RC24045 (W0609-007) September 1, 2006
Computer Science

IBM Research Report

"Tail Splitting" to Predict Failing Software Modules -- with a Case Study on an Operating Systems Product

Ram Chillarege
Chillarege Inc.

P. Santhanam
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

"Tail-Splitting" to Predict Failing Software Modules - with a Case Study on an Operating Systems Product

Ram Chillarege
Chillarege Inc.
ram @chillarege.com
www.chillarege.com

P. Santhanam
IBM T.J. Watson Research Center
pasanth @us.ibm.com
Hawthorne, NY 10532

Abstract -- "Tail-Splitting" is a new technique to identify defect prone modules by enhancing the focus of the Pareto distribution by a development process factor. The simple yet powerful influence of a varying tail membership as a function of development process phases is captured by the tail-split-string which tags each module. The case studies on an operating systems product demonstrate that the tail-split-string identifies a small set of modules with a high probability of field failure. The tail-boundary in the algorithm provides for a natural tuning parameter to control the size of the identified set to suit the resources available for rework. Release managers have found that the method is particularly useful to sift modules, with low false positive, for late stage rework.

1. Introduction

Managing a software release is a complicated task. With legacy products there is always the concern about the inherent defect rate of the base code. These defect rates need to be managed and reduced as products gain market share and customers become more demanding of a product's quality. This leads to a regular amount of rework, in the background, to identify latent defects and remove them from future releases. The benefits are not merely to the customer but also to the developer since it reduces the overall cost of test and development, which grows non-linearly with size and complexity. At a gross level there exists knowledge in an organization based on historical data and experience. However we need module identification at a finer level of granularity to drive specific test and development activities with resources attached to them.

The problem of identifying faulty modules has been well studied. References [1-12] provide a sample of this large body of work. The different approaches vary in terms of input factors, analyses techniques and outputs predicted. Consequently they require various degrees of tracking and historical data. Analysis techniques include: classification trees [1], module ordering models [2], discriminant analysis [3,4], regression models [5], Dempster-Shafer belief networks [6], random forests [7], complexity analysis [8], neural networks [9], etc. A recent study [10] even considered the number of defects discovered by static analysis as a predictor of fault prone modules. In addition to the work based on code/product based metrics some studies [11-14] have also considered software process related metrics in predicting fault prone modules. Some process metrics are: development activity, designer experience in the module [11], number of defects found by process phases (e.g. function test, system test, integration test, etc.) [12,13], and the number of defects from prior releases [14]. Clearly, there is no one simple answer to this problem. The applicability of a particular approach depends largely on the nature and volume of data available and the investment to achieve the desired improvement.

The Pareto principle [15] (also called the 80/20 rule) is widely used to focus quality improvement. When applied to software defect densities, it translates to 20% of the modules containing 80% of the defects. A belief behind such an approach is that modules with most faults are fault prone due to some intrinsic reasons. These reasons could be complexity, legacy, skill, etc., thus necessitating further scrutiny and rework. Biyani and Santhanam [14] analyzed defect data from four releases of a large mature application product with thousands of modules and showed that that higher number of pre-release defects in modules during development did predict higher number of defects in the field usage, supporting the above assumption. However, the results from Fenton and Ohlsson [13] based on a study of the subset of a few hundred modules from two releases of a large telecommunication system software were in conflict with the results from [14]. The value of empirical studies is that they provide us perspectives on the nature of software engineering. The answers to the underlying nature of product defects may lie in understanding whether testing and customer usage are approaching the limit of the finiteness of defects in software. However, in large complex software products with on-going development work, it is unlikely that we approach this limit.

This paper describes a method called 'Tail Splitting' to focus the application of the Pareto analysis in identifying fault-prone modules. The key idea is to include information about the effect of the pre-release test phases in exposing the defects across the development cycle. This approach gives a finer control in managing the number of the defect prone modules identified to a few that are most likely to fail in the field. We report two case studies: the first uses field defect data to validate the fault-prone module selection and the second that supplements the field validation with explicit comments on the modules from the development team.

2. Data

Our approach to this problem has been motivated by the specific needs of the release manager in an operating system product with a large and robust code base with a mature development process. As markets evolves and new

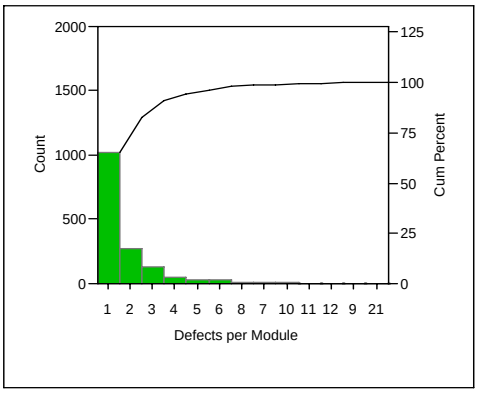

Figure 1: Pareto distribution of 2971 Severity 1 and 2 defects in 1565 modules. Left axis is the count of modules and right axis the percentage.

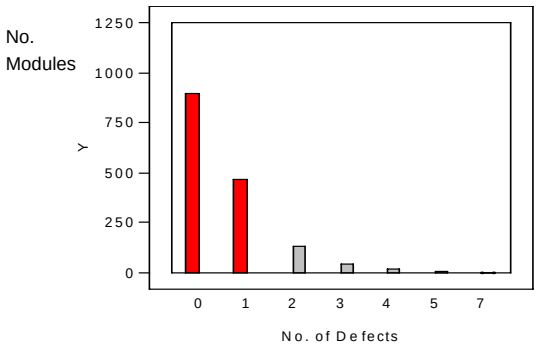

Figure 2: For each phase of testing, the tail boundary partitions the disribution. Modules in the tail get the label '1' and others get '0'.

features are added the demand for higher product quality accelerates. The release we studied had a base of several tens of million lines of code and half a million lines of new and changed code. The release manager wanted to identify a small number of modules with a high probability of field failure. The practical limitations were that input data should be readily available and the analysis be simple and straightforward. This would then increases the chances that managers could relate to the data and commit scarce resources to take action.

In this work, a 'module' refers to the basic program unit which is tracked and typically ranges in size up to a few thousand lines of code. This product contained more than thirteen thousand modules. A 'defect' refers to a specific unique programming fault exposed by testing or customer usage. The pre-release testing activities exposed 5544 unique defects across all the four severities (1 through 4, 1 being the highest) touching 3732 modules. Thus, there were a large number of modules with no defects. To reduce the scope of analysis to a manageable size, the development team wanted to focus on defects with severities 1 and 2. This step reduced the scope of the analysis to 2791 defects in 1565 modules. Figure 1 shows the Pareto distribution of modules vs. the number of defects, along with the cumulative distribution. Focusing on modules with more than 3 defects, yields 276 modules (17.6%) contributing to 1229 defects (44%). If we consider modules with more than 4 defects, 145 modules (9.3%) contained 836 defects (33%) - all these amount to numbers of modules, well beyond the resources the release manager could allocate for late stage rework.

The release manager wanted a range of 20-60 modules that have a high chance of failing in the field. A larger set with higher false positives would be of little value since it's resource intensive. The method should factor recent data and have a tuning parameter to control the size of the indicated set. If the method was verified to yield low false positives, then the modules could be the subject of rework competing for scarce resources from the mostly stretched and skeptical development managers.

3. Tail-Splitting Algorithm

3.1 Process Influence

In a mature organization with decades of experience in developing release after release, people tend to anticipate the amount of defect removal efforts based on release content and experiences from prior releases. The development process and the testing methods have a significant influence on whether defect prone modules get the adequate coverage. As we progress through different phases of defect removal activities such as inspection, unit testing, function testing, and system testing, each exercises the code in different ways. The ability of a module to pass without defects through one or more of these phases is an indication of its robustness. On the other hand when modules tend to break in more than one of these phases, it is an indication of the fragility in the modules. This thought, we have found, agrees with instinct of many development managers and is a motivating factor in developing this algorithm and the quantification of this measure. The tail splitting algorithm incorporates the notion that successively defective modules across phases are more prone to fail in the field than others. Our case studies corroborate this observation and validate the value of the algorithm. Since the data for this identification comes from the current release the calculations are done in-process and not retrospectively. Such an in-phase calculation automatically factors in the influence of any recent best practices or process changes incorporated in the release.

3.2 Tail-Boundary

In Figure 2, we show the defect density distributions corresponding to a particular phase of the project, say, Function Test. We first pick a tail-boundary, which partitions modules into two groups. Those with defect density greater than or equal the tail boundary, and those less than the tail boundary. One choice for the definition of the tail-boundary is the mean number (μ) of defects, in that phase. If the

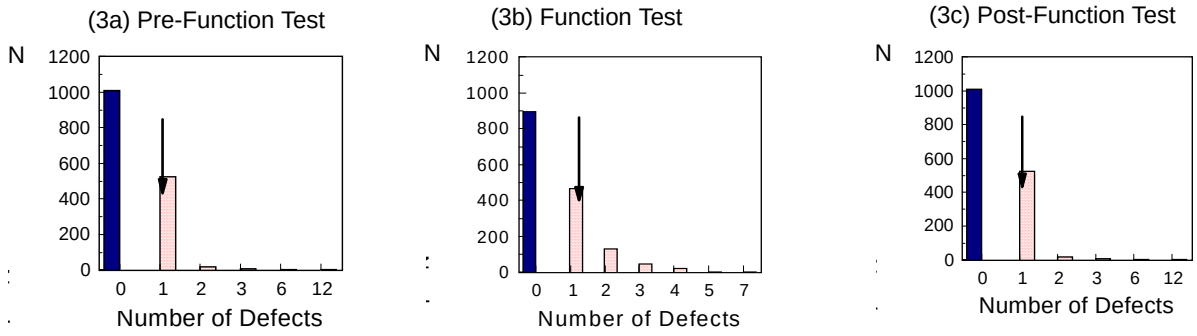

Figure 3: The distribution of modules by number of defects for the three phases: Pre-Function test, Function test and Post-Function test. N, is the number of modules and the arrow points to the tail boundary at 1 defect/module.

number of modules identified by the analysis is too large to allocate actions satisfactorily, then we could choose the tail-boundary at $\mu + \sigma$, where σ is the standard deviation.

3.3 Tail-Split-String

Each module is tagged by a tail-split-string that captures how the module traversed through the development phases in terms of its position in the defect density distribution. The length of the string is equal to the number of phases that are used for this analysis. The tail-split string captures the influence of the process on the Pareto at each of the phases. The following description communicates the algorithm and its implications. Let us consider a development process that could be partitioned into three distinct phases, A, B and C. These phases are typically test or verification phases, but could as well be design and development so long as there are distinct phases that identify defects and tag them against modules. Figure 2 illustrates fault density distribution for one such phase where a given module may or may not belong to the tail and accordingly receive a label of either a 1 or a 0 respectively.

As the development process progresses from phase A to phase B, there are changes in the faults that are identified yielding a different fault density distribution. Statistically these distributions might look similar; but the parts that contribute to making up of that distribution could be very different. A module may belong to the tail in phase A, but may not belong to the tail in phase B, thereby the label it acquires through the second fault distribution could be different from the one that it had during phase A. Thus, by generating a sequence of these fault density distributions we would be bringing into effect the factors of the development process that influence the fault density distributions. Each module that goes through the development process would result in a label of 0's and l's that reflect whether it belonged to the tail or not. This sequence of 0's and l's that label a module is called the tail-split string reflecting the dynamics of the software development process. For a three phase process, there are 8 three-bit strings possible and each module will acquire one of the eight strings as a result of the analysis of its membership in the tail.

Let us examine the tail-split-strings with leading 0s followed by trailing 1s. They reflect modules that were not considered very defective early, but that changed later in the development process, and there is less guarantee that they will not fail in the field, especially when there is more than one consecutive trailing 1. Modules with tail-split-strings such as: (000), (100), (010) and (110) are more likely to be considered satisfactory while modules with tail-split-strings (111), (001), (011) and (101) will be considered risky. Among the risky modules the tail-split-string of (111) represents the most consistent behavior and hence a top priority. On the other hand, when there is a suspicion that a certain phase did not have adequate testing, then those modules with the tail-split-string of trailing 0s with leading 1s may represent inadequate test. For instance (110) could represent inadequate system test.

3.4 Fine Tuning

3.4.1 Tuning: Number of Phases

The number of phases that can be used with this algorithm is flexible. At the same time the definition of what makes up a phase is flexible. A practical application usually makes these choices based on what is meaningful to the development process. Often there are many sub-phases in a development process, which can be collapsed to a set of larger phases. We have found that practitioners would run the Tail-Splitting algorithm with multiple choices of phases to see what difference occurs in the outcome. For our case studies, we grouped the various sub-phases of development into three broad phases termed pre-function test, function test and post-function test. This choice was motivated by the amount of data in these phases which helped tie these conclusions to specific activities in the development organization. This usually provides for better understanding of the data, and communication on the impact of the results.

At the same time, it is not necessary that this analysis be restricted to the phases of a development process or a release. The three phases could very well be three different releases of the same product. In which case, we would analyze the change in membership of what constitutes faulty parts across releases.

Table 1. The first row are the eight tail-split-strings representing the three phase process. The second row contains the numbers of modules matching the strings. The third row is the perentage of modules that failed in the field after six months and fourth row after 30 months.

Tail-Split-String	**(000)**	**(001)**	**(100)**	**(010)**	**(101)**	**(110)**	**(011)**	**(111)**
Number of Modules	0	391	350	439	82	220	64	19
% of modules failing after 6 months		8	15	28	10	50	42	63
% of modules failing after 30 months		13	22	35	10	58	52	74

3.4.2 Tuning: Tail Boundary

Another tuning parameter to obtain fewer numbers of modules is the tail boundary. Moving the boundary to a larger value will reduce the number of modules chosen for consideration. In this paper, the two case studies illustrate the different choices of tail boundary.

4. Case Studies

The two case studies discussed below involve the same set of data across three phases during development, defined as pre-function test, function test, and post-function test. The development process has several sub phases which were collapsed into these three phases, which are meaningful to the development team. Function test is mostly black box testing. Pre-function test can be unit testing and some inspection as well, whereas, post function test tends to have far greater integration testing, systems testing, performance testing, etc. As discussed in the earlier section, Tail-Splitting lends itself to any number of sequential phases that can be deemed meaningful to a process. It also does not have to be a waterfall process - any process, but with defined phases with changes in activities will suffice.

First case study used 1 defect/module in each phase as the tail boundary, while the second case study used 2 defect/module as the tail boundary. We evaluated the performance of the tail splitting algorithm against the field data which recorded the actual incidence of failed modules. The field data was tracked for 30 months. In the second case study, we also include the comments on the selected modules by the development team.

4.1 First Case Study

Figures 3a-3c show the defect density distribution for three phases. For Pre-Function test, $\mu = 0.63$, $\sigma = 0.90$, for-Function test, $\mu = 0.71$, $\sigma = 1.17$, for post-Function test, $\mu = 0.39$, $\sigma = 0.65$. Looking at the mean defect density per module, it is already clear that the different test activities are not equally effective in exposing defects in the modules. Table 1 shows the results of the analysis with 1 defect/module as the tail boundary. There are 19 modules that get the tail-split-string (111) and represent the modules that consistently fail across the test phases. The

D	**N**	**R**	**(001)**	**(100)**	**(010)**	**(101)**	**(110)**	**(011)**	**(111)**
1	1016	1565	377	275	364	0	0	0	0
2	273	549	11	60	43	67	55	37	0
3	131	276	2	12	21	13	62	17	4
4	54	145	0	3	7	2	35	4	3
5	33	91	0	0	3	0	26	2	2
6	28	58	1	0	1	0	19	3	4
78	6	30	0	0	0	0	6	0	0
8	8	24	0	0	0	0	6	0	2
9	1	16	0	0	0	0	1	0	0
10	6	15	0	0	0	0	4	0	2
11	5	9	0	0	0	0	5	0	0
12	3	4	0	0	0	0	1	0	2
21	1	1	0	0	0	0	0	1	0
Total	1565		391	350	439	82	220	64	19

Table 2.

D: Defects/Module
N: No. of Modules
R: Reverse Cumulative Count

The table shows the composition of the Pareto distribution versus the tail-split-string populations. Column R has the reverse cumulative count of modules in the tail, starting from row D=21. This is useful to compare Tail-Splitting with the Pareto, by picking a small number of modules and studying how the two would make different choices.

Table 3. Analysis results for the second case study, with tail boundary at 2 defects/module. The first row are the eight tail-split-strings, the second row the numbers of modules identified by the tail-split-strings, and the third row the percentage of modules that failed after six months in the field.

Tail-Split-String	**(000)**	**(001)**	**(100)**	**(010)**	**(101)**	**(110)**	**(011)**	**(111)**
Number of Modules	1270	24	156	62	2	45	5	1
% of modules failing after 6 months	19	25	35	45	50	64	80	100

field data 6 months after product release shows that 63% of those modules failed and the percentage increases to 74% after 30 months of customer usage. (110) and (011) are next in the priority list. In addition, note the marginal increase in the failure probability between 6 months in the field and 30 months for all the tail-split-strings.

4.1.1 Comparing Tail-Splitting with the Pareto

These data also allow us to compare the performance of Tail-Splitting with the classical Pareto approach. Table 2 shows the distribution of defects in modules grouped by the tail-split-strings. If we were to pick most fault prone modules, based on the simple Pareto Distribution (represented in the first two columns) we will start with the one module with 21 defects, then 3 modules with 12 defects, then 5 modules with 11 defects, etc., in that order. In contrast, using the Tail-Splitting method, we will select the 19 modules with the string (111) in the last column which comprises 4 modules with 3 defects, 3 modules with 4 defects, 2 modules with 5 defects, etc. Interestingly, we will not be selecting the module with 21 defects as the highest priority since it has the less risky (011) string. This illustrates the fact that the population of modules selected can be quite different based on the two methods. The probability of failure being the highest for the (111) modules, the same number of modules selected by the simple pareto will have less probability of failure in the field.

Another perspective that illustrates the differential benefit of Tail-Splitting over the Pareto is illustrated when chosing a fixed number of error prone modules. Picking the top 30 modules (using the reverse cumulative count column in Table 2) would result in different choices. Tail-Splitting would pick several more modules from tail-split-strings of (111) and (011) which have a high probability of field failure. The Pareto selection gets over populated by modules with the tail-split-string of (110).

4.2 Second Case Study

This study used a tail-boundary of 2 and Table 3 shows the distribution of the modules by the tail-split-string and the percentage of modules that failed after six months of customer usage. It is quite fascinating that the modules with multiple consecutive 1's failed with a higher likelihood. Do note that while the number of modules with multiple 1s is lower it is still interesting since we are after finding fewer modules to rework, but want to find those with a higher chance of failing in the field. The fact that a change in the tail boundary yielded results very similar to the first case study reinforces our understanding that the tail boundary can be used as a tuning parameter. The choice of tail boundary determines the number of modules that will be flagged as more error prone, and the tail-split-string sequence helps associate a probability of field failure. This is particularly useful to a release manager, who could use this to sift the modules towards the end of the release and choose the range of modules they have resource to focus on.

4.2.1 Developer Review

We had the development team review some of the modules that were identified by the tail-split-string, to associate a technical and qualitative understanding of the types of modules that were indicated. From failure data and the discussion in section 2, it is evident that tail-split-strings that have consecutive 1s, especially towards the right of the string are more likely to fail. It is interesting to note the comments made by developers on some of these modules. Table 4 gives some of these insights against the tail-split-string pattern.

Table 4: Sample comments by the development team

Tail Split String	**Samples of developer's insights**
111	"Critical timing module rewritten this release"
011	"Complex module, abbreviated development"
011	"Serializaiton problems surface under stress"
011	"Needed a formal design change review"
101	"Very complicated code - will remain problematic unless redesigned"
001	"Old module, only one person understands this"
001	"Old module - steady stream of field failure"
001	"Many individual changes - needs redesign"
110	"Inadequate testing"

5. Summary

This paper presents a technique that can enhance the notion of the Pareto principle by factoring in the process influence of development. The technique called Tail-Splitting studies the change in the makeup of the tail of the distribution as the product advances through development and test. These changes factor in the influence of test and development practices on the defect density. The Tail-Splitting algorithm divides the tail membership as a function of the process - represented by a string of 0s and 1s. The strings with trailing consecutive 1s identify groups of modules that are far more likely to fail in the field.

Application of the Tail-Splitting algorithm in two case studies using real data from an operating system product helps us to understand and compare the performance of this algorithm against the classic Pareto analysis by observing the performance of these predictors for 30 months of field customer use. While the classical Pareto spots potentially defect prone modules, Tail-Splitting provides the additional parameters and algorithms that identify the few modules that have a higher risk of failing in the field. A side by side comparison of real data reveals the differential advantage of Tail Splitting. The product development team also performed a more detailed assessment of the modules identified by Tail-Splitting method to provide a qualitative understanding of the nature of problems in those modules and their comments support the selection of the modules as defect prone. Overall, Tail-Splitting algorithm provides a simple, intuitive, flexible approach to identify fault prone modules in a large complex software system.

Acknowledgments

We thank the software development team at IBM Poughkeepsie for their cooperation, enthusiasm and encouragement.

References

[1] T.M. Khoshgoftaar, E.B.Allen, Yuan Xiaojing, W.D. Jones, and J.P. Huderpohl, "Preparing measurements of legacy software for predicting operational faults", Proceedings of the IEEE International Conference on Software Maintenance, pp. 359–368.

[2] T. M. Khoshgoftaar, and E.B. Allen, "Predicting the order of fault-prone modules in legacy software", Proceedings of the IEEE International Symposium on Software Reliability Engineering, 1998, pp. 344-353.

[3] T.M.Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, and N. Goel. “Early quality prediction: A case study in telecommunications”, IEEE Software, 13(1):65- 71, Jan.1996.

[4] J.C. Munson and T.M.Khoshgoftaar. “The detection of fault-prone programs”, IEEE Transactions on Software Engineering, 18(5):423-433, 1992.

[5] T.M. Khoshgoftaar, E.B. Allen, R. Halstead, and G.P. Trio, “Detection of fault-prone software mod- ules during a spiral life cycle”, in the Proceedings of the IEEE International Conference on Software Maintenance, 1996, pp. 69-76,

[6] L. Guo, B. Cukic, and H. Singh, "Predicting fault prone modules by the Dempster-Shafer belief networks", Proceedings of the IEEE International Conference on Automated Software Engineering, 2003, pp. 249-252.

[7] L. Guo,Y. Ma, B. Cukic and H. Singh, "Robust prediction of fault-proneness by random forests", Proceedings of the IEEE International Symposium on Software Reliability Engineering, 2004, pp.417-428.

[8] N.Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches", IEEE Transactions on Software Engineering, Vol. 22, 1996 pp. 886-894.

[9] SungBack Hong and Kapsu Kim ,"Identifying fault prone modules: an empirical study in telecommunication system" Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, 1998, pp.179 - 183.

[10]N. Nachiappan, L. Williams, J. Hudepohl, W. Snipes and M. Vouk, "Preliminary results on using static analysis tools for software inspection" Proceedings of the IEEE International Symposium on Software Reliability Engineering, 2004, pp. 429-439.

[11]“EMERALD: A case study in enhancing software reliability”, J. Hudepohl, W. Jones, and B. Lague, Proceedings of the IEEE International Symposium on Software Reliability Engineering-Case Studies, 1997 pp. 85 - 91.

[12]T.M.Khoshgoftaar, E.B. Allen, R. Halstead, G.P. Trio, and R. Flass. “Process measures for predicting software quality”, in the Proceedings of the IEEE High-Assurance Systems Engineering Workshop, 1997, pp.155-160.

[13]N.E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system”, IEEE Transactions on Software Engineering, Vol. 26, 2000, pp. 797-814.

[14]S.H. Biyani and P. Santhanam, “Exploring defect data from development and customer usage on software modules over multiple releases”, Proceedings of the IEEE International Symposium on Software Reliability Engineering , 1998, pp. 316-320.

[15]J. M. Juran, and F. M. Gryna, *Quality Control Handbook,* Fourth Edition, McGraw Hill, New York , 1988.