SOFTWARE RELIABILITY – DOES IT EXIST?

Dmitry Maevsky

Odessa National Polytechnic University
e-mail: Dmitry.A.Maevsky@opu.ua
Ukraine

Abstract: Software reliability has been studied for over 60 years. During this time, a large number of different software reliability models have been created. But none of them can accurately describe the process of detecting defects in software systems. The models do not allow to take into account secondary defects that are introduced into the program when correcting found defects. The article shows that the reliability of software as a scientific direction has led to a dead end. The reason for this is the wrong choice of the basis of the software reliability theory - the assumption that the process of identifying defects is a random process. Moreover, it has been shown that the term "reliability" cannot be applied to software. It is shown that the replacement of the conceptual basis by the theory of non-equilibrium processes allows to eliminate all contradictions in the software reliability theory.

Key words: software reliability, reliability models, secondary defects, random processes, non-equilibrium processes.

1. INTRODUCTION

The modern theory of the reliability of hardware began to develop in the 1950s, when the United States Air Force organized a group to study the problems of the reliability of electronic equipment. The probability theory and the theory of random processes became the mathematical apparatus of the theory of reliability. And this is logical, since hardware failures do have a random nature. The assumption of the random nature of failures is the conceptual basis of the entire modern theory of reliability. On its basis, mathematical models have been developed that describe well the distribution of equipment failures over time. In the same period, concepts such as the probability of failure, the failure rate, and the mean time between failures were introduced. Without these concepts, the modern theory of reliability cannot exist.

In the early 60s of the last century, a new direction began to develop intensively - the development and creation of software. And then it became clear that for software there is a phenomenon that is very similar to a failure - when a program, due to programmer’s mistakes (so-called “defects”), sometimes worked incorrectly.

It was at this time that the need arose for the emergence of the theory software reliability in the framework of the general and well-developed science of reliability.

Probably, the article by R. Barlow and E. Scheuer “Reliability Growth during a Development Testing Program”, which was published in 1966 [1], can be considered the historically first publication on software reliability. The term “software reliability” is not yet explicitly stated in it, but the first feature of software systems has already been noted — their reliability increases over time.

Over the period from 1966 to 2019, more than five thousand scientists contributed to the theory of software reliability. During this period, they have published more than 2000 articles, books and monographs. There was a huge variety of models of reliability. Some authors, for example, Michael Lyu [2], argue that the total number of such models today exceeds 60.

All this time, the science of software reliability has evolved as a kind of general theory of reliability and uses its conceptual basis - the theory of probability. All reliability models are built on the assumption of the random nature of its failures (detection of software defects). But none of these models can describe the process of detecting defects in programs with acceptable accuracy. But none of these models can take into account the so-called "secondary defects", which are introduced into the program when correcting defects that are detected. And finally, there is no single universal model of reliability for software.

All this gives a good reason to doubt that the reliability of software can be described by a mathematical apparatus, which is used for the reliability of technical systems. According to the apt expression of the famous expert in the theory of reliability, Igor Ushakov [3], “attempts to put hardware” shoes on “software legs” are absolutely wrong.

In this article we will try to understand why hardware reliability shoes do not fit on software legs. To do this, we will need to rethink and question the basics of the theory of software reliability. Moreover, we will have to question whether the term “reliability” can be used for software systems. So, let's start with the basics.

2. WHY SOFTWARE RELIABILITY MODELS DO NOT WORK CORRECTLY?

As already mentioned, the modern theory of software reliability has about sixty different models. But all these models are designed to describe the same physical process - namely, the process of identifying defects in software. Such a large number of models cannot be a sign that the theory of software reliability is well developed. Rather, this fact indicates that we understand something wrong in the theory of reliability.

But first of all, it is necessary to answer the question: why was such a large number of models created at all? The answer is very simple. The fact is that the same model may well describe the process of detecting defects in one software system, but at the same time produce incorrect results in another. The reason for this is not

software. What is the difference, for example, between the accounting software system and the program for calculating the trajectory of a spacecraft? Both systems consist of the same elements - operators. There is a very limited number of types of such operators. For example, an assignment statement, a conditional statement, or a loop statement. The difference between these systems is only in the order of the operators! Obviously, the cause should be sought in the reliability models themselves. That is why other reliability models were created to describe the process of detecting defects in other software systems. Let's try to figure out what exactly is wrong with the models?

Any model of any process must have some assumptions. If these assumptions are fulfilled and the model is made correctly, then the results of its work will not differ much from the experiment. But if the assumptions are not fulfilled, then the model is likely to give incorrect results.

Different software reliability models have different assumptions. However, there are several basic assumptions that are common to all models without exception. These are the so-called basic assumptions:

1. Faults are removed immediately.
2. New faults are not brought in the process of debugging.
3. All faults are occurred independently from each other.

Let's analyze whether these assumptions are really made for software systems.

1. The assumption that defects are removed immediately after their detection. As the review of practical methods of software testing [4, 5, 6, 7] shows, the testing process is not organized in the right way. At the beginning, a team of testers is testing software and compiling a list of errors found. This list indicates that on a specific set of input data the program does not work as it is necessary according to the specifications. And only after such a list has been compiled, programmers begin to analyze the program code and identify defects that led to the indicated functioning errors. Thus, the assumption that defects are eliminated immediately after their detection does not correspond to actual testing practice.

2. The assumption that new (secondary) defects are not introduced during the removing of detected defects. In fact, if you follow this assumption, you need to assume that the correction of defects is performed by ideal programmers who never make mistakes. In reality, everyone can be wrong. When fixing defects, programmers often need to rewrite or modify large sections of the program code. The likelihood that during such changes will be made new errors, increases with the size of the code. And this probability is never zero. Thus, due to the introduction of secondary defects into the program, a gap arises between the real system and its model. In a real program, the number of defects is always greater than its model. And this gap widens with time. By increasing the number of defects in a real program, defects in it will be detected more often than the ideal model. Thus, a model that does not take into account secondary defects will never correspond to a real program.

3. The assumption that defects arise independently of one another. Unfortunately, this assumption is also not always true. In actual testing conditions, there are cases when the detection and elimination of one defect leads to the possibility of detecting many other defects [8]. For example, suppose that a defect that existed in a program led to the fact that some part of the program code could never get control. Such a defect, for example, may be the incorrect recording of a condition in a conditional statement. As a result of this, defects that existed in this part of the code could never be detected. After the defect is fixed, the possibility of performing this section appears and, accordingly, the possibility of detecting a series of new defects.

And of course, this assumption is completely dissatisfied with secondary defects. Indeed, because these defects arise only as a result of the discovery of other defects. Thus, secondary defects are dependent on the detection of other defects.

In addition to the general assumptions discussed above, each software reliability model has assumptions about the distribution law of a random variable that is predicted. For most models, this value is the time interval between the detection of defects. If the process of detecting defects is random, then the time interval between the detection of two consecutive defects will be random. As shown in [9], most software reliability models assume that the time interval between the detection of defects obeys the Poisson distribution, Normal or Exponential distribution.

It should be noted that in all reliability models, the assumption of the distribution law is simply postulated, but the reasons for this are not explained in any way [10]. At the same time, it remains unclear how the distribution law is related to the natural and measurable characteristics of software – the environment and programming language, the amount of program code, the subject area, and the development methods. The reason for this connection also remains unclear. It is also not clear what time distribution law between the identification of defects should be used for new subject areas, such as, for example, IoT [11] or hardware-software complexes [12].

Nevertheless, it is of great interest to investigate whether the time intervals between the failures of real software systems obey these distribution laws. Such a study was carried out in [13]. For the study, we took real time series of testing of 321 real software systems, presented on the Internet resource GitHub [14]. Time series are diverse in terms of software languages (mobile systems, WEB, database, business and other). The minimum number of intervals is 51, and the maximum is 11799.

To test hypotheses, an Internet resource [15] was used. This resource makes it possible to verify statistical hypotheses about the belonging of a number of values of a random variable for four types of distributions: Normal, Poisson distribution, Binomial distribution, and Exponential distribution. The check is performed with a level of statistical significance of 0.05. The results of testing the distribution hypotheses are given in Table 1, which is borrowed from [13]:

Table 1.

Distribution	***Satisfy the demands***	
	Num. of 321	***Percent***
Poison	20	6,23
Normal	4	1,25
Exponential	29	9,03
Total	53	16,5

As can be seen from this table, only about 17% of all investigated software systems obey any distribution law. The remaining 83% do not correspond to any of the assumptions regarding the distribution law.

Thus, the basic assumptions that are used by all software reliability models are not actually performed or are performed very rarely. Therefore, we cannot demand from these models a high accuracy of conformity with the actual defect detection processes.

But there is another very significant circumstance that leads to the fact that all models of software reliability are doomed to failure. The fact is that absolutely all models imply that the process of detecting defects in software is a random process. That is why the mathematical apparatus of all models is a theory of probability. Let's see if the process of identifying defects is random?

3. ABOUT RANDOMNESS OF THE DEFECTS DETECTION

The traditional theory of reliability is based on the theory of random processes. Its mathematical apparatus is probability theory. The basis of the theory of reliability is the concept of "failure". Refusal is a malfunction of the object, in which the system or element ceases to perform its functions in whole or in part [16]. Failure is considered to be a random event, that is, an event that, when all conditions are repeated, can either occur or not [17]. For the theory of reliability of technical systems, this is absolutely true. In technical systems, failures are really random.

By the time the first complex programs appeared, the theory of reliability was well developed conceptually and mathematically. Therefore, when the first programs suddenly started to fully perform their functions, this phenomenon was called refusal by inertia and assigned all the attributes from the traditional theory of reliability to it. Failures of software systems began to be considered random and were characterized by the probability of failures, failure rates, average time between failures and so on. In fact, the theory of reliability of technical systems was mechanically transferred to the processes that occur in software systems. The difficulties of the modern theory of software reliability are precisely the result of this mechanical transfer.

At first glance, software failures do seem like random events. But let's see why programs suddenly stop working correctly, why failures occur in them. Failures of

programs are associated with the presence of programming errors in their program code. These errors in the theory of software reliability are called defects. Defects can be associated with mechanical errors when writing software code or with logical errors in this code. The introduction of a defect in the program code is indeed a random event. But is the accidental occurrence of a software failure due to a defect? Let's try to find out.

As already mentioned, a characteristic feature of a random event is that if the same conditions are repeated, the event may or may not occur. For example, a shotgun was fired from a gun, which was firmly fixed, and it hit the target. But when repeating a shot from the same gun with the same ammunition, re-hitting the target may not happen. Hitting the target under the same external conditions is a random event.

Such external conditions for the program are its source data. And under the source data we will understand not only any set of digits at the program input. The source data in the broad sense of the word can also include the actions that the user performs before and after entering this data. If any source data set caused a software failure, then such a failure will be repeated each time the data is repeated. This is explained by the fact that the processing of the specified data set resulted in the computing process hitting the section of the program code that contains the defect. And with the repetition of this data, we will each time go to the same area with the same defect. That is, the re-occurrence of a program failure will be inevitable. It can be said that the phenomenon of software system failure is not the result of the internal properties of the system itself, but mostly depends on the data that the program processes. Therefore, the process of occurrence of failures in software cannot be considered as a random process. And the phenomenon of software failure is not an accidental phenomenon. Consequently, probability theory cannot be used to describe the process of detecting defects in software. This explains the fact that software reliability models work very poorly. After all, they are built on the assumption of the randomness of software failures.

Thus, the theory of hardware reliability cannot be applied to software. That is why Igor Ushakov had the right to talk about failing to put “hardware reliability shoes” on “software legs”. That is why to describe the process of detecting defects in software, it is necessary to create a fundamentally different theory with a fundamentally different conceptual basis.

Before creating this theory, you need to figure out whether it can be called "theory of software reliability." Can the concept of "reliability" be associated with the concept of "software"? Here we come to the question that was put in the title of this article. Is there any software reliability?

4. WHAT IS “SOFTWARE RELIABILITY”?

The term "reliability" is traditionally defined as «The ability of an apparatus, machine, or system to consistently perform its intended or required function or

mission, on demand and without degradation or failure» [18]. Despite the universality of this definition, it is understandable. Let's look at the definition of software separately.

The international standard ISO/IEC 2382-1 in the section “01.01.08. Software” indicates that the software is “Any part of Programs, Procedures, rules and associated documentation of an Information processing system”. Thus, the reliability of the software – means “the reliability … of programs, procedures, rules and documentation”. And here already many questions arise.

We start with the last part of this definition. Can the term “reliability” be used for rules and documentation? If this is possible, then how are basic reliability indicators defined for rules or documentation? What is, for example, “rejection rules” or “rejection of documentation”? How to determine the failure rate of documentation, how to calculate the average time between failures for the rule? It is clear that all these questions have no answers. Because to use the concept of "reliability" for rules or, especially, documentation, is meaningless. Thus, to apply the concept of "reliability" for the last part of the definition is incorrect. Now consider the first part - the reliability of programs and procedures.

The same standard [18] in the section “01.05.01. Program; Computer program” defines program as “Syntactic unit that conforms to the rules of a particular Programming language and that is composed of Declarations and Statements or Instructions needed to solve a certain function, task, or problem”. Perform an analysis of this definition. Is a program really a syntactic block built according to the rules of a certain programming language?

Suppose that a certain programmer wrote a program in the “C” language in strict accordance with the syntactic rules of this language. Looking at this program, you can really see the syntax block with the operators of this language. So far, everything corresponds to the definition of the program.

However, this program cannot be executed by the computer directly. To do this, it must be converted into machine codes that are understandable to the central processor and can be executed by it. To do this, the program must be compiled and converted into machine codes. But after compiling, we get a completely different syntax block, which is built according to completely different rules of a completely different language!

Formally, according to the definition, we are dealing with another program that corresponds to a completely different language. However, on the other hand, from the point of view of the main functions of this software - computer hardware management - the program has not changed. Only the form of its presentation has changed, nothing more, and the content of the program has remained the same. But then it turns out that a program is not just a set of rules of any programming language! After all, the program has not changed, only the rules of the language have changed.

Therefore, it can be reasonably argued that a program is nothing more than information that controls a computing device. In our example, the information has not changed, only its coding rules have changed. But if a program is information,

then how can the concept of “reliability” be applied to the concept of “information”? What is “information reliability”? What is "information failure"? These questions have no answer most likely because no one has ever asked such questions because of their meaninglessness.

Thus, the term “software reliability” does not make sense at all. To the program, as to the information, apply another term - trustworthiness. And we should talk about the theory of software trustworthiness. And this will be a quite different theory, which will be built on quite different principles.

5. NEW PRINCIPLES OF SOFTWARE RELIABILITY. TRUSTWORTHINESS THEORY

A new theory, which describes the process of detecting defects in software, is shown in [20] and called the "Theory of Software Systems Dynamics". In this theory, software is viewed as an open system that interacts with its environment. As a result of this interaction, the number of software defects changes over time. As a rule, this quantity decreases.

First, consider what can be considered an environment (environment) for a software system. Oddly enough, if we pay attention to the process of identifying defects, then neither the computer nor the user can act as such an environment.

First of all, let us try to answer the question of how we generally know that the program is working incorrectly, that during the operation of the program we are faced with an internal defect. To distinguish the correct result of the program from the wrong, it is necessary to determine the standard of correctness. It is often said that the specification of a program or the terms of reference for its development can serve as such a reference. At first glance, that is exactly what it is. But this is only at first glance.

Firstly, the technical documentation of the program itself may contain errors. And then it turns out that a program that works in full accordance with its documentation, gives the wrong result from the point of view of the end user.

Secondly, the technical documentation of the program is not created from scratch. The programming task arises before the documentation is compiled. This task is the subject area, the processes in which the program should implement (model, execute, track). Technical documentation is only a reflection of this subject area. Therefore, such documentation is not primary. Primary in relation to the program is the subject area of this program. Here it does not matter how the subject area is defined. It is only important that this subject area exists independently of the program. And any action of this program or any result of its work can be checked for “right” or “wrong” by comparison with the subject area.

This view may seem strange. After all, the subject area is not a material concept, even if it is described in hundreds of volumes of documentation. But let us recall the conclusion we reached in the previous section. After all, the program is also not material. A program is information that controls the operation of a computer. And

the information is not material. Thus, comparing the program and its subject area, we compare objects of the same nature. Therefore, the environment for the program is the subject area of this program. And the result of the interaction of the program and the subject area is the process of identifying defects in this program.

Thus, a program is an open system that interacts with its subject area. The result of this interaction is the conclusion that the program is working correctly or incorrectly. As a result of the interaction, the process of identifying and eliminating defects occurs.

What is the driving force behind this process? Immediately there is an incorrect answer that the driving force is the user (or programmer, or tester), which identifies and eliminates defects. But it is not. After all, the user (programmer, tester) is always there, and the process of identifying defects may or may not be. Moreover, depending on the number of defects, the intensity of this process may be different. Indeed, if absolutely all program operators are defective, then each time a user accesses the program, he will detect them. The intensity of the process of identifying defects will be maximum. If there are fewer defects, then the intensity of the process of their detection decreases. And finally, if there are no defects in the program at all, that is, if it works in full compliance with the requirements of its subject area, the process of identifying defects is terminated.

Thus, the driving force behind the process of identifying defects is the lack of balance between the program and the subject area. In the program, defects are present, but in the subject area they are not by definition. Thus, the process of identifying defects in the program can be viewed as a non-equilibrium process. What then is the role of the user? No matter how offensive it is, the user in this process plays the role of the interface between the program and its environment.

That is why the basis of the theory of the dynamics of software systems is the theory of nonequilibrium processes. In any non-equilibrium open system, transfer phenomena occur. Depending on the nature of the system, spatial transfer of electric charge, matter, momentum, energy, entropy, or any other physical quantity may occur in it. The general theory of transport that can be applied to any system is given by the thermodynamics of nonequilibrium processes, which was created by Lars Onzager in 1931 [21].

According to this theory, in any non-equilibrium system, transport flows arise, aimed at establishing equilibrium in the system. In the theory of the dynamics of software systems, two streams of defects are considered. The first stream is directed from the software system and forms a stream of detected defects. The second stream is sent to the software system. The defects in this flow correspond to the secondary defects mentioned above. Thus, in the theory of the dynamics of software systems, in contrast to the traditional theory of software reliability, secondary defects are present and taken into account initially.

In more detail with the basic equations of the theory of the dynamics of software systems and their solution can be found in the article [20]. There are also the results of verification of this theory. These results show that, based on the theory of software

systems dynamics, a unified theory of software trustworthiness can be created, which with high accuracy simulates the process of detecting defects in any program.

4. CONCLUSION

This article is about how confusion in terms can lead to a dead end a whole scientific field. In this article, the author does not pretend to state the truth in the last resort. This article is debatable and is an invitation to a thoughtful dialogue about the causes of serious problems in the theory, which continues to be called the theory of software reliability by inertia. A program is information. And the term "reliability" cannot be applied to information.

Of course, the author understands that the term "software reliability" is well-established. After sixty years of existence of this term, thousands of scientists and hundreds of thousands of programmers are unlikely to use any other term to describe the patterns of detection of defects in programs. So be it. However, it should be remembered that the meaning of this term for software systems is completely different than for technical systems.

New theory of software reliability only at the beginning of its development. Since its inception in 2013, only 6 years have passed. This is ten times less than the time of the formation of the traditional theory. And the author wants her to develop with the help of joint efforts and joint new ideas of a large number of caring researchers.

REFERENCES

[1] Barlow, R., Scheuer, E.: Reliability Growth during a Development Testing Program. *Technometrics*, 8(1), 1966, pp.53-60.

[2] Lyu, M. R. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, U.S.A. and IEEE Computer Society Press, Los Alamitos, California, U.S.A, 1996.

[3] I. Ushakov. Reliability theory: history & current state in bibliographies, *RT&A*, 0124 (vol. 1), 2012, pp. 8-35.

[4] S. Kawaguchi. Trial of Organizing Software Test Strategy via Software Test Perspectives, *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, Cleveland, OH, 2014, pp. 360-360. doi: 10.1109/ICSTW.2014.42

[5] J. Kasurinen. Elaborating Software Test Processes and Strategies, *2010 Third International Conference on Software Testing, Verification and Validation*, Paris, 2010, pp. 355-358. doi: 10.1109/ICST.2010.25

[6] D. Towey and T. Y. Chen. Teaching software testing skills: Metamorphic testing as vehicle for creativity and effectiveness in software testing, *2015 IEEE*

International Conference on Teaching, Assessment, and Learning for Engineering (TALE), Zhuhai, 2015, pp. 161-162. doi: 10.1109/TALE.2015.7386036

[7] V. Garousi, M. Felderer and T. Hacaloğlu. What We Know about Software Test Maturity and Test Process Improvement, *IEEE Software*, 1 (vol. 35), Jan-Feb 2018, pp. 84-92, doi: 10.1109/MS.2017.4541043

[8] B. Littlewood, P. T. Popov, L. Strigini and N. Shryane. Modeling the effects of combining diverse software fault detection techniques, *IEEE Transactions on Software Engineering*, 12 (vol. 26), Dec 2000, pp. 1157-1167, doi: 10.1109/32.888629

[9] R. Jiang. Required Characteristics for Software Reliability Growth Models, *2009 WRI World Congress on Software Engineering*, Xiamen, 2009, pp. 228-232. doi: 10.1109/WCSE.2009.157

[10] V. Nagaraju, "Software Reliability Assessment: Modeling and Algorithms. *2018 IEEE International Symposium on Software Reliability Engineering Workshops* (ISSREW), Memphis, TN, 2018, pp. 166-169. doi: 10.1109/ISSREW.2018.000-4

[11] Y. Kondratenko, G. Kondratenko and I. Sidenko. Multi-criteria decision making for selecting a rational IoT platform, *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 2018. Available: 10.1109/dessert.2018.8409117

[12] A. Drozd, M. Kuznietsov, S. Antoshchuk, A. Martynyuk, M. Drozd and J. Sulima. Evolution of a Problem of the Hidden Faults in the Digital Components of Safety-Related Systems, *2018 IEEE East-West Design & Test Symposium (EWDTS)*, 2018. Available: 10.1109/ewdts.2018.8524806.

[13] D. Maevsky, D. Stetsyuk, E. Maevskaya and B. Stetsyuk. Probabilistic assumptions of software reliability growth models, *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 2018. Available: 10.1109/dessert.2018.8409163.

[14] *Build software better, together*, GitHub, 2019. [Online]. Available: https://github.com. [Accessed: 03- Jul- 2019].

[15] *Verification of the hypothesis about the type of distribution*, [Online] Available: https://math.semestr.ru/group/hypothesis-testing.php. Accessed: [12- Jan- 2019].

[16] M. Todinov, *Reliability and risk models*, 2nd ed. Wiley, 2015

[17] L. Blanco Castañeda, V. Arunachalam and S. Dharmaraja, *Introduction to Probability and Stochastic Processes with Applications*. Somerset: Wiley, 2014.

[18] P. O'Connor and A. Kleyner, *Practical reliability engineering*, 5th ed. Wiley, 2012.

[19] *ISO/IEC 2382-1:1993. Information Technology*, Iso.org, 2019. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-1:ed-3:v1:en. [Accessed: 04- Jul- 2019].

[20] D. Maevsky. A New Approach to Software Reliability, *Lecture Notes in Computer Science*, pp. 156-168, 2013. Available: 10.1007/978-3-642-40894-6_13 [Accessed 4 July 2019].

[21] L. Onsager. Reciprocal Relations in Irreversible Processes. I., *Phys. Rev.*, 4 (vol. 37), 1931, pp. 405-426 [Accessed 4 July 2019].

Information about the author:

Dmitry Maevsky - Doctor of Technical Sciences, Professor of the Odessa National Polytechnic University. His research interests are software reliability, the theory of nonequilibrium processes, transients in electrical systems. The author of the new concept of the theory of software reliability - the theory of the dynamics of software systems.

Manuscript received on 05 July 2019