

An approach to assessing the reliability of software systems based on a graph model of method dependence

Valerii Krutko, Iryna Spivak and Svitlana Krepych

West Ukrainian National University, 11 Lvivska Str., Ternopil, 46009, Ukraine

Abstract

The paper deals with one of the main software development tasks, namely, the research of one of its main quality criteria - reliability. The analysis of recent studies of the subject has shown that architecture-based software reliability assessment methods are more informative and adequate. However, their accuracy in many cases will depend on the complexity of the software architecture. The authors of the paper propose an approach to assessing the reliability of software systems based on a graph model of method dependency. This approach includes elements of structural analysis of the source code of the system under study in order to build method dependency graph to assess its reliability based on stochastic reliability indicators of each method. The overall reliability calculation takes into account the complexity of the software system. Based on this approach, an intellectualized system for assessing the reliability of software systems with different architectures has been developed and tested on the example of two systems of different complexity.

Keywords

software system, reliability, evaluation, stochastic indicators, graph model, probability

1. Introduction

In today's world, software has become an integral part of many areas of our daily lives, from ensuring the efficient operation of businesses to creating comfort for individuals. The development market is also rapidly changing and adapting to meet the growing demand for new software. And the key aspects of these changes are, first of all, the desire to increase the speed of program development and reduce the final price of the software product.

It is well known that high quality software is an integral part of a successful product. However, even with a fairly slow development pace, developers often make mistakes that lead to serious problems, affecting security, reliability, and user satisfaction, not to mention development in a short time. That is why ensuring the high quality of a software product is one of the main tasks that must be solved at the development stage [1].


CS&SE@SW 2023: 6th Workshop for Young Scientists in Computer Science & Software Engineering, February 2, 2024, Kryvyi Rih, Ukraine

✉ jave123@ukr.net (V. Krutko); spivak.iruna@gmail.com (I. Spivak); msya220189@gmail.com (S. Krepych)

🌐 <https://www.wunu.edu.ua/educational-subdivisions/faculty/fkit/department-kn-fkit/staff-kn-fkit/7027-spivak-iryna-yaroslavivna.html> (I. Spivak); <https://www.wunu.edu.ua/educational-subdivisions/faculty/fkit/department-kn-fkit/staff-kn-fkit/8416-krepych-svitlana-yaroslavivna.html> (S. Krepych)

🆔 0000-0003-4831-0780 (I. Spivak); 0000-0001-7700-8367 (S. Krepych)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

From the point of view of reducing the cost of development and shortening its time, it is important to identify mistakes as early as possible since in iterative development, the complexity, and as a result the cost of mistake fixing, increases over time, and the probability of fixing it correctly (without introducing new mistakes) decreases [2]. On the other hand, almost all large software products are too complex to contain no mistakes at all. Considering mentioned above, the development task is not to guarantee the absence of mistakes, but to maintain a balance between the conditional reliability of the final software system and the costs at all development stages.

All existing software reliability assessment methods can be divided into three categories [3, 4]:

- black box reliability models;
- software metric based reliability models;
- architecture-based reliability models.

The first two categories are technically not reliability evaluation methods, but rather failure prediction methods, as they are based on the paradigms of increasing reliability and failure rate distribution over time. Since the probability of failure for software systems does not directly depend on the operating time [3], such an estimate is rather inaccurate. The third type of models is a more attractive alternative to the previous ones, as it clearly links program reliability to component reliability, which makes it easier to identify components that are critical in terms of reliability. In addition, these methods can be used for early program reliability assessment [5].

The analysis of available architecture-based reliability models has shown that they are either too complex to be used by ordinary programmers [5], or use entire software modules as architectural units [6, 7], which has a negative impact on accuracy.

The purpose of this paper is to develop an approach that would allow automated assessment of the reliability of software systems without the need to calculate and validate auxiliary coefficients. In addition, this approach is an attempt to move away from the rather traditional strategy of adapting hardware reliability methods to assess software systems.

2. Problem statement

To solve the problem of assessing the reliability of software systems, a number of reliability models are currently used [8]. However, all of these models are partially or fully based on models for assessing the reliability of hardware systems with minor changes, so they contain a number of simplifications and assumptions, which significantly limits the possibility of their application with real software products.

The difference between software and hardware systems is that in software systems, the probability of detecting an error does not directly depend on time, since if a static set of data is input, the program will work without fail indefinitely. This leads to the fact that expressing reliability in terms of time between failures evaluates the tester's qualifications rather than the actual reliability of the program.

Although mistakes are related to the concept of reliability, they cannot be a unit of measurement of reliability by themselves because, even when divided by the number of structural

elements of the program, they do not characterize the ability (or inability) of a software product to convert a set of input data into the expected result of execution. In addition, the number of mistakes in a program does not directly affect reliability, since:

- the result of observing the system is a failure, while the mistake remains hidden;
- failure can be caused by several mistakes simultaneously;
- mistakes can compensate each other in a way that after correcting one of them, the number of failures increases;
- some mistakes can be manifested only in the case of input data set that is impossible or extremely unlikely to be generated under real operating conditions;
- some mistakes lead to a false result regardless of the input data, so in fact, one mistake can cause zero program reliability.

3. Description of the proposed method

A flowchart of the approach to assessing the reliability of software systems based on a graph model of method dependence is shown in figure 1.

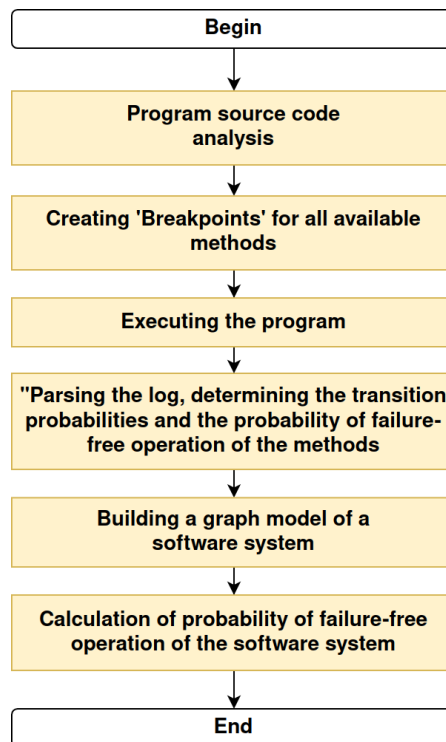


Figure 1: Flowchart of the approach to assessing the reliability of software systems based on a graph model of method dependence.

So, the first step of this approach is “Program source code analysis”, the purpose of which is to break down the program system as a whole into smaller structural elements – methods. To

evaluate the reliability of such a system, we use a graph model [9] in which the vertices of the graph are methods, and the edges are the links between these methods in the program under study. Figure 2 shows an example of such a graph model for a software system consisting of four methods.

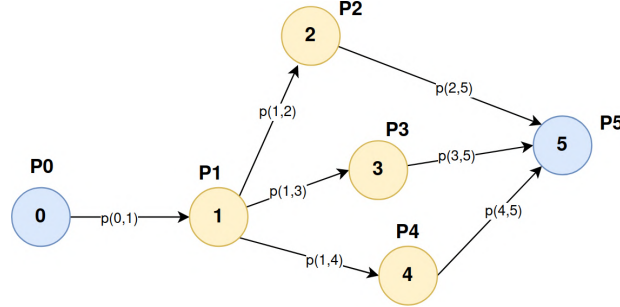


Figure 2: Graph model of the program consisting of four methods.

This oriented graph contains 6 vertices, four of which are responsible for methods, and another two are dummy vertices representing points of entry to and points of exit from the program code.

The values P_1 , P_2 , P_3 and P_4 are the probabilities of failure-free operation of the corresponding methods, and $p(i, j)$ are the probabilities of invocation of the j -th method from the body of the i -th method.

The probability of failure-free operation of each method is calculated as the ratio of the number of its invocations that caused an exception throwing to the total number of method invocations. The transition probabilities are calculated in proportion to the number of invocations to nested methods from the body of the i -th method, so that the sum of all transition probabilities from the body of the i -th method is one (1):

$$\sum_{j=j_{min}}^{j_{max}} p(i, j) = 1 \quad (1)$$

These probabilities are calculated by performing the following steps of the proposed approach, namely “Creating ‘Breakpoints’ for all available methods”, “Executing the program” and “Parsing the log, determining the transition probabilities and the probability of failure-free operation of the methods”.

The next step of the approach is “Building a graph model of a software system”. First of all, a transition probability matrix P_1 with dimension $m \times m$ is constructed, where m is the number of vertices in the graph. In this matrix, the value of the element $P(i, j)$ is the value of the probability of the corresponding transition between the corresponding methods, i.e. $p(i, j)$. If there is no relationship between the corresponding methods, then zero is written in place of the corresponding element. The next step is to calculate the matrix G , for which the j -th row of the matrix P is multiplied by the probability of failure-free operation of the j -th method (for dummy vertices, this parameter is equal to one).

Finally, it is necessary to calculate the matrix T , which in the case of an acyclic graph is calculated using formula (2).

$$T = I + G + G^2 + \dots + G^m \quad (2)$$

And in the case of non-acyclic, by formula (3).

$$T = (I - G)^{-1} \quad (3)$$

The probability of failure-free operation of the software system under test, which is calculated at the last step of the approach, is the value of the element $T_{0,m-1}$ of the matrix T .

4. An example of application of the proposed approach for assessing the reliability of a software system based on a graph model of method dependence

To illustrate the implementation of the proposed approach for assessing the reliability of a software system based on a graphical model of method dependency, the reliability assessment of a simple program has been illustrated, the program code of which is shown in figure 3.

```

public class Main {
    private static int i = 0;

    public static void main(String[] args){
        for(; i++ < Integer.MAX_VALUE;){
            try{
                calculateSomething();
            } catch (Exception ignored){}
        }
    }

    public static void calculateSomething(){
        double a = generateDividend() / generateDivisor();
    }

    public static double generateDividend(){
        if(i % 21 == 0)
            throw new NumberFormatException("Something happened");
        if(i % 3 == 0) return genRandomValue();
        else return getConstantValue();
    }

    private static double genRandomValue(){
        if(i % 99 == 0)
            throw new NumberFormatException("Something happened");
        return (Math.random() * (Math.random() > 0.5d ? 1 : -1));
    }

    private static double getConstantValue(){
        if(i % 95 == 0)
            throw new NumberFormatException("Something happened");
        return 1.751;
    }

    public static Double generateDivisor(){
        if(i % 75 == 0) return null;
        return Math.random();
    }
}

```

Figure 3: Source code of the program under test.

For the convenience of visual demonstration, all methods of a software system are located in the same class, but the proposed approach will work even if all methods belong to different classes.

To emulate the presence of mistakes, conditional statements have been added to the bodies of methods that throw exceptions of the `NumberFormatException` type at certain values of the iterative variable i . The linkage to the iterative variable was made solely in terms of repeatability of results for ease of testing. Building a graph model is started here by creating a methods invocation tree. The tree structure of the program under study in JSON format is shown in figure 4.

```

{
  "name": "calculateSomething",
  "location": "Main.java",
  "nestedMethods": [{
    "name": "generateDividend",
    "location": "Main.java",
    "nestedMethods": [{
      "name": "genRandomValue",
      "location": "Main.java",
      "nestedMethods": []
    }, {
      "name": "getConstantValue",
      "location": "Main.java",
      "nestedMethods": []
    }
  ]
}, {
  "name": "generateDivisor",
  "location": "Main.java",
  "nestedMethods": []
}
]}

```

Figure 4: Structure of the program under study in JSON format.

As figure 4 shows, the program’s structural model contains the “location” field, meaning that this approach will work correctly with methods of the same name located in different classes.

To estimate the number of method executions and the number of failures, IntelliJ Idea’s debug mode is applied to make it log all entries into the methods and all exceptions thrown [10]. ‘Breakpoints’ settings are shown in figure 5.

The log contains all the information necessary for further analysis. After processing this data, we can calculate the probability of failure for each method. The results of the calculations are shown in table 1.

Using the information from table 1, we can build a graph model of the system under study, as illustrated in figure 6.

As it can be seen, the graph shown in figure 6 does not contain cycles, so calculations will be performed by using formula (2). The graph contains seven vertices, so the matrices P , G and T will be of dimension 7×7 . Figure 7 shows the values of all matrices.

As noted above, the probability of system’s failure-free operation is determined by the element $T_{0,m-1}$ of the matrix T , since in our case $m = 7$, then $T_{0,m-1} = T_{0,6} = 0.93$



Figure 5: 'Breakpoints' settings.

Table 1

Log analysis results.

Method	Number of invocations	Number of failures	Probability of failure-free operation
calculateSomething()	2 147 483 647	23 798 952	0.9889
generateDividend()	2 147 483 647	102 261 126	0.9524
genRandomValue()	613 566 756	18 592 932	0.9697
getConstantValue()	1 431 655 765	15 070 061	0.9895
generateDivisor()	2 011 559 528	0	1

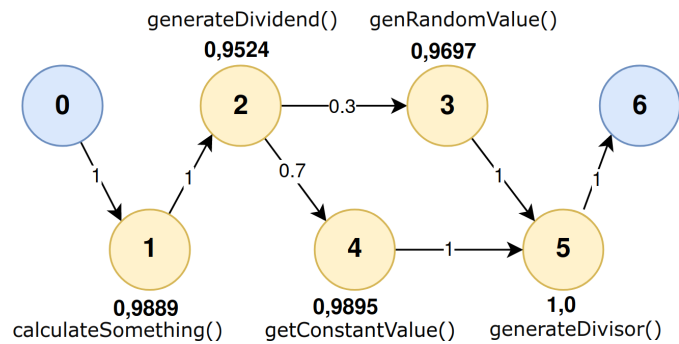


Figure 6: Graph model of a software system.

P	G	T
$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.3 & 0.7 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$	$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.99 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.29 & 0.67 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.97 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.99 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$	$\begin{pmatrix} 1.0 & 1.0 & 0.99 & 0.28 & 0.66 & 0.93 & 0.93 \\ 0.0 & 1.0 & 0.99 & 0.28 & 0.66 & 0.93 & 0.93 \\ 0.0 & 0.0 & 1.0 & 0.29 & 0.67 & 0.94 & 0.94 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.97 & 0.97 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.99 & 0.99 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$

Figure 7: The values of the matrices P , G and T .

Test program whose code is shown in the figure 1 is simple and contains only one entry point. It was made intentionally to be able to check calculations results using traditional methods. According to the fundamentals of reliability theory [11], the probability of system failure is defined as:

$$P(t) = 1 - \frac{n(t)}{N_0}$$

where N_0 is the number of elements at the beginning of the test and $n(t)$ the number of elements that failed during the time interval t .

As it can be seen, program invokes the `calculateSomething()` method 2 147 483 647 times (`Integer.MAX_VALUE`). The number of application failures in this case would be equal to the total number of generated exceptions. Using the data from Table 1, system's failure-free operation probability can be calculated as:

$$P = 1 - \frac{23\,798\,952 + 102\,261\,126 + 18\,592\,932 + 15\,070\,061 + 0}{2\,147\,483\,647} = \frac{159\,723\,071}{2\,147\,483\,647} = 0.9256$$

As it can be seen, the results match, which indicates that proposed approach works correctly.

5. An example of using the method for more complex systems

Now let's apply the proposed method to measure the reliability of a more complex program. The class diagram of the test program is shown in the figure 8.

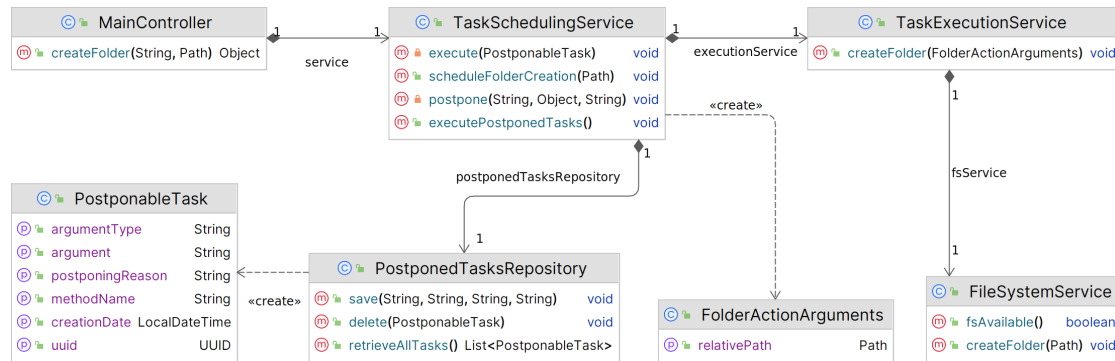


Figure 8: Test application class diagram.

This program is a web application that was created using the Spring framework. The program contains 7 classes and 11 methods and 2 entry points – the first is implemented through the controller, and the second through the task scheduler.

As in the previous example, we will get information about the number of method invocations and errors that have occurred by parsing the log. Structured data are shown in table 2.

As it can be seen, there are three methods `createFolder()` in the source code of the program, but they were separated by the model and processed correctly. Figure 9 shows a graphical model of the system under study, built using data from table 2.

Table 2
Method invocation statistics.

Method	Number of invocations	Number of failures	Probability of failure-free operation
MainController.createFolder()	1 000	0	1
scheduleFolderCreation()	1 000	0	1
TaskExecutionService.createFolder()	3765	0	1
fsAvailable()	3765	35	0.991
postpone()	333	5	0.985
save()	328	2	0.993
FileSystemService.createFolder()	1074	75	0.93
delete()	326	1	0.997
executePostponedTasks()	114	0	1
retriveAllTasks()	114	2	0.982
execute()	2768	13	0.995

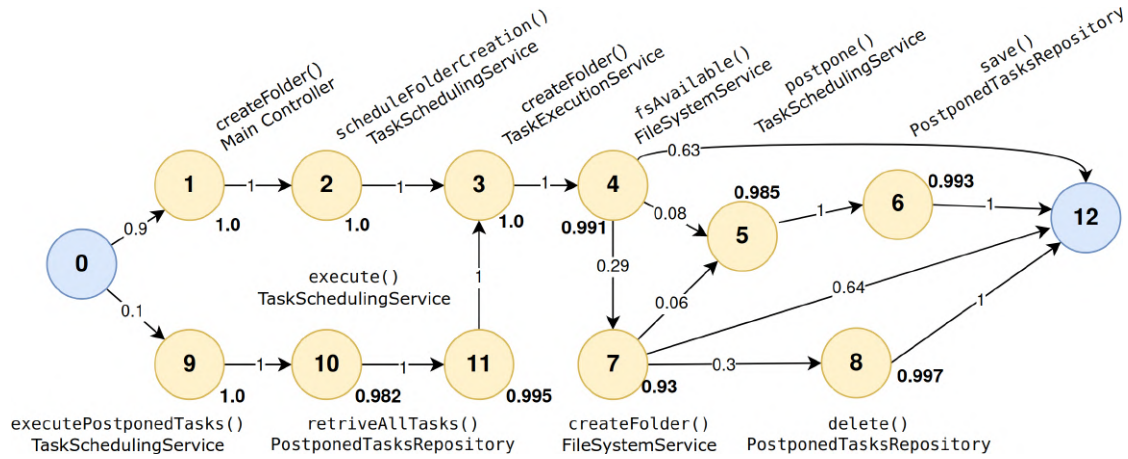


Figure 9: Graph model of a software system under study.

Since graph contains 13 vertices, matrices P , G and T will be of dimension 13×13 . Figure 10 shows the values of all matrices.

As in the previous example, the probability of system's failure-free operation is determined by the element $T_{0,m-1}$ and therefore $T_{0,m-1} = T_{0,12} = 0.97$.

From the perspective of model performance, parsing log files turned out to be the most difficult task, as expected. Since the sizes of the logs in the first and second examples differ significantly, it would be more appropriate to compare the time spent on calculations. The time to calculate the matrices for the first and second examples was 2.2ms and 3.6ms, respectively. In general, the calculation time (including log parsing) did not exceed 50ms in both cases. Taking this into account, it can be concluded that the proposed model can calculate reliability indicators for much larger software systems in an acceptable amount of time.

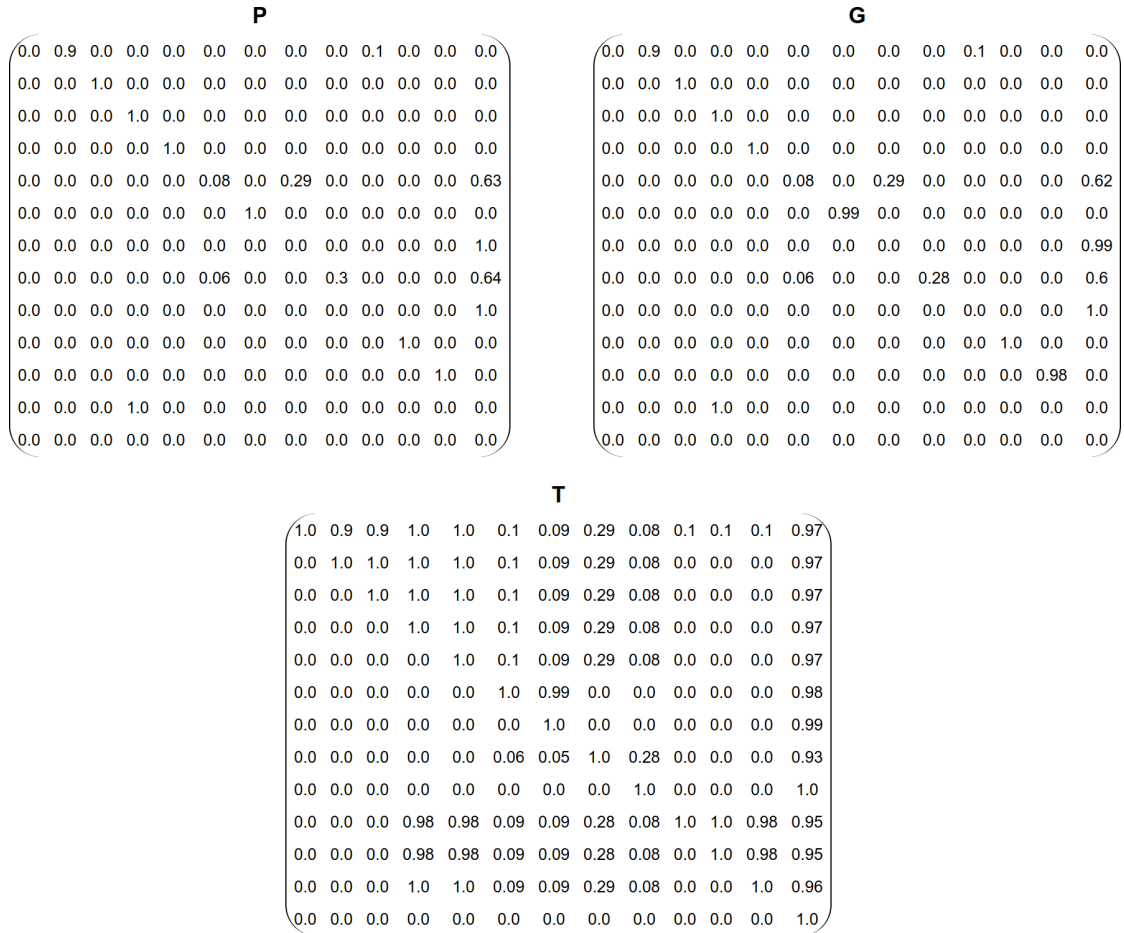


Figure 10: The values of the matrices P , G and T .

6. Conclusions

To test the proposed approach, it was implemented as an plug-in for integrated development environment IntelliJ Idea which could measure the reliability of programs written in Java. With minor changes, it can be adapted to work with code written in any object-oriented programming language. The objective was to develop an approach that would make it possible to evaluate the reliability of both – the entire software system and its specific modules at the testing stage. Upon testing completion, reliability indicator is calculated, which could be used to make decisions about the feasibility of further program improvement.

The example shows the effectiveness of this approach. The developed intellectualized system based on the proposed approach can be used to assess software systems reliability of any complexity and architecture.

References

- [1] ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, Standard, International Organization for Standardization, 2011.
- [2] V. V. Vyshnivskiy, V. V. Vasylenko, M. P. Hnidenko, *Osnovy nadiinosti ta diahnostryky informatsiinykh system* [Fundamentals of reliability and diagnostics of information systems], FOP Huliaieva V. M., 2020.
- [3] I. Eusgeld, F. Fraikin, M. Rohr, F. Salfner, U. Wappler, *Software Reliability*, in: I. Eusgeld, F. C. Freiling, R. Reussner (Eds.), *Dependability Metrics: Advanced Lectures*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 104–125. doi:10.1007/978-3-540-68947-8_10.
- [4] N. R. Barraza, *Five Decades of Software Reliability, Past, Present, Future and New Challenges*, in: *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2018, pp. 88–94. doi:10.1109/ICRITO.2018.8748556.
- [5] A. Dimov, S. Punnekkat, *Fuzzy Reliability Model for Component-Based Software Systems*, in: *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2010, pp. 39–46. doi:10.1109/SEAA.2010.46.
- [6] K. Goševa-Popstojanova, K. S. Trivedi, *Architecture-based approach to reliability assessment of software systems*, *Performance Evaluation* 45 (2001) 179–204. doi:[https://doi.org/10.1016/S0166-5316\(01\)00034-7](https://doi.org/10.1016/S0166-5316(01)00034-7), *Performance Validation of Software Systems*.
- [7] V. C. Skanda, S. Srinivasa Prasad, G. R. Dheemanth, N. S. Kumar, *Assessment of Quality of Program Based on Static Analysis*, in: *2019 IEEE Tenth International Conference on Technology for Education (T4E)*, 2019, pp. 276–277. doi:10.1109/T4E.2019.00072.
- [8] V. Yakovyna, V. Matseliukh, *Ohliad i analiz modelei nadiinosti prohramnoho zabezpechenia* [Review and analysis of software reliability models], *Visnyk Natsionalnoho universytetu "Lvivska politekhnika". Kompiuterni nauky ta informatsiini tekhnolohii* 864 (2017) 103–138. URL: <https://science.lpnu.ua/sites/default/files/journal-paper/2018/jul/13773/17.pdf>.
- [9] S. Krepych, I. Spivak, S. Spivak, *Methodology of Formation of the Individual Study Plan of the Student Based on the Graph Model of the Dependence of Disciplines*, *CEUR Workshop Proceedings* 3426 (2023) 298–307. URL: <https://ceur-ws.org/Vol-3426/paper24.pdf>.
- [10] *Breakpoints | IntelliJ IDEA Documentation*, 2024. URL: <https://www.jetbrains.com/help/idea/using-breakpoints.html>.
- [11] Y. Yan, Y. Peng, C. Liu, H. Li, *A New Software Reliability Evaluation Model Based on the Probability of the Failure Mode*, in: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018, pp. 35–41. doi:10.1109/QRS-C.2018.00020.