 1 defect per 50 source language statements). This goal is judgmental based on the range of defect densities observed for this application.

Figure 3 is used to investigate whether the evolution of software releases is achieving the defect density goal. To do this, we plot predicted *DD (t)* for three systems of modules. System 1 is tested for t = 22 and Systems 2 and 3 are tested for *t* = 22 and then operated for 8 time intervals. We see that none of the systems consistently achieve the goal of .02 defects per source language statement. Thus additional inspection and testing is necessary.

4.1 Cost of Testing

While we have no information on the actual cost of testing for the NASA application, we can use a surrogate, *n*, the number of random tests required to achieve a probability of failure bound P_f, with a confidence of α, in equation (0.3) [8]:

$$n = [\frac{\log(1-\alpha)}{\log(1-P_f)}] - 1 \quad (0.3)$$

The probability of failure bound can be computed from the defect count at time t is summed over m modules, as follows:

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

$$P_f = \frac{D(t)}{\sum_{t=1}^{m} D(t)} \quad (0.4)$$

Equations (0.3) and (0.4) are put to use in Figure 4 that shows that the situation is stabilizing with respect to number of tests, as additional systems are analyzed. That is, the number of tests for System 2 has much less variability than for System 1, and System 3 has much less variability than System 2. This is evidence of progress in the evolution of releases, with respect to operating time but, more important, it will be necessary to see whether the *reliability* goal is met based on number of tests. This check is made in Figure 5, where if System 3 is tested 250 times, the predicted defect count is less than one. If the reliability goal, in terms of defect count is less than one, the reliability goal has been achieved. Whether NASA could afford 250 tests can be assessed by predicting the number of defects removed in the tests and whether this would be a cost-effective process. This issue will be addressed next by computing the probability of repairing defects at time *t*, $P_r(t)$, in equation (0.5) [15].

Probability of defect repair: $P_r(t) = \frac{n_r(t)}{D(t)}$, (0.5)

where $n_r(t)$ is the number of defects repaired at time *t*. Now to obtain another expression for $P_r(t)$, we can reason that it is the ratio of cumulative number of tests at time *t*, for detecting and repairing defects, to the cumulative number of tests over *m* modules, where *i* is a summation index, in equation (0.6):

$$P_r(t) = \frac{\sum_{i=1}^{t} n(t)}{\sum_{i=1}^{m} n(t)} \quad (0.6)$$

Then equating equations (0.5) and (0.6), we compute the expected number of defects repaired at time t in equation (0.7):

$$n_r(t) = \frac{D(t) * \sum_{i=1}^{t} n(t)}{\sum_{i=1}^{m} n(t)} \quad (0.7)$$

Once $n_r(t)$ is computed, we define testing efficiency $e(t)$ at time *t* in equation (0.8), which is also equal to the probability of defect repair in equation (0.5):

$$e(t) = n_r(t) / D(t) \quad (0.8)$$

Figure 6 indicates success in testing and therefore success in achieving the reliability goal because at test time $t = 28$, for System 3, testing efficiency exceeds the limit of .90. At $t = 30$, we expect that all defects would be repaired.

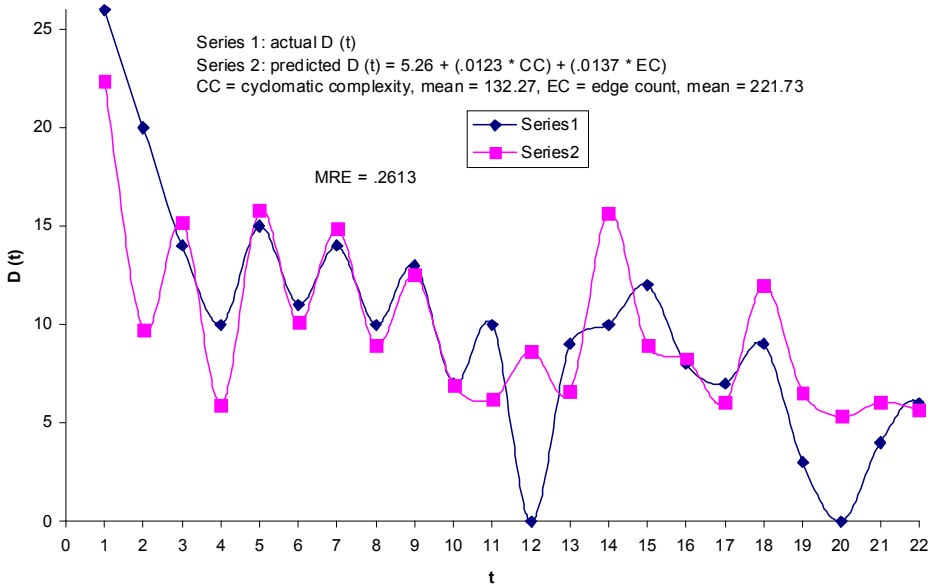


Figure 1. Defect Count D (t) vs. Test Time t, NASA JM1 Software

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

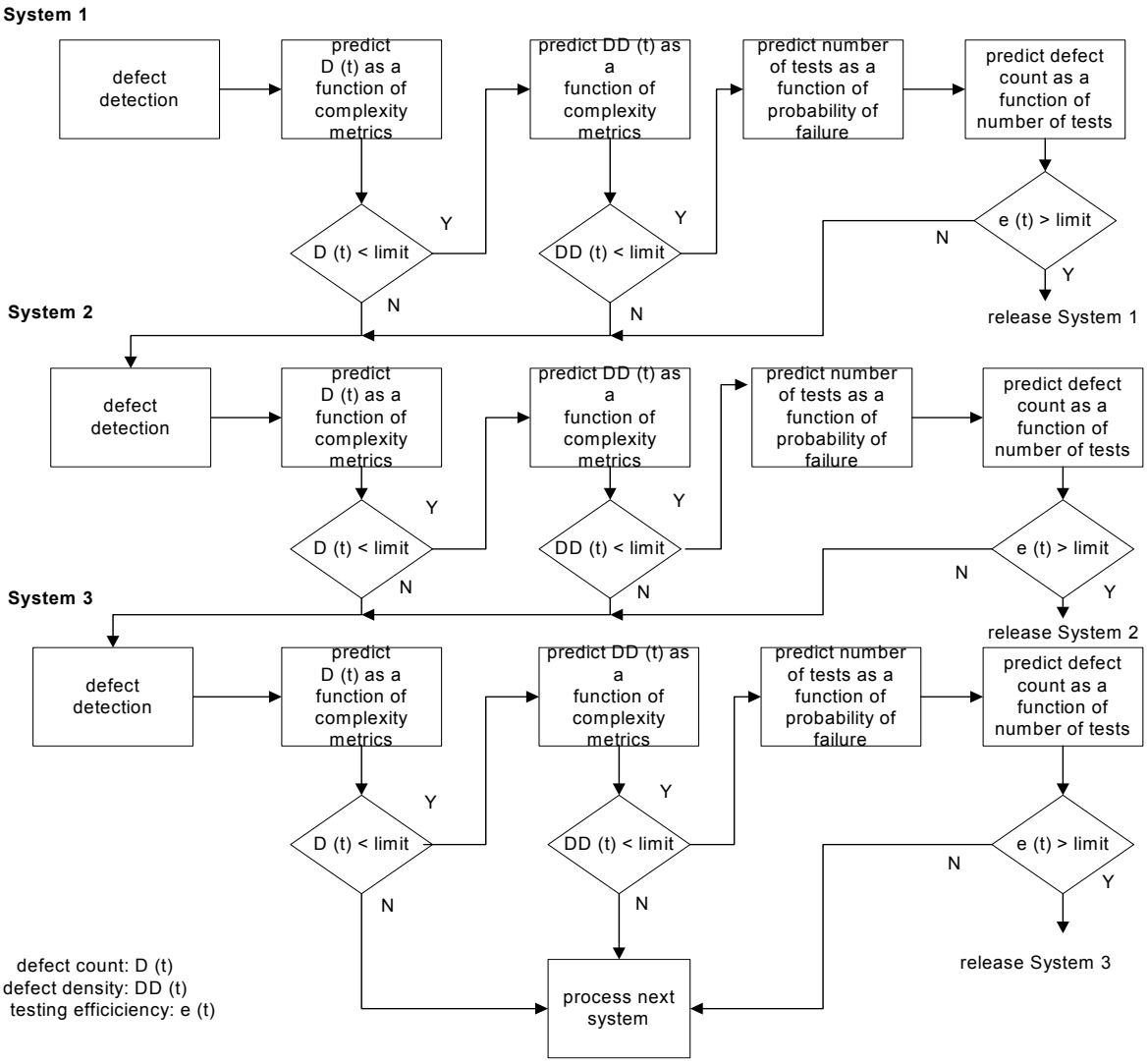
5. Efficiency of Testing Using Directed Graph of Program

One of the important benefits of automatically generating test cases is freedom from bias. When called upon to construct test cases, software developers usually suffer from biases that result in under-exploration of many testing regions.

Knowledge of system internals and operational expectations influence the design of test suites. Automated tools can augment the testing process by providing test cases that are not susceptible to the same biases as the human developers [17]. One major factor left out of this argument is that humans design the test tools and their biases are injected into the tools! Therefore, we believe it is important to consider the internals of software when developing test strategies, and to do it in an unbiased fashion by exercising all paths in tests, as described below.

Previously, we had predicted number of tests based on probability of defect repair that, in turn, is based on predicted defect count in equation (0.1). While predicted defect count uses program complexity metrics in its computation, there is structural information of a program, like its directed graph, that is *not* used. In order to assess testing efficiency, using program structure, we developed a directed graph of module 30 of System 3 in Figure 7. We picked this module because Figure 6 told us that its testing efficiency, as computed in Equation (0.8), is 1.000. Thus, we were interested in comparing testing efficiency by two methods—the second method based

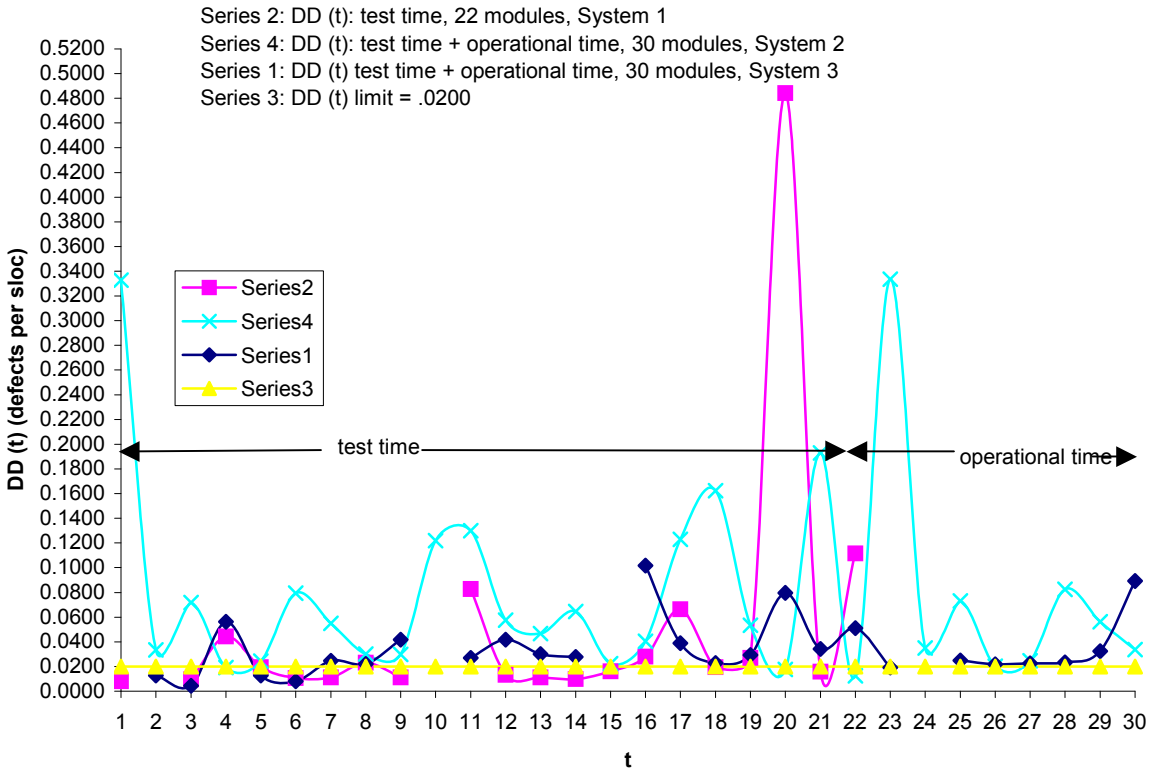
Figure 2. NASA Satellite Software JM1 Evolutionary Process



Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

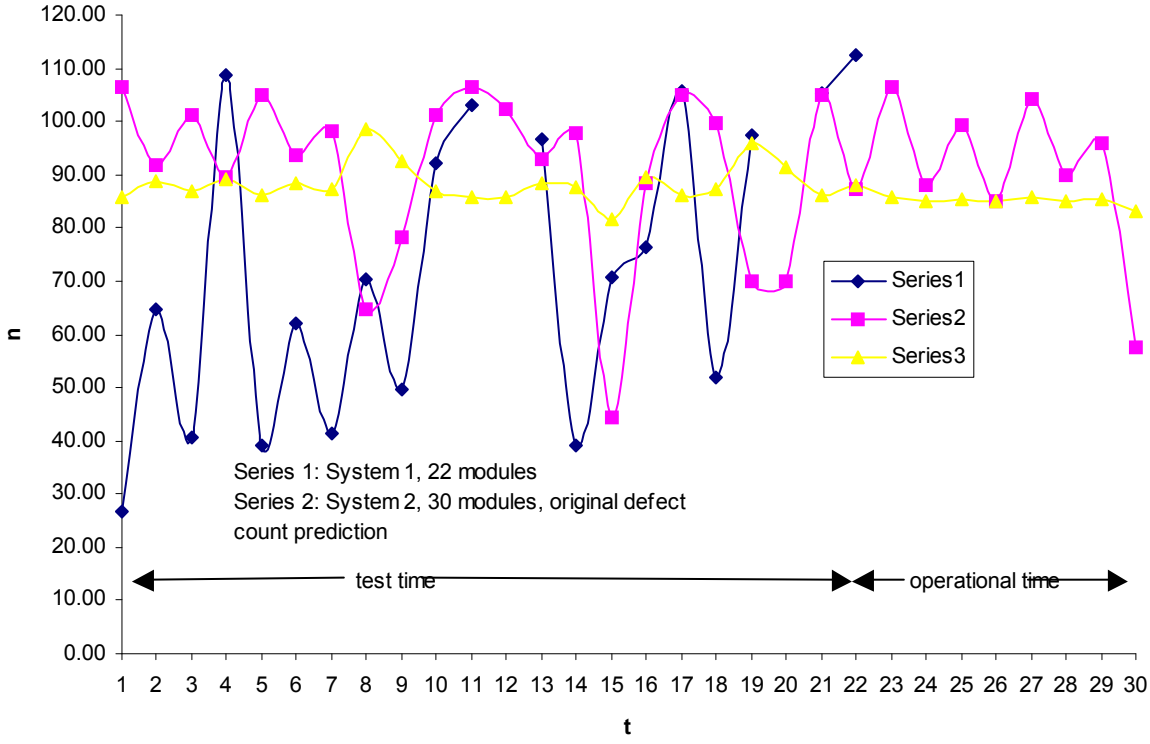
on efficiency of detecting defects using path testing in a directed graph.

Figure 3. Evolution of Predicted Defect Density DD (t) vs. Time t, NASA JM1Software



Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

Figure 4. Evolution of Predicted Number of Random Tests n (t) vs. Time t, NASA JM1 Software



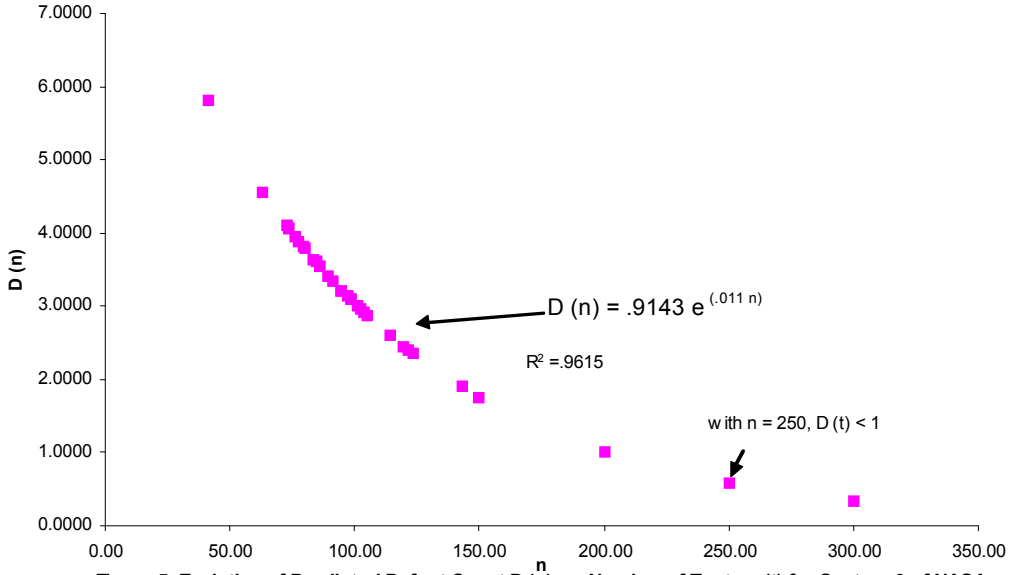


Figure 5. Evolution of Predicted Defect Count D (n) vs. Number of Tests n (t) for System 3 of NASA JM1 Software

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

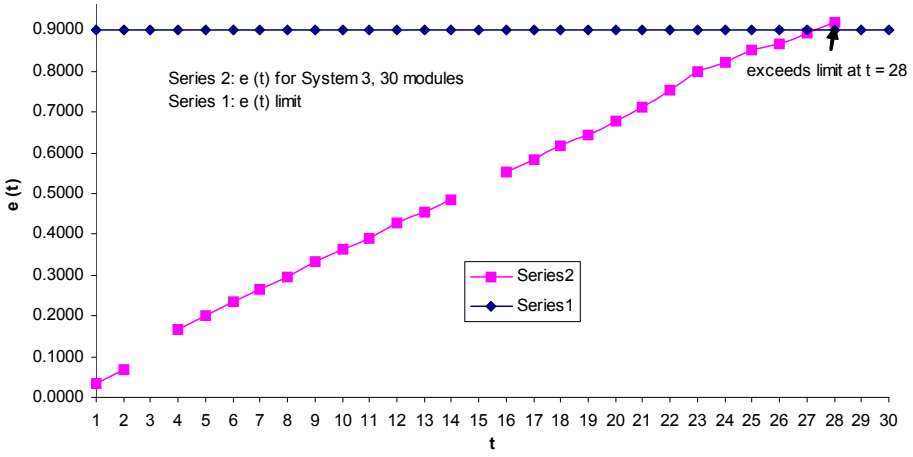
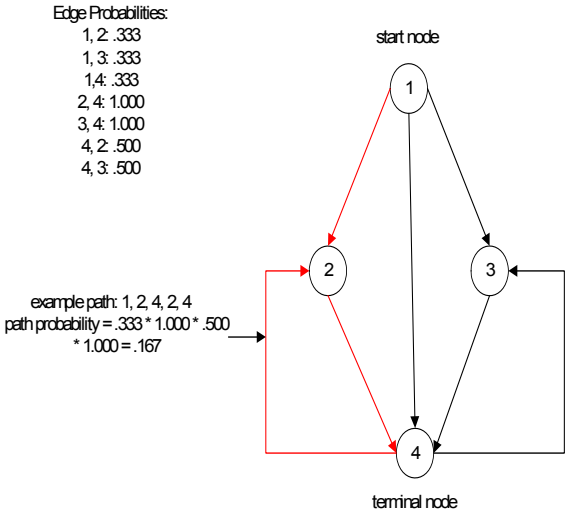


Figure 6. Testing Efficiency e (t) vs. Test Time t

Figure 7. Directed Graph of NASA JM1 Module 30 in System 3



We document the parameters of the directed graph of Figure 7 in Tables 1 and 2, where we assume that edge probabilities are equal and that only a single iteration is performed where loops occur in the program. We make these assumptions because we do not have the code for the NASA JM1 software. We only have cyclomatic complexity, edge count, and defect count per module. While these assumptions obviously affect defect prediction for individual modules, there is no effect on a relative basis across modules. Referencing Table 2, where the total expected number of defects detected = 4.0030 *D (t)* and the number of path tests required to detect the defects = 7, the path testing efficiency is computed as follows:

Testing efficiency = 4.003 *D (t)* / 7 =. 5719 *D (t)*.

In other words, for each defect present in the software, we would expect to detect .5719 of them with path testing. Now this is much less than the *e (t)* = 1.000 from Figure 6 based on random testing. However, the latter does not account for the structure of the software, and therefore, its accuracy may suffer relative to the directed graph approach. Thus, we believe it is better to do the analysis of Tables 1 and 2 and Figure 7, but recognizing the greater time and effort involved.

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

Table 1. Paths and Path Probabilities CC =4, EC = 7, NC =4

Path Identification	Edge Sequence	Path Probability
1	1, 2, 4	.333
2	1, 3, 4	.333
3	1, 4	.333
4	1, 2, 4, 2, 4	333 * 1.000 * .500 * 1.000 = .167
5	1, 3, 4, 3, 4	333 * 1.000 * .500 * 1.000 = .167
6	1, 4, 2, 4	.333 * .500 * 1.000= .167
7	1, 4, 3, 4	. 333 * 1.000 * .500 * 1.000 = .167

Table 2. Paths, Edges, and Expected Number of Defects Detected (one defect per edge assumed)

Path Identification	Path Probability	Number of Edges	Expected D (t)
1	.333	2	.333*2 = .666
2	.333	2	.333*2 = .666
3	.333	1	.333*1 =. 333
4	.167	4	.167*4 = .668
5	.167	4	.167*4 = .668
6	.167	3	.167*3 = .501
7	.167	3	.167*3 = .501
Total	7 path tests	4.003 D (t)	

8. Conclusions

We found that for the NASA satellite JM1 software, the complexity metrics, cyclomatic complexity, and edge count are accurate predictors of defect count and defect density. However, we do not claim that these results would hold for all software because other researchers have arrived at contrary results. For example, in [3], the authors did not find evidence in evaluating *their particular software* that complexity metrics are good predictors of either fault-prone or failure-prone modules. In addition, they have reservations about using fault density as a predictive metric. Since defects usually lead to faults, the same assessment could be made of defect density.

In another depressing report, it was concluded that a major problem is the inability of complexity measures to predict different software quality attributes accurately, for example, reliability, and maintainability [18]. But what is meant by "accurately"? In our experience, over a number of NASA and DoD software projects, a reliability prediction accuracy of MRE = .20 – .30 can be obtained. While this may not be considered accurate, it is better than no prediction! Therefore, all is not lost because while results will vary dependent on the characteristics of software, it is important to adopt a *process*, such as the one used in this research, to evaluate software reliability.

We also conclude that using an evolutionary model of software reliability and complexity is advantageous because it provides the ability to iterate the complexity predictors across successive systems until the desired reliability is achieved and the software can be released. In addition, several criterions for judging whether software should be released should be used. Among these are satisfying defect count, defect density, and testing efficiency thresholds.

Finally, we observed that when evaluating testing efficiency, it is important to use structural information such as the directed graph of a program.

Acknowledgment

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

References

1. C. Andersson and P. Runeson, A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems, *IEEE Trans. on Software Engineering*, **33**(5):273-286, May 2007.
2. S.A. Bohner, An Era of Change-Tolerant Systems, IEEE Computer, June 2007.
3. N.E. Fenton, N. Ohlsson, Quantitative Analysis of Faults and Failures in a Complex Software System, *IEEE Trans. on Software Engineering*, **26**(8):797-814, August 2000.
4. J.L. Fiadeiro, Designing for Software's Social Complexity, *Computer*, **40**(1):34-39, January 2007.
5. B. George, S.A. Bohner, R. Prieto-Diaz, Software Information Leaks: A Complexity Perspective, In *Proc.*

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.

Ninth IEEE International Conference on Engineering Complex Computer Systems (ICECCS '04), pp 249-258, IEEE Computer Society Press, 2004.

6. G.K. Gill and C.F. Kemerer, Cyclomatic Complexity Density and Software Maintenance Productivity, *IEEE Trans. on Software Engineering*, **17**(12):1284-1288, December 1991.
7. N.E. Gold, A.M. Mohan and P.J. Layzell, Spatial Complexity Metrics: An Investigation of Utility, *IEEE Trans. on Software Engineering*, 31(3):203-212, March 2005.
8. W. Jones and M.A. Vouk, Field Data Analysis, In M.R. Lyu, ed., *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill Book Company, 1996.
9. M. M. Lehman, Rules and Tools for Software Evolution Planning and Management, International Workshop on Feedback And Evolution In Software And Business Processes, Imperial College, London, U.K., 10-12 July, 2000. Revised and extended version in Annals of Software Engineering **11**:15-44, November 2001.
10. T. McCabe, A Software Complexity Measure, *IEEE Trans. on Software Engineering*, **2**(4):308-320, April 1976.
11. J.C. Munson and D.S. Werries, Measuring software evolution, In *Proc. Third International Software Metrics Symposium (METRICS'96)*, 1996.
12. J. Peña, M.G. Hinchey, M. Resinas, R. Sterritt and J.L. Rash, Designing and managing evolving systems using a MAS product line approach, *Science of Computer Programming* **66**, pp 71–86, Elsevier, 2007.
13. N. F. Schneidewind, Software Evolution and Feedback, In N. Madhavji, ed., *Requirements Risk and Software Reliability*, pp 407-421, John Wiley & Sons, 2006.
14. N.F. Schneidewind, Software reliability engineering process, , *Innovations in Systems and Software Engineering: a NASA Journal*, 3(2):179-190, Springer, September 2006.
15. N. Schneidewind, Software Defect Process and Product Model for High Assurance Applications, *AIAA Journal of Aerospace Computing, Information, and Communications*, Volume 4, March 2007.
16. L. Sha, Using Simplicity to Control Complexity, *IEEE Software*, 18(4):20-28, July/August, 2001.
17. A. Watkins, D. Berndt, K. Aebischer, J. Fisher and L. Johnson, Breeding Software Test Cases for Complex Systems, In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, Track 9, 2004.
18. C. Wohlin, Revisiting Measurement of Software Complexity, In *Proc. Third Asia-Pacific Software Engineering Conference (APSEC'96)*, 1996.

Authorized licensed use limited to: University of Limerick. Downloaded on July 19,2010 at 10:28:40 UTC from IEEE Xplore. Restrictions apply.