Software Robustness: A Survey, a Theory and Prospects

Justyna Petke
Computer Science
University College London
U.K.
j.petke@ucl.ac.uk

David Clark
Computer Science
University College London
U.K.
david.clark@ucl.ac.uk

William B. Langdon
Computer Science
University College London
U.K.
w.langdon@ucl.ac.uk

ABSTRACT

If a software execution is disrupted, witnessing the execution at a later point may see evidence of the disruption or not. If not, we say the disruption failed to propagate. One name for this phenomenon is software robustness, but it appears in different contexts in software engineering with different names. Contexts include testing, security, reliability, and automated code improvement or repair. Names include coincidental correctness, correctness attraction, transient error reliability, and other. As witnessed, it is a dynamic phenomenon but any explanation with predictive power must necessarily take a static view. As a dynamic/static phenomenon it is convenient to take a statistical view of it which we do by way of information theory. We theorise that for failed disruption propagation to occur, a necessary condition is that the code region where the disruption occurs is composed or succeeded with a subsequent code region that (statically) suffers entropy loss over all executions– and the higher the entropy loss, the higher the likelihood that disruption in the first region fails to propagate to the downstream observation point. We survey different research silos that address this phenomenon and explain how the theory can be exploited in software engineering.

CCS CONCEPTS

• **Software and its engineering → Software reliability**.

ACM Reference Format:
Justyna Petke, David Clark, and William B. Langdon. 2021. Software Robustness: A Survey, a Theory and Prospects. In *Proceedings of The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXX

1 INTRODUCTION

With the ever increasing size and complexity of software, and the plethora of devices and environments in which it is run, ensuring its correct function under all conditions is an inherently difficult task. The software quality concept of *robust* software is thus of great importance. In the taxonomy of software dependability [4], robustness is defined as “dependability with respect to external faults, which characterizes a system reaction to a specific class of faults.” The classes of faults might not only relate to users providing invalid inputs, but also to physical factors. This makes robustness center stage in the fields of safety-critical systems, such as space shuttles, but also chip design, where environmental conditions, such as fluctuations of electric currents, are hard to predict.

It is thus unsurprising that robustness considerations are most often seen during the software testing stage. Beizer et al. [7] state that around 80% of more mature software’s test suites are composed of robustness tests. Shahrokni and Feldt [20] conducted a systematic literature review on software robustness. They concluded that “robustness verification and validation (V&V) is the largest focus group in software robustness phases”, with testing being the main technique used in this category. Shahrokni and Feldt also note that the second largest category concerns design solutions for robustness. Taking robustness consideration upfront during software architecture design can help avoid future failures. Of course, it is not an easy task given the number of unpredictable conditions under which software could be run. Shahrokni and Feldt’s survey shows that the most common strategy has been to use wrappers that try to prevent errors from propagation. This category contains self-healing and antifragile software that try to detect and repair errors during runtime. Alternative strategies for increasing software’s robustness propose use of more robust programming languages.

Both robustness testing and the techniques for designing robust software aim to ensure the system functions correctly in the presence of invalid inputs or stressful environmental conditions. In other words, aim that errors arising from such conditions do not propagate, so do not hinder the system’s functional behaviour. If we treat software as a black-box, from a user’s perspective, as long as the output is correct, it is of little importance how much the error might have propagated throughout the code. However, from a software tester’s perspective, the point at which the error could have been observed (i.e., the placement and observation power of the oracle) is of uttermost value. Traditionally error-handling techniques would be put into place to limit *failed error propagation* [2]. However, the rise of techniques for the improvement of software’s non-functional behaviour sheds new light on the role of robustness.

Although, historically code has been regarded as brittle [17], Langdon and Petke [14] argued it is not as fragile as previously thought. This observation stemmed from their work on genetic improvement [18], in particular, automated improvement of software’s runtime. The plastic surgery hypothesis [5], which underpins much work on automated program repair, also challenges the view of code as inherently brittle. The underlying assumption of the aforementioned automated software improvement techniques is that robustness of code can be exploited to find better software variants. Currently such techniques usually have to navigate a prohibitively

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXX

large space of software changes, which can be compared to searching for a needle in a haystack. Identification of robust code regions, where local disruption would not be propagated to the point of changing software's functionality, would be of great value.

Prediction of robust code regions could benefit both software testers aiming to find possibly critical errors that sometimes fail to propagate, and software engineers aiming to automatically improve code. Androutsopoulos et al. [2] investigated the relationship between failed error propagation and a concept from information theory, namely conditional entropy. They empirically showed strong correlations between the two, opening avenues for future work.

In what follows we present how entropy loss analysis can be used as a predictor for *failed disruption propagation*, and how the presented theory can be exploited in software engineering.

2 SOFTWARE ROBUSTNESS AND ITS FLAVOURS

Even though the concept of software robustness has been an established term for many years, the rise of automated software improvement techniques has brought new views and new terminology, which we briefly discuss in this section.

Definition 2.1 (Robustness [1]). Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Back in 2012, with the rise of pioneering work on automated program repair, Schulte et al. [19] introduced the concept of *software mutational robustness*. It is described as the fraction of random mutations to program code that leave the program's behaviour unchanged. In 2019 Harrand et al. [10] presented a conceptual replication of Schulte et al.'s study and defined *code plasticity* as the code's "intrinsic capability at being changed to another code, while keeping functional correctness, with respect to a given test suite". They state that the conceptual difference with software mutational robustness is that the latter reasons "about the ability to tolerate perturbations", while code plasticity aims to characterise "the ability of code to exist in multiple forms. It should be noted that the *plasticity* term has first been used by Barr et al. [5], in the plastic surgery hypothesis, which states that "Changes to a codebase contain snippets that already exist in the codebase at the time of the change, and these snippets can be efficiently found and exploited."

Both Schulte et al. [19] and Harrand et al. [10] introduce measures which can be viewed as proxies for software robustness, in that the higher these measures are the more robust the software is, with respect to the given test suite.

Definition 2.2 (Software Mutational Robustness and Neutral Variant Rate [10, 19]). Let $M(P) = V$ be the set of program variants generated by applying mutations from M to the program P. Let $V_c \subset V$ denote all program variants that compile and $V_T \subset V_c$ denote all program variants that pass all the tests in P's test suite T. Then software mutational robustness (SMR) and neutral variant rate (NVR) are calculated as follows:

$$SMR = \frac{|V_T|}{|V|} \qquad NVR = \frac{|V_T|}{|V_c|} \tag{1}$$

The above measures are also in line with traditional measures for software robustness, in that they are defined with respect to a given test suite.

Another term related to robustness that has been discussed in the field of automated program repair is *antifragile software*. It was inspired by the definition of antifragility by Taleb [23] which states that "a system is antifragile if it thrives and improves when facing errors" [15]. Monperrus provides examples of antifragile software [15], including self-healing systems [22], that are fault-tolerant, and those that self-inject bugs during production, such as encountered in the discipline of chaos engineering [6]. The concept of fragility has also appeared in the work on non-functional property improvement and was empirically investigated by Langdon and Petke [14]. In that work it is used as a synonym for robustness.

The above-mentioned authors of antifragile software and code plasticity terminology published a paper together on "correctness attraction", as defined below:

Definition 2.3 (Correctness Attraction [9]). An execution perturbation is a runtime change of the value of one variable in a statement or an expression. An execution perturbation has three characteristics: the time when the change occurs (e.g. at the second and the fourth iterations of a loop condition), the location in the code (e.g. on variable 'i' at line 42) and the perturbation model – what is this change according to the type of the location? (e.g. +1 on an integer value). The perturbation space for an input is composed of all possible unique perturbed executions according to a perturbation model, for that given input. Correctness attraction is the phenomenon by which the correctness of an output is not impacted by execution perturbation. Correctness attraction means that one can perturb an execution while keeping the output correct according to a perfect oracle.

Danglot et al. [9] related the concept of correctness attraction to antifragility and robustness using correctness ratio, which they define as the "percentage of correct executions over the whole perturbation space". In particular, if, in their experiments, a program point had correctness ratio of 100% it was deemed antifragile. It was called robust if the ratio was above 75%. This shows that indeed antifragility is regarded as a subset of robustness in this body of work.

The above terms either provide synonyms for robustness, or specialise it to a particular case. We will use the term "robustness" throughout this paper. We also note that there are different terms to describe code changes under which robust code would function as intended. These changes have been classified as errors (also faults or bugs), mutations, or perturbations. In this work we use the term "disruption" to cover all types of such changes.

Another term worth mentioning that relates to software robustness, and has been used in the automated program repair field, is coincidental correctness [11]. It can include failed error propagation as well as the case where, in spite of the error being exercised, there is no state infection, as it "arises when a defective program produces the correct output despite the fact that the defect within was exercised" [3]. Sometimes the two cases are distinguished using the propagation, infection, execution (PIE) framework, for example, in work by Jahangirova et al. [13].

The concept of failed error propagation also corresponds to the difference between weak and strong mutation testing [16]. In particular, a program mutation m is weakly killed by a test case t, if the execution of t on the original program and its mutant produces different program states. The disruption m, however, does not need to propagate to the output. If it does, we say that m is strongly killed. In non-functional genetic improvement [18] , weak, but not strong mutants are desired.

We will use the term "failed disruption propagation" to mean execution with infection without successful propagation [24].

3 AN ENTROPY-BASED PREDICTIVE MODEL

We first present a model of disruption propagation failure, and show how robustness can be explained by it.

Assume a deterministic, imperative program language and consider a program, P, in this language together with its Control Flow Graph $G(P)$. Assume there is a random variable, I_P, in the initial states of P. Consider the set of all executions of P, $E(P)$. Then there is a random variable $E(P)$ whose events are the executions of P with the same probability distribution as I_P. Consider the collecting semantics of P that collects all the states that may occur at each program point in $G(P)$ as a result of executions in $G(P)$. Then I_P induces a random variable in possible states *at each program point* in $G(P)$ as a result of all the executions in $E(P)$

Suppose that an observer is observing a property of a state at a program point op in $G(P)$ resulting from an execution. The observer may observe the whole state (strongest observation power) or some property of it (weaker observation power). For example, in software testing it is common for the oracle (observer) to assess the correctness of the output of a program, output being a property of the final state.

Consider that the execution is disrupted at some program point, dp, occurirng before the observation point, producing a state that is not expected for that execution. This could be caused by an undiscovered error in P's syntax, deliberate mutation of the syntax, a disruption of an execution state via lasers, gamma rays, microwaves or program instrumentation, or the exploitation of a security vulnerability. How can the observer fail to witness an unexpected state at a subsequent program point in $G(P)$, at op? Two conditions have to be met:

(1) There is a program fragment, F, where F is the *chop* of source dp and sink op [12], and $F(s) = t$, $F(s') = t'$, where s is the expected state at dp, s' is the disruption state at dp and t and t' respectively are the resulting states at the observation point, op, in each case.
(2) The observer observes a property of a state and $\alpha(t) = \alpha(t')$, where $\alpha : \Sigma \to A$ is the function that extracts the property value from the observed state, Σ is the set of all possible states and A the set of possible values of the observed property.

An example of the second condition is a test oracle that checks the program output for each test. Perhaps output is a truth value evaluated using the final state. Then function α is the evaluation function mapping the set of final states to Boolean. Making α explicit explains two papers with very different results. Assi et al. examined coincidental correctness in the Defects4J benchmark suite of Java programs [3]. Each program comes with at least two versions, with and without real faults, as well as test cases. Using the test cases they found coincidental correctness to be highly prevalent in Defects4J. On the other hand, Jahangirova et al. examined the same benchmark, but generated their own tests and used the final state as an oracle [13]. By contrast, they found negligible failed error propagation (Assi et al.'s strong coincidental correctness) at unit test level although this increased considerably at system level.

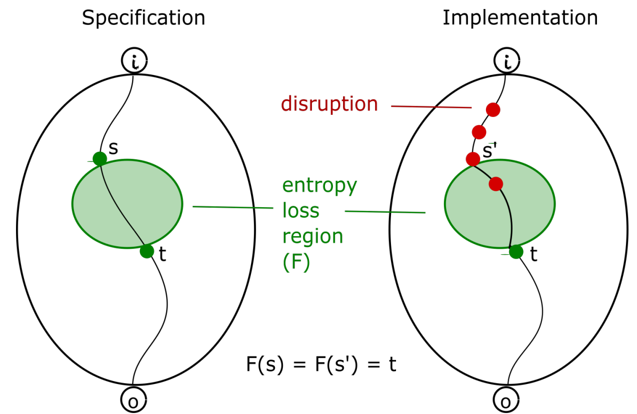

Figure 1: Simplified FDP model. Here $F(s) = F(s')$ and α is identity.

The first condition is problematic as it is completely dependent on the exact disruption state and its relation to the semantics of F. Ideally, software engineering would like to be able to predict when code regions such as F have a high likelihood of causing failed disruption propagation (FDP). Note that, for a given execution observation to fail, function $\alpha \circ F$ must collide the two different inputs at output. The diagram in Figure 1 illustrates the concept in the scenario where $F(s) = F(s')$ and α is identity, i.e. the observer observes the whole state.

Previous work has suggested function entropy loss as a good predictor of the likelihood of range collisions, better than simple input/output size ratios [8]. Experimental evidence has borne this out. Androutsopoulos et al. have experimentally demonstrated strong rank correlations (> 0.95) between entropy loss measures and likelihood of failed error propagation [2], while Assi et al. have provided indirect evidence, given the intuition that larger code and longer execution paths correlate with more entropy loss. They noted that test cases were more likely to suffer coincidental correctness if the execution paths were longer and "comprised a higher number of conditional, modulo, multiplication, division, and invocation statements" [3].

Entropy, $\mathcal{H}$, is a statistic of a probability distribution that measures how disordered or, alternatively, how regular the distribution is. It is defined as the expected value of the log of the inverse probability p across the support [21]. For a random variable X:

$$\mathcal{H}(X) = -\sum_{x \in X} p(x)\, log\, p(x)$$

Entropy loss over all executions of a computation can be measured as the conditional entropy of the inputs given the outputs.

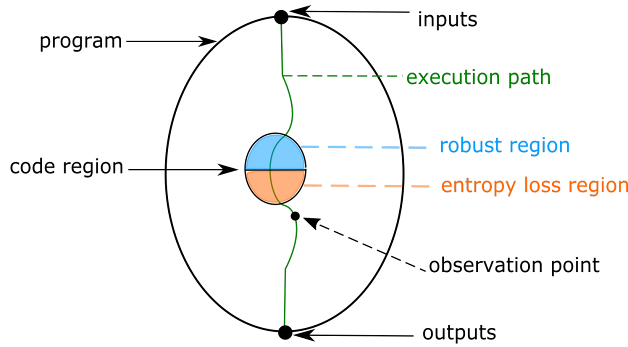

Figure 2: Robust Code Region (blue shaded area)

Below we use the Chain Rule as a proxy definition of conditional entropy.

Given random variables in inputs, In, and outputs, O, the conditional entropy of In given O, $\mathcal{H}(In \mid O)$, is given via the Chain Rule as $\mathcal{H}(In \mid O) = \mathcal{H}(In, O) - \mathcal{H}(O)$ where $\mathcal{H}(In, O)$ is the entropy of the joint random variable $\langle In, O \rangle$. When O is a function of In the calculation is simpler with $\mathcal{H}(In \mid O) = \mathcal{H}(In) - \mathcal{H}(O)$ but this only holds in deterministic languages.

Consider this example of failed disruption propagation (FDP):

```
x = 3 * x;    // error, should be x = x + 2;
if (x > 0 )
    x = x % 4; // F ( entropy loss region)
else x = x;
```

Assuming we have the following two test cases: $t_1 : x = 3$ and $t_2 = -5$, if we only run t_1, we will miss the error, as the output of the original and modified code will be 1. However, if we run t_2, x will equal to -15 , and not the intended -3.

It is the loss of entropy in the function $\alpha \circ F$ that is a good predictor of the probability of FDP at a given program point. How does this model of FDP connect to software robustness? Our intuition is that rather than programs being robust, it is regions of programs that are robust.

4 PROSPECTS

We expect that an entropy loss region affects not just a single program point but a *robust region* of program points (code). This region is created by being composed with a following region which has significant entropy loss as in Figure 2. The entropy loss region has a masking or occluding effect on the robust region for an observer of a later program point. Disruptions have a lower chance of being detected, the code is more plastic, less fragile, and more resistant to tampering, as is the state and the program data in this region.

We can predict robust regions by identifying entropy loss regions via estimates produced with dynamic or static analyses. Prediction of robust code regions could benefit both software testers aiming to find possibly critical errors that sometimes fail to propagate, and software engineers aiming to automatically improve code. Further away, on the horizon, is investigating the relationship between program input-output equivalence and robustness, offering the possibility of controlling robustness in programs, both increasing and decreasing it so that, for example, code robustness can be increased after testing and prior to deployment.

5 CONCLUSIONS

We have discussed software robustness, making a case that robustness is a probabilistic, non-functional property of programs that can be explained by a model of failed disruption propagation (FDP). We have argued that this model unites differently named software phenomena in the software engineering research literature, such as code plasticity and correctness attraction. Then we have argued, on the basis of experimental evidence, that information theory provides a suitable abstraction, via entropy loss, that enables prediction of FDP. We have hypothesised that program robustness is actually a property of program *regions* that are "guarded" from the observer/oracle by high entropy loss regions. In future work we intend to develop these ideas further and apply these to diverse areas of software engineering.

6 ACKNOWLEDGEMENTS

This research was supported by the FaceBook funded ELVEN project and EPSRC grants EP/P023991/1, EP/P005888/1.

REFERENCES

[1] 2017. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. (2017). https://doi.org/10.1109/IEEESTD.2017.8016712
[2] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman. 2014. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *ICSE*. ACM, 573–583.
[3] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. 2019. Coincidental correctness in the Defects4J benchmark. *Softw. Test. Verific. Reliab.* 29, 3 (2019).
[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.* 1, 1 (2004), 11–33.
[5] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *SIGSOFT FSE*. ACM, 306–317.
[6] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2017. Chaos Engineering. *CoRR* abs/1702.05843 (2017).
[7] Boris Beizer. 1995. *Black-box testing - techniques for functional testing of software and systems.* Wiley.
[8] David Clark and Robert M. Hierons. 2012. Squeeziness: An information theoretic measure for avoiding fault masking. *Inf. Process. Lett.* 112, 8-9 (2012), 335–340.
[9] Benjamin Danglot, Philippe Preux, Benoit Baudry, and Martin Monperrus. 2018. Correctness attraction: a study of stability of software behavior under runtime perturbation. *Empir. Softw. Eng.* 23, 4 (2018), 2086–2119.
[10] Nicolas Harrand, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monperrus, and Benoit Baudry. 2019. A journey among Java neutral program variants. *Genet. Program. Evolvable Mach.* 20, 4 (2019), 531–580.
[11] Robert M. Hierons. 2006. Avoiding coincidental correctness in boundary value analysis. *ACM Trans. Softw. Eng. Methodol.* 15, 3 (2006), 227–241.
[12] Daniel Jackson and Eugene J. Rollins. 1994. Chopping: A Generalization of Slicing. In *Proc. of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*.
[13] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2020. An Empirical Study on Failed Error Propagation in Java Programs with Real Faults. *CoRR* abs/2011.10787 (2020).
[14] William B. Langdon and Justyna Petke. 2017. Software is Not Fragile. In *First Complex Systems Digital Campus World E-Conference.* Springer.
[15] Martin Monperrus. 2017. Principles of Antifragile Software. In *Programming*. ACM, 32:1–32:4.
[16] A. Jefferson Offutt and Stephen D. Lee. 1991. How Strong is Weak Mutation?. In *Symposium on Testing, Analysis, and Verification*. ACM, 200–213.
[17] Gordon C. Osbourn. 1999. Towards an Approach to Overcome Software Brittleness. https://doi.org/10.2172/15150
[18] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Computation* 22, 3 (2018), 415–432.

[19] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genet. Program. Evolvable Mach.* 15, 3 (2014), 281–312.

[20] Ali Shahrokni and Robert Feldt. 2013. A systematic review of software robustness. *Inf. Softw. Technol.* 55, 1 (2013), 1–17.

[21] Claude E. Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal* 27, 4 (1948), 623–656.

[22] Michael E. Shin. 2005. Self-healing components in robust software architecture for concurrent and distributed systems. *Sci. Comput. Program.* 57, 1 (2005), 27–44.

[23] Nassim Nicholas Taleb. 2012. *Antifragile.* Random House, New York.

[24] Jeffrey M. Voas. 1992. PIE: A Dynamic Failure-Based Technique. *IEEE Trans. Software Eng.* 18, 8 (1992), 717–727.