View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Aberto da Universidade do Porto

— END —
A Lightweight Algorithm to Estimate the Number of Defects in Software

Technical Report

André Riboira[1] and Rui Abreu[2]

HASLab/INESC-TEC
Informatics Department - University of Minho
Campus de Gualtar 4710-057 Braga - Portugal
[1]andre.riboira@fe.up.pt, [2] rui@computer.org

October 2013

Defect precision provides information on how many defects a given software application appears to have. Existing approaches are usually based on time consuming model-based techniques. A viable alternative is the previously presented Abacus algorithm, which is based on Bayesian fault diagnosis. This paper presents a novel alternative approach - coined End - that uses the same input and produces the same output as the Abacus algorithm, but is considerably more time efficient. An experiment was conducted to compare both the accuracy and performance of these two algorithms. The End algorithm presented the same accuracy as the Abacus algorithm, but outperformed it in the majority of executions.

1 Introduction

This document presents the End[1] algorithm, and is organized as follows: This chapter introduces the motivations that underlie the work presented herein, as well as the goals. Chapter 2 details the End algorithm. The following chapter presents the experiment conducted to validate the algorithm. Chapter 4 contains a discussion of the results obtained in the experiment. Finally, Chapter 5 concludes this document.

[1]End is an acronym that stands for "Estimating the Number of Defects".

1.1 Motivation

Defect prediction provides information on how many defects need to be removed before shipping [5]. Existing approaches are usually based on model-based techniques [6]. These require the fitting of a known defect prediction model to estimate the number of defects that still exist in the software. The selection of an appropriate model requires a large amount of historical data containing several metrics collected from the development process, and is usually a time consuming task [10, 7]. Moreover, with few exceptions, no model accurately represents the software under analysis.

The previously presented ABACUS algorithm aims to overcome the issues of the techniques based on defect prediction models [14]. This defect prediction reasoning approach is based on Bayesian fault diagnosis [2] using abstractions of program traces (also known as program spectra) [15]. However, the performance of the ABACUS algorithm suffers from the need to deal with a Minimal Hitting Set (MHS) problem [11]. To circumvent this problem, ABACUS takes advantage of the use of the STACCATO algorithm [1]. STACCATO is a statistics-directed approximate minimal hitting set algorithm. Despite being an viable alternative comparing to the usage of defect prediction models, the performance of the ABACUS algorithm is not ideal when dealing with a large number of simultaneous independent faults [14].

1.2 Goals

The goal of the END algorithm is to use a different approach to avoid the need to deal with a MHS problem [12]. The accuracy of the results is accepted to be lower than the one resulting from the use of the ABACUS algorithm. The reason is that it is often not necessary to know the exact number of faults needed to be fixed, but to have an approximated idea of that value to be able to better manage the software development process [8]. Therefore, a decreased accuracy of the results seem acceptable in favor of increased performance.

2 Algorithm

Based on a program spectra, it is trivial to determine the minimal number of independent defects (considering only the observed failures). One can determine which candidate C participates in the largest number of failed tests. If that candidate is not responsible for all the failed tests, it is known that the software has more than a single defect. To know the minimum number of independent defects, this process must be executed recursively, using in each call the subset of the program spectra that does not include the candidate C of the previous call, neither the tests hit by that candidate.

Algorithm 1 END algorithm.

```
1: procedure END(P, N)
2:     if EV_HAS_ERRORS(P) then
3:         R ← SC(P)
4:         T ← MAX(R)
5:         P ← UPS(P, T)
6:         return END(P, N + 1)
7:     else
8:         return N
9:     end if
10: end procedure
```

The number of recursive calls will reveal the minimum number of independent defects of the software under analysis.

The minimal number of independent defects is usually not a rather helpful information, because software usually embeds some sort of `main` procedure, and the minimal number of independent faults tends to be 1. More useful than knowing the minimal number of independent faults, is knowing the predicted number of independent faults. It is currently not possible to accurately determine this number without knowing the actual defects, but it is possible to roughly predict it. The ABACUS algorithm already do this rather accurately, but at the cost of having to deal with a MHS problem. In turn, the END algorithm uses an approach similar to the one used to determine the minimal number of independent faults. The main difference is that instead of selecting the candidate C which participates in a larger number of failed tests, it selects the candidate T presenting the higher similarity coefficient. Like the process to determine the minimal number of independent faults, the END algorithm also recursively removes the candidate T of the previous call, as well as the tests hit by that candidate, while recording the number of calls. See Algorithm 1 for details.

The END algorithm receives as input the program spectra P (in the same format as received by the ABACUS algorithm) as well as a counter N with the current call. It verifies if the received error vector[2] still has any record of a failed test (using procedure `Ev_has_errors`). If not, the algorithm reached the final result. Otherwise, the algorithm calculates the similarity coefficient for each candidate (using procedure `SC`), resulting into vector R. It then selects the candidate T with the maximum similarity coefficient value, and updates the current program spectra P (using procedure `USP`) removing both the candidate T and the tests hit by it. Finally, the algorithm recursively calls itself using the updated program spectra P and incrementing the counter N.

[2]The error vector is received as input inside P together with the program spectra matrix, and is a binary vector recording for each test if it passed `(0)` or failed `(1)`.

The End algorithm offers as an additional advantage the ability to define boundaries for the recursive process. One can define a maximum number of independent defects to look for, in order to avoid unacceptably long execution delays when dealing with very large projects, or projects with a considerably large number of independent defects. Due to its nature, the End algorithm accuracy deeply depends on how the top positions of the similarity coefficients ranking compares to the actual faulty candidates.

3 Experimental Setup

An experiment was conducted to verify how the End algorithm compares to the Abacus algorithm in terms of performance and accuracy. It was used the same version of the same web-based application presented in the previous Abacus paper [14], as well as both the same test suite and injected defects.

The real-world application used in this experiment was Wordpress[3] 3.4, obtained from trac.wordpress.org repository, changeset 22222. This Wordpress version consists of 439 PHP[4] files, containing 110,891 LOC. It uses PHPUnit[5] [4] as the unit testing software framework of its choice. The unit test suite related to this Wordpress version was obtained from the unit-tests.trac.wordpress.org repository, changeset 1081. This test suite is composed by 1019 passing tests and several other tests marked to be "skipped" during the test suite execution. During this experiment only the passing unit tests were considered to use with the PHPUnit version 3.6.12 framework. These unit tests exercised 1059 different functions of the Wordpress application. The accuracy and efficiency of both algorithms were verified using 255 different faulty versions of Wordpress. These faulty versions were obtaining by exploring all the possible combinations of 8 different faults injected directly into the Wordpress source code. The 8 injected faults are detailed in page ABACUS2013[6] of the Promisedata repository [13].

An implementation of the End algorithm was developed using the same conditions as the Abacus experiment: It was developed using the same version of PHP, running on the same virtual server. This experiment was conducted in the Amazon Elastic Compute Cloud (Amazon EC2)[7] infrastructure [9], using a M1 Medium Instance, with 3.75 GiB memory and 2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit). The executions of both algorithms were interspersed during the tests to minimize the impact of changes in external conditions. Thus, it was possible to not only compare the accuracy of these two algorithms, but also to compare their performance.

[3] Wordpress: http://wordpress.org/
[4] PHP: http://php.net/
[5] PHPUnit: http://phpunit.de/
[6] Repository at: https://code.google.com/p/promisedata/wiki/ABACUS2013
[7] Amazon EC2: http://aws.amazon.com/ec2/

4 Results and Discussion

The results from the END algorithm are roughly the same as those from the ABACUS algorithm in terms of accuracy. Due to its nature, the ABACUS algorithm presented an approximation of the predicted number of defects (in a float format). Rounding this result to the nearest integer, we obtain exactly the same prediction from both algorithms. These results were slightly unexpected because the END algorithm is supposed to simply return a rough approximation of the number of faults, because it does not deal with a MHS problem. However, the END algorithm predicted correctly the precise number of defects in all 255 tests.

While the accuracy of both algorithms shown to be similar, they performed considerably different in terms of execution time (see Table 1). The END algorithm took in average less than a second to return each result, while the ABACUS algorithm took roughly 5 times more. Regarding the minimum execution time, the difference between both algorithms is not relevant, where the ABACUS algorithm outperformed the END merely in 0.022 seconds. However, the maximum execution time is considerably different. The END algorithm proved to reduce more than 67 times the ABACUS execution time in the most time consuming scenario (8 simultaneous independent defects).

Table 1: END and ABACUS execution times.

	END	**ABACUS**	**Δ Performance**
Min.	0.226 s	0.204 s	+ 0.022 s (+ 10,78 %)
Avg.	0.913 s	4.869 s	- 3.956 s (- 81,25 %)
Std. Dev.	0.316 s	11.949 s	—
Max.	1.790 s	120.774 s	- 118,984 s (- 98,52 %)

To better understand the performance differences, refer to Figure 1 where a bar chart is presented containing the average execution time by number of simultaneous independent defects.

When dealing with a low number of simultaneous independent defects, the differences in the performance of both algorithms is not very significant. However, when the number of simultaneous independent defects increases, the performance differences are evident. The END algorithm took approximately $n \times t$ seconds to compute the result, being n the number of simultaneous independent defects and t the time spent to compute a single defect. In turn, the ABACUS algorithm execution time increased exponentially as the number of defects increased.

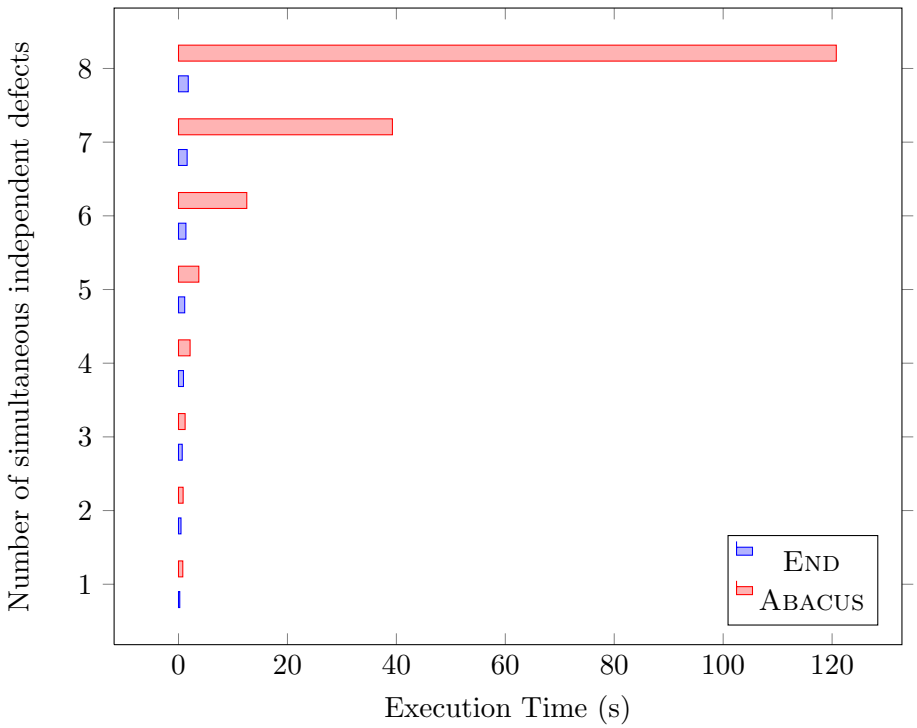

Figure 1: Execution times by number of simultaneous independent defects.

5 Conclusions

Predicting the number of defects present in a given software application is important to evaluate not only how reliable the software appears to be, but also to adequately plan the development process to fix those defects. Model based techniques are very time consuming tasks, and the selected models may not adequately fit the software under analysis.

The previously presented Abacus algorithm [14] tries to overcome these issues using an approach based on data collected during software test executions. However, its performance degrades considerably when dealing with a large number of simultaneous independent defects, mainly because it has to deal with a MHS problem. To solve this issue, the End algorithm presented in this document follows a rather different concept to avoid to deal with a MHS problem, despite using the same input and intending to produce a similar output as the Abacus algorithm.

An experiment was conducted to compare the accuracy and performance of both algorithms. The End algorithm results were the same as the Abacus algorithm in terms of accuracy. However, the End algorithm outperformed the previous algorithm in most of the tests. The worst performance differ-

ence was a 10% increase of the execution time. However, one should notice that these situations only occur when dealing with single faults, and that this increase of 10% corresponds to a delay of less than one tenth of a second. Overall, the END algorithm clearly outperformed the ABACUS, having execution times in average 81% lower. The most significant difference occurred when dealing with 8 simultaneous independent faults, where the END algorithm dramatically outperformed the ABACUS, having an execution time 98% lower, from more than 2 minutes to less than 2 seconds.

The END algorithm proved to be a viable alternative to the ABACUS algorithm in predicting the number of defects existing in software applications.

As future work, the algorithm implementation should be tuned and compiled, and not interpreted (as it was during this experiment, using PHP), in order to increase its performance [3]. Note that although the ABACUS framework is built using PHP, its core processing (staccato and abacus algorithm itself) is implemented in C. Other work that could be done is to significantly increase the number of simultaneous independent defects to verify if the trend seen in this experiment still verifies. Finally, this new approach may also be useful in fault diagnosis field. Perhaps it could be developed to allow a multiple fault diagnosis approach without the need to deal with a MHS problem.

Acknowledgments

This work is financed by Portuguese National Funds through the FCT - Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) within Ph.D. Grant SFRH/BD/88535/2012.

References

[1] Rui Abreu and Arjan J. C. van Gemund. "A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis". In: *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*. Ed. by Vadim Bulitko and J. Christopher Beck. Lake Arrowhead, California, USA: AAAI Press, 2009.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. "Spectrum-Based Multiple Fault Localization". In: *Proceedings of the 2009 IEEE /ACM International Conference on Automated Software Engineering*. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99. ISBN: 978-0-7695-3891-4.

[3] Varsha Apte, Tony Hansen, and Paul Reeser. "Performance comparison of dynamic web platforms". In: *Computer Communications* 26.8 (2003), pp. 888–898.

[4] S BERGMANN. "PHPUnit Official Manual". In: *Git Hub Inc, Alemania* (2010).

[5] Marco D'Ambros, Michele Lanza, and Romain Robbes. "Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison". In: *Empirical Softw. Engg.* 17.4-5 (Aug. 2012), pp. 531–577. ISSN: 1382-3256.

[6] Norman E Fenton and Martin Neil. "A critique of software defect prediction models". In: *Software Engineering, IEEE Transactions on* 25.5 (1999), pp. 675–689.

[7] Norman E Fenton and Martin Neil. "A critique of software defect prediction models". In: *Software Engineering, IEEE Transactions on* 25.5 (1999), pp. 675–689.

[8] Brent Hailpern and Padmanabhan Santhanam. "Software debugging, testing, and verification". In: *IBM Systems Journal* 41.1 (2002), pp. 4–12.

[9] Gideon Juve et al. "Scientific workflow applications on Amazon EC2". In: *E-Science Workshops, 2009 5th IEEE International Conference on*. IEEE. 2009, pp. 59–66.

[10] Paul Luo Li et al. "Empirical Evaluation of Defect Projection Models for Widely-deployed Production Software Systems". In: *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. SIGSOFT '04/FSE-12. Newport Beach, CA, USA: ACM, 2004, pp. 263–272. ISBN: 1-58113-855-5.

[11] Carsten Lund and Mihalis Yannakakis. "On the Hardness of Approximating Minimization Problems". In: *J. ACM* 41.5 (Sept. 1994), pp. 960–981. ISSN: 0004-5411.

[12] Carsten Lund and Mihalis Yannakakis. "On the hardness of approximating minimization problems". In: *Journal of the ACM (JACM)* 41.5 (1994), pp. 960–981.

[13] Tim Menzies et al. *The PROMISE Repository of empirical software engineering data*. 2012. URL: `http://promisedata.googlecode.com`.

[14] A. Riboira and R. Abreu. "How Many Defects Need to be Fixed?" In: (TBP).

[15] Peter Zoeteweij et al. "Diagnosis of embedded software using program spectra". In: *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*. IEEE. 2007, pp. 213–220.