Mutant Density: A Measure of Fault-Sensitive Complexity

Ali Parsai
ali.parsai@uantwerpen.be
University of Antwerp and FlandersMake
Antwerp, Belgium

Serge Demeyer
serge.demeyer@uantwerpen.be
University of Antwerp and FlandersMake
Antwerp, Belgium

ABSTRACT

Software code complexity is a well-studied property to determine software component health. However, the existing code complexity metrics do not directly take into account the fault-proneness aspect of the code. We propose a metric called mutant density where we use mutation as a method to introduce artificial faults in code, and count the number of possible mutations per line. We show how this metric can be used to perform helpful analysis of real-life software projects.

CCS CONCEPTS

• **Software and its engineering → Software post-development issues**; **Software development techniques**.

KEYWORDS

mutant density, software health, complexity metrics, fault-proneness

ACM Reference Format:
Ali Parsai and Serge Demeyer. 2020. Mutant Density: A Measure of Fault-Sensitive Complexity. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387940.3392210

1 INTRODUCTION

The software has grown from a niche afterthought to an all-encompassing aspect of products in many industries. In addition, software products are becoming more and more complex everyday. Ensuring quality of the software is therefore becoming more important and more difficult at the same time. This makes the quality of the software an important aspect of its health [8].

The complexity of the software is often tied to its quality. In particular, software complexity metrics are used as a predictor of its health and maintainability [11, 16]. These metrics are used extensively in defect prediction literature as well (e.g. [27] and [7]), and are generally known to be an indication of fault -proneness of the code. Therefore they can be used as an indication for software developers to predict and avoid defects [29].

However, complexity metrics by themselves are not tied to different defect types. While high complexity of a software component is a sign of potential for a fault, it does not clarify what parts of code are more likely to contain the fault. Therefore, a metric that pinpoints statements more likely to be at risk of a fault is a useful tool for the developer in maintaining the software. Such a metric needs to take into account all possible ways a statement can become faulty according to a particular fault model.

Such a mechanism already exists in the field of mutation testing. Mutation testing is a process in which a fault is deliberately introduced in a software component and then the tests are executed to see whether they are able to detect the fault. The faulty version of the software in this method is called a *mutant*. The mutants are created based on fault models that aim to replicate common mistakes of the domain in which the test quality needs to be measured, hence the mutants are sensitive to the type of faults that are intended to be caught [9, 18].

In this article, we leverage the mentioned property of mutants to create a new fault-sensitive metric for complexity. For this, we count the number of mutants that can be generated for each line of code. We call this metric *mutant density*. We showcase two types of analysis by using mutant density. Through such analysis, developers can improve the quality of their software components, and consequently the health of their software.

This article is structured as follows: In Section 2 we briefly describe the background information and related work in the context of this article. In Section 3 we describe mutant density metric in detail. In Section 4 we provide few ways this metric can be utilized. Finally, in Section 5 we conclude the article and present future research directions.

2 BACKGROUND

In this section, we briefly describe the background information and related work in the context of this article.

2.1 Complexity Metrics

Complexity metrics are used to quantify the complexity of a unit of software. McCabe cyclomatic complexity is a widely-used and yet controversial metric created by McCabe in 1976 [15]. This metric measures the number of linearly independent paths through a unit of code. While its use has been advocated since 1980's in software maintenance [11], it is considered an inferior metric for this purpose in academic circles [6]. Yet, it is still used extensively in academic case studies and in practice (e.g. [3, 17]). It is known that developers tend to change their behavior in order to avoid hot spots indicated by such metrics [25]. As a result, code complexity is a useful metric in the context of the software health, and particularly software component health [16] and defect prediction [13].

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392210

2.2 Mutation Operators

A *mutation operator* is a pattern that changes part(s) of a software system in order to introduce a fault. The faulty version of the code produced in this manner is called a *mutant*. The traditional mutation operators were first reported in King et al. [12] for FORTRAN77 programming language. These operators basic language elements such as arithmetic operators, conditional statements, etc. In 1996, Offutt et al. show that a smaller set of mutation operators can produce a similarly capable test suite. This reduced set of operators remains popular in most mutation testing tools to date.

A major branch of academic research in mutation operators is focused on inventing new mutation operators to target emerging fault patterns and new programming paradigms such as targeting certain security problems [23, 28], language specific mutation operators [1, 4, 19, 20, 24], or object-oriented mutation operators [5, 14].

Artificial faults created by mutants are known to be a good replacement to real faults in software experiments [2, 10]. The mutation score also correlates well with defect density [26].

2.3 LittleDarwin

LittleDarwin is a mutation testing tool designed for Java source code. LittleDarwin has been used in several studies, and is capable of performing mutation testing on complicated software systems [19, 21]. LittleDarwin contains two distinct set of mutation operators: traditional and null-type. For more information about LittleDarwin and its structure please refer to Parsai et al. [22]. LittleDarwin is open source software, and the latest version can be found at https://littledarwin.parsai.net/.

3 MUTANT DENSITY

Mutant density is defined as the number of mutants that are generated for each line of code. Average mutant density for a source file is then defined as the sum of mutant density of each *relevant* line divided by the number of lines of code in the file. For the calculation of this metric, we only consider non-blank lines of code within methods and constructors relevant. Figure 1 shows a mutation report where mutant density is calculated. In this figure, the colored highlights show mutants, the white background means the line is relevant, and gray background means non-relevant. For example, `for(int i = 0; i < NUM_FACTS; i++)` has a mutant density of two, since `i < NUM_FACTS` can be mutated to `i >= NUM_FACTS` and `i++` can be mutated to `i--`.

It is apparent that this metric is subject to change based on the fault model that the mutation engine uses. For example, using a different set of mutation operators, more mutants can be generated from `i < NUM_FACTS`, perhaps changing the constant to zero, or changing the '<' operator to '<=' and '=='. This, however, is not a bug, but rather a feature of this metric: it allows the developer to fine-tune the fault model to measure what matters in the particular context of their project rather than rely on the generic fault models.

4 ANALYSIS

For the purposes of this article, we bring examples of the use of mutant density metric from real-life open source Java project AddThis Codec[1]. What follows is two sample analyses that we provide using this metric. We use LittleDarwin to calculate mutant density for traditional and null-type faults.

Average Density: 0.64

```
0001 public class Factorial {
0002     public static void main(String[] args) {
0003         final int NUM_FACTS = 100;
0004         for(int i = 0; i < NUM_FACTS; i++)
0005             System.out.println( i + "! is " + factorial(i) );
0006         }
0007
0008     public static int factorial(int n) {
0009          int result = 1;
0010          for(int i = 2; i <= n; i++)
0011              result *= i;
0012          return result;
0013     }
0014 }
0015
0016
```

Figure 1: An Example of the Usage of Mutant Density

Finding Fault-Prone Components

Using average mutant density metric at compilation unit level allows us to visualize the complexity of each unit with regards to the used fault model. Therefore it helps the developer in locating fault-prone components, and to improve the quality of the code.

Figure 2 shows the combined average mutant density for compilation units in AddThis Codec. In this Figure, the gray bars show the value of average mutant density based on traditional mutation operators, and black bars show the value of average mutant density based on null-type mutation operators for each compilation unit. Here, we can see that `binary.CodecBin2` and `jackson.CodecIntrospector` are the two most complex compilation units in this project. In addition, `binary.CodecBin2` is more prone to traditional faults (such as mistakes in arithmetic and conditional operators) than null-type faults. An interesting observation is the fact that `json.Codec.JSON` is not really prone to traditional faults, since it lacks those mutable structures, however, it is highly susceptible to null-type faults.

Rewriting Complex Statements

By visualizing mutant density of each line in a compilation unit, it becomes instantly apparent where the complexity lies. Using this information, a developer can quickly refactor and rewrite high-complexity statements and reduce the chance for future mistakes.

In Figure 3, two methods from AddThis Codec located in compilation unit `reflection.CodableFieldInfo` are shown. Both these methods consist of a single return line where all the logic of the method is performed. Using mutant density one can instantly see the complexity of this statement is high. As a solution, these return statements can be separated into several lines of code, and by doing so, one can increase the readability and decrease fault-proneness of these statements.

[1] https://github.com/addthis/codec

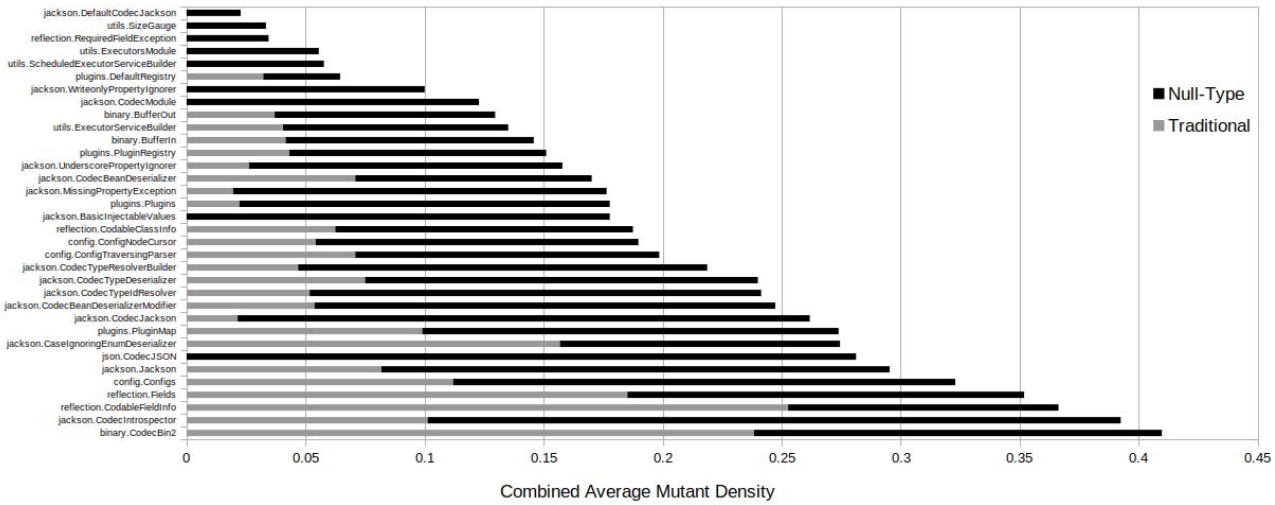

Figure 2: Combined Average Mutant Density for Compilation Units in AddThis Codec

```
0172    @Nullable public Class getMapValueClass() {
0173        return ((genTypes != null) && (genTypes.length == 2)) ? (Class) genTypes[1] : null;
0174    }
0175
0176    public boolean isCollectionArray() {
0177        return ((genArray != null) && (genArray.length == 1)) ? genArray[0] : false;
0178    }
```

Figure 3: An Example of Fault-Prone Statements in `reflection.CodableFieldInfo` to be Rewritten

5 CONCLUSION AND FUTURE WORK

Code complexity is often used as a surrogate metric for fault-proneness of software components, however, the existing code complexity metrics do not directly aim at fault-proneness aspect of the code. In this article, we proposed the use of a new metric called mutant density that uses a customizable fault model to calculate complexity of each line of code. We show how this metric can be used by developers to extract useful insight into the health of their software components.

The research into mutant density can be continued in many future directions. The relation between mutant density and defect density can show whether the previous assumptions about the quality of artificial faults holds true. The customization of the fault-model allows for several types of mutation operators to be tried out. The current set of mutation operators are optimized for increasing the quality of software tests. Therefore, it is possible that a different set of mutation operators is required to produce more accurate mutant density calculations. Finally, the effects of redundant and equivalent mutants on this metric are unknown at this time and demand further investigation.

ACKNOWLEDGMENTS

This work is sponsored by:
(a) ITEA^3 TESTOMAT Project (number 16032), sponsored by VINNOVA – Sweden's innovation agency;
(b) Flanders Make vzw, the strategic research centre for manufacturing industry.

REFERENCES

[1] Robin Abraham and Martin Erwig. 2009. Mutation Operators for Spreadsheets. *IEEE Transactions on Software Engineering* 35, 1 (jan 2009), 94–108. https://doi.org/10.1109/TSE.2008.73

[2] James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th international conference on Software engineering - ICSE ́05* (St. Louis, MO, USA) *(ICSE '05)*. ACM Press, New York, NY, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[3] V. Antinyan, M. Staron, W. Meding, P. ÃŰsterströ̈m, E. Wikstrom, J. Wranker, A. Henriksson, and J. Hansson. 2014. Identifying risky areas of software code in Agile/Lean software development: An industrial experience report. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 154–163. https://doi.org/10.1109/CSMR-WCRE.2014.6747165

[4] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. 2006. ExMAn: A Generic and Customizable Framework for Experimental Mutation Analysis. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006) (MUTATION '06)*. IEEE, Washington, DC, USA, 4. https://doi.org/10.1109/mutation.2006.5

[5] Huo Yan Chen and Su Hu. 2006. Two New Kinds of Class Level Mutants for Object-Oriented Programs. In *2006 IEEE International Conference on Systems, Man and Cybernetics*, Vol. 3. IEEE, 2173–2178. https://doi.org/10.1109/icsmc.2006.385183

[6] C. Ebert and J. Cain. 2016. Cyclomatic Complexity. *IEEE Software* 33, 6 (Nov 2016), 27–29. https://doi.org/10.1109/MS.2016.147

[7] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. 2011. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience* 41, 5 (2011), 579–606. https://doi.org/10.1002/spe.1043 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1043

[8] Sami Hyrynsalmi, Marko SeppÃd̆nen, Tiina Nokkala, Arho Suominen, and Antero JÃd̆rvi. 2015. Wealthy, Healthy and/or Happy — What does 'Ecosystem Health' Stand for?. In *Software Business*, João M. Fernandes, Ricardo J. Machado, and Krzysztof Wnuk (Eds.). Springer International Publishing, Cham, 272–287.

[9] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (sep 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[10] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654âĂŞ665. https://doi.org/10.1145/2635868.2635929

[11] D. Kafura and G. R. Reddy. 1987. The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering* SE-13, 3 (March 1987), 335–343. https://doi.org/10.1109/TSE.1987.233164

[12] Kim N. King and A. Jefferson Offutt. 1991. A fortran language system for mutation-based software testing. *Software: Practice and Experience* 21, 7 (jul 1991), 685–718. https://doi.org/10.1002/spe.4380210704

[13] G. Lajios. 2009. Software Metrics Suites for Project Landscapes. In *2009 13th European Conference on Software Maintenance and Reengineering*. 317–318. https://doi.org/10.1109/CSMR.2009.22

[14] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. 2002. Inter-class mutation operators for Java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.* IEEE Comput. Soc, 352–363. https://doi.org/10.1109/issre.2002.1173287

[15] T. J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (Dec 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[16] John Yates Monteith, John D. McGregor, and John E. Ingram. 2014. Proposed Metrics on Ecosystem Health. In *Proceedings of the 2014 ACM International Workshop on Software-Defined Ecosystems* (Vancouver, BC, Canada) *(BigSystem âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 33âĂŞ36. https://doi.org/10.1145/2609441.2609643

[17] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. 2011. An Empirical Model of Technical Debt and Interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt* (Waikiki, Honolulu, HI, USA) *(MTD âĂŹ11)*. Association for Computing Machinery, New York, NY, USA, 1âĂŞ8. https://doi.org/10.1145/1985362.1985364

[18] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2018. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* (2018). https://doi.org/10.1016/bs.adcom.2018.03.015

[19] Ali Parsai and Serge Demeyer. 2019. Do Null-Type Mutation Operators Help Prevent Null-Type Faults?. In *SOFSEM 2019: Theory and Practice of Computer Science*, Barbara Catania, Rastislav Královič, Jerzy Nawrocki, and Giovanni Pighizzini (Eds.). Springer International Publishing, Cham, 419–434. https://doi.org/10.1007/978-3-030-10801-4_33

[20] Ali Parsai, Serge Demeyer, and Seph De Busser. 2018. C++11/14 Mutation Operators Based on Common Fault Patterns. In *Testing Software and Systems*, Inmaculada Medina-Bulo, Mercedes G. Merayo, and Robert Hierons (Eds.). Springer International Publishing, Cham, 102–118. https://doi.org/10.1007/978-3-319-99927-2_9

[21] Ali Parsai, Alessandro Murgia, and Serge Demeyer. 2016. Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE ´16* (Limerick, Ireland) *(EASE '16)*. ACM Press, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/2915970.2915992

[22] Ali Parsai, Alessandro Murgia, and Serge Demeyer. 2017. LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems. In *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers*, Mehdi Dastani and Marjan Sirjani (Eds.). Springer International Publishing, 148–163. https://doi.org/10.1007/978-3-319-68972-2_10

[23] Hossain Shahriar and Mohammad Zulkernine. 2008. Mutation-Based Testing of Format String Bugs. In *2008 11th IEEE High Assurance Systems Engineering Symposium (HASE '08)*. IEEE, Washington, DC, USA, 229–238. https://doi.org/10.1109/hase.2008.8

[24] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. 2012. Mutation operators for concurrent programs in MPI. In *2012 13th Latin American Test Workshop (LATW) (LATW '12)*. IEEE, Washington, DC, USA, 1–6. https://doi.org/10.1109/latw.2012.6261240

[25] D. StÃěhl, A. Martini, and T. MÃěrtensson. 2019. Big Bangs and Small Pops: On Critical Cyclomatic Complexity and Developer Integration Behavior. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 81–90. https://doi.org/10.1109/ICSE-SEIP.2019.00017

[26] D. Tengeri, L. VidÃącs, ÃĄ. BeszÃĺdes, J. JÃąsz, G. Balogh, B. Vancsics, and T. GyimÃşthy. 2016. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 174–179. https://doi.org/10.1109/ICSTW.2016.25

[27] M. J. P. v. d. Meulen and M. A. Revilla. 2007. Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. 203–208. https://doi.org/10.1109/ISSRE.2007.12

[28] Fanping Zeng, Liangliang Mao, Zhide Chen, and Qing Cao. 2009. Mutation-Based Testing of Integer Overflow Vulnerabilities. In *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM'09)*. IEEE, Piscataway, NJ, USA, 4416–4419. https://doi.org/10.1109/wicom.2009.5302048

[29] H. Zhang, X. Zhang, and M. Gu. 2007. Predicting Defective Software Components from Code Complexity Measures. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. 93–96. https://doi.org/10.1109/PRDC.2007.28