METHOD TO ASSESS RELIABILITY OF COMPLEX SOFTWARE FUNCTIONING

A.N. Kovartsev, D.A. Popova-Kovartseva

Samara National Research University, Samara, Russia

Abstract. It has been established that the software reliability concept has many specific features and differs significantly from the concept of reliability of technical systems. This greatly complicates the task of assessing overall reliability of a complex software product. The paper introduces a new reliability indicator—average reliability—as well as a method to assess average reliability of a complex software system based on known characteristics of its modules. The introduced approach is theoretically justified.

Keywords: software reliability, unreliability, execution route, software module.

Citation: Kovartsev AN, Popova-Kovartseva DA. Method to assess reliability of complex software functioning. CEUR Workshop Proceedings, 2016; 1638: 813-819. DOI: 10.18287/1613-0073-2016-1638-813-819

Introduction

Reliability is an important and natural prerequisite for modern hardware and software suites (HSSs). Nowadays, technical devices and systems are highly reliable. For example, in [1], it is shown that the probability of computer failure in the TU-324 flight navigation system is 10^{-10}, which meets modern requirements for fail-free performance of equipment used aboard space- and aircrafts. As for mathematical methods of assessment of HSS software reliability, the situation is not quite satisfactory.

For a long time, general attention was paid directly to programming techniques, and problems of testing and assessment of software reliability were considered ‘necessary evil’. Only practical use could reveal software failures. In the late 1990s, the attitude toward assessment of software reliability started to change due to a significant increase in complexity of software and understanding that high reliability of an HSS could be achieved only by fault-free operation of both its hard- and software.

In ISO 9126:1991 [2], reliability is named one of the main characteristics of software quality.

Modern software reliability models, both analytical and empirical, either explicitly or implicitly use the mathematical apparatus of the technical systems reliability theory, i.e. they view the general system reliability concepts as dependent on time, which is incorrect in case of software.

1 Software functioning reliability concept

The problem of software reliability has at least two aspects: assurance and measurement (assessment) of software reliability. Almost all existing literature deals with the first aspect, whereas not enough attention is paid to the problem of measurement of software reliability.

In [3], reliability is defined as 'the probability that software will not cause the failure of a system for a specified time'. Then it is stated that software failure is a subjective notion: one and the same program would suit one user, but would not another one. Moreover, software failures impact the end result differently: some lead to a catastrophe, others simply to orthography mistakes displayed on the screen. The introduced software reliability definition uses the concept of time factor, borrowed from terminology related to technical devices reliability.

The general postulate of the technical devices reliability theory states that the failure rate of an element depends on operation time, which makes no sense in relation to software reliability.

In software, failures occur with specific, clearly defined combinations of input data. Countless software starts with one or a limited number of sets of input data that do not cause the software to fail will not cause a failure regardless of the operation time. Moreover, reliability of software modules can only increase over time as errors are being diagnosed and fixed. Theoretically, there can occur a situation when testing of a software module does not reveal errors anymore.

This renders concepts of operating time to failure and failure probability for a given period completely useless for the purposes of programming.

The classic reliability theory uses an axiomatic assumption that a probability of failure p_0, albeit small but higher than zero, always exists for any technical device due to design errors or operational wear of the device. This hypothesis is proven experimentally. However, software is not subject to wear or damage; therefore, lack of software reliability can be caused exclusively by programming errors made at the design stage. At the same time, an unlimited number of simple programmes with zero probability of failure can be developed. This renders the entire apparatus of the classic reliability theory useless for practical application.

The most general definition of software reliability is proposed by B. Meyer in [4], where he defines reliability as 'the ability of software to provide reasonable results under any possible circumstances and, in particular, under abnormal conditions'.

Then, unreliability may be viewed as ratio of cardinality of error situations $|\Omega_E|$ to cardinality of input data $|\Omega_{In}|$ ($\Omega_E \subset \Omega_{In}$). In connection with computers, this is ratio between the number of combinations of software input data that cause errors to the general number of combinations of input data. In this case, unreliability can be calculated according to the following formula: $q = |\Omega_E| / |\Omega_{In}|$.

However, the number of input data combinations for software modules is generally so high that it is virtually impossible to account for every combination for a modern computer. Positive results have nowadays been acquired for relatively simple software modules with just a few input parameters. For some modules, their correctness

[4], i.e. $q = 0$, can be proven. In other cases, q can be established experimentally [4, 5].

Input data for modern complex software can include hundreds and thousands of variables. That renders direct testing methods ineffective and virtually impossible as the task of software testing becomes as complicated as the task of global optimization [6, 7, 8].

It looks natural to use known characteristics of software components to assess reliability of complex software, just like for technical systems. However, this approach also leads to unexpected results.

First, no matter how complex software structure is, software elements (modules) are always connected (reliability-wise) sequentially.

Second, software reliability depends not only on reliability of its modules but also on correctness of organisation of software functioning logic [9, 10].

Third, software reliability cannot be calculated directly based on reliability of its components (software modules).

Let us review the last argument in detail.

2 Challenges in assessment of a complex software system's reliability

Let us assume we know reliability characteristics of all modules of a software product. Let q_i be unreliability of *i*-th module $A_i : X_i^{In} \to Y_i^{Out}$ calculated with account of all possible applications of the module, i.e. over a significantly wide range of input data, $X_i^{In} = (x_{1i}^{In}, x_{2i}^{In}, ..., x_{n_i i}^{In}) \in \Omega_{In}^i$, where Ω_{In}^i is the range of values of the input data vector of the *i*-th module.

Let us assume there are no logic errors in organisation of software functioning logic (this problem needs to be reviewed separately, see [5]).

Let us assume the software system has *L* execution routes [8]. Let us define every route as $M_j = i_{j_0} i_{j_1} ... i_{j_k}$, where i_{j_k} is number of a software module of the system.

Let us assume every *i*-th object on the *j*-th route is called $\tilde{m}_{ij}$ times on average.

Unreliability of the *i*-th module on the *j*-th route for $\tilde{m}_{ij}$ calls can therefore be assessed as:

$$Q(\tilde{m}_{ij}) = 1 - (1 - q_i)^{\tilde{m}_{ij}} = Q_{ij}. \quad (1)$$

It is obvious that, should an error occur in any module on the route, the entire route may be deemed a failure. Therefore, route unreliability can be assessed as:

$$Q_{M_j} = 1 - \prod_{i=1}^{j_k} (1 - Q_{ij}). \quad (2)$$

Let us assume every route is executed with probability r_j, and $\sum r_j = 1$ is true. Therefore, unreliability of the software system may be assessed as:

$$Q_{PS} = \sum_{j=1}^{L} r_j Q_{M_j} \ . \tag{3}$$

The formula (3) is true if assessments of unreliability of software system modules are constant for various software. However, this is not the case.
As an example, let us review a programme for thermogasdynamical calculation of a two-shaft turbojet engine (see Fig. 1).

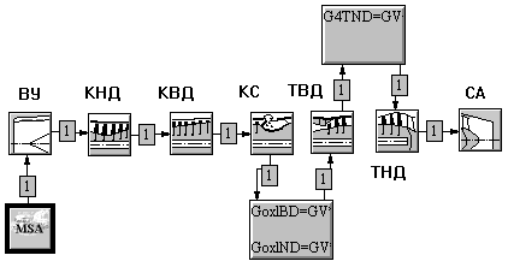

Fig. 1. Programme for thermogasdynamical calculation of a two-shaft turbojet engine

This programme is rather simple and employs only one route. However, functional nonlinear transformations that occur in every module cause changes in laws of distribution of in- and output data (input data for the next module) for every single module.
Figures 2a–2f show ranges of change of input data for calculation of turbojet engine and its components (high pressure compressor, combustion chamber, high and low pressure turbines, and nozzle diaphragm).

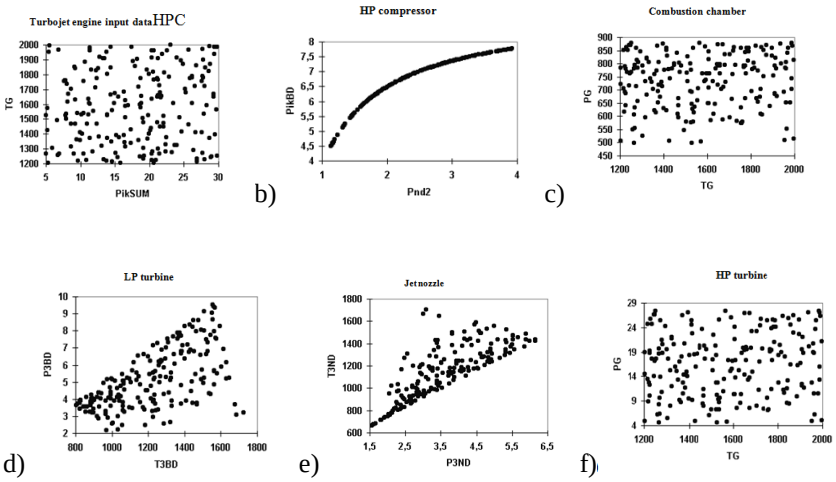

Fig. 2. Ranges of change of input data for calculation of a turbojet engine and its components

As is seen from the figures, the laws of distribution of input data have significantly transformed for every module, up to the point of loss of scale. At the same time, input data were supplied uniformly for each module during autonomous testing. This could lead to over- or underestimation of reliability of software modules the system is composed of and distort the entire assessment of software reliability.
This means software reliability assessment should use probability distribution for input data for every module used in the software and, apparently, account for particular characteristics of all software tasks.
Let us review influence of the above factors on changes in assessment of software modules' reliability.

3 A method to assess reliability of a complex software system

Let us assume there is a module that only has one input parameter x, and the module has passed a series of tests (of N tests) without errors; also, $x \in [a,b]$. From this, it may be concluded that $q \le 1/(N+1)$ and $V(\Omega_E) \approx (b-a)/(N+1)$. Let us assume then that input parameter value range $x \in [c,d] \subset [a,b]$ of the module has changed during its use as part of software; at the same time, the law of distribution has not changed. Therefore, the final unreliability of the module, with account to probability $P\{\Omega_E \subset [c,d]\}$, may be assessed as $\tilde{q} = \frac{V(\Omega_E)}{d-c} P\{\Omega_E \in [c,d]\} = \frac{V(\Omega_E)}{d-c} \cdot \frac{d-c}{b-a} = \frac{V(\Omega_E)}{b-a} = q,$ i.e. the unreliability characteristic does not change its value if the uniform law of distribution of parameter x is observed.
For simplicity, let $a = c = 0$. Let us assume next that over an intercept [0, d] , x follows a non-uniform, quasiexponential distribution with the distribution density function $f(x) = K\lambda e^{-\lambda x}$, where $K = \frac{1}{1-e^{-\lambda d}}$. Let $V(\Omega_E) = \Delta$. Unreliability of the module may be defined using equation $\tilde{q}(x) = P\{\Delta \in [0,d]\} \int_x^{x+\Delta} K\lambda e^{-\lambda x} dx = \frac{d}{b} \frac{e^{-\lambda x}}{1-e^{-\lambda d}} (1-e^{-\lambda \Delta})$ and depends on where the error situations range lies. The unreliability is not equal to value q found during the autonomous testing (Fig. 3).

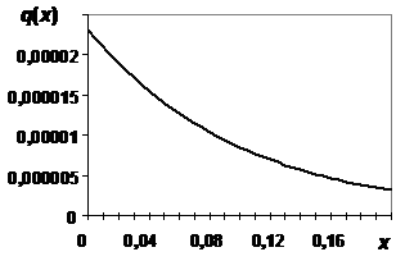

Fig. 3. Dependence of unreliability of the module on parameter x

What should be considered a characteristic of unreliability of the module then?
Let us assume the error situations range lies randomly at any spot of the intercept [***0, d***] and let us calculate the average value $\tilde{q}(x)$,

$$q_m = \int_0^d \tilde{q}(x)\frac{1}{d}dx = \frac{P\{\Delta \in [0,d]\}K(1-e^{-\lambda\Delta})}{\lambda d}\int_0^d \lambda e^{-\lambda x}dx = \frac{d}{b}\frac{(1-e^{-\lambda\Delta})K}{\lambda d}(1-e^{-\lambda d}) = \frac{(1-e^{-\lambda\Delta})}{\lambda b} \quad (4)$$

Characteristic q_m does not depend on the parameter *x* and is only defined by the size of the error range Δ and the exponential distribution law parameter λ.
As calculations show (see table below), significant changes of values λ (distribution non-uniformity degrees) lead to virtually no changes of the value q_m. Moreover, $q_m \approx q$. In this example, $q = 0.00001, \mathrm{d} = 0{,}2, \mathrm{b} = 1$. For significant non-uniformity $\lambda > 10$, the condition $q(x) > q_m$ is observed on relatively small intercepts of the input parameter definition range, and in other cases, $q(x) \le q_m$ is observed. If λ is small, then $\max\ q(x) \approx q_m$ is true.

Table. Influence of non-uniformity of input data distribution law on q_m

λ	q_m	max q(x)	Δx
0.1	9.99999E-06	1.01E-05	0.1
1	9.99995E-06	1.1E-05	0.0975
10	9.9995E-06	2.31E-05	0.0825
100	9.995E-06	0.0002	0.0275
1000	9.95017E-06	0.00199	0.005

If $q_m \approx q$ is true for any distribution law, q can be interpreted as the average and expected assessment of module unreliability and used for calculation of software reliability by interpreting Q_{PS} as the average expected value of unreliability of software as a whole. However, this result needs yet to be extended to the multidimensional case.
For a random distribution law $F(x)$ of input data of a software module with distribution density $\varphi(x)$, the following general regularities may be revealed.
Let us define a normalisation condition using the following equations:
$\int_0^d K\varphi(x)dx = 1$ or $K = 1/\int_0^d \varphi(x)dx$.
Next,

$$\tilde{q}(x) = P\{\Delta \in [0,d]\}\int_x^{x+\Delta} K\varphi(x)dx = KP\{\Delta \in [0,d]\}(F(x+\Delta) - F(x)).$$

If $\Delta \to 0$, then

$$\lim_{\Delta\to 0}\tilde{q}(x) = KP\{\Delta \in [0,d]\}\Delta \lim_{\Delta\to 0}\frac{(F(x+\Delta) - F(x))}{\Delta} = KP\{\Delta \in [0,d]\}\varphi(x)\Delta.$$

Then,

$$q_m = \int_0^d \tilde{q}(x)\frac{1}{d}dx = \frac{P\{\Delta \in [0,d]\}K\Delta}{d}\int_0^d \varphi(x)dx = \frac{d}{b}\frac{\Delta}{d} = \frac{\Delta}{b} = q. \quad (5)$$

Therefore, the average unreliability of the software module (5), in the limit of $\Delta \to 0$, coincides with the unreliability assessment calculated for the uniform input data distribution law.

Conclusion

It may be concluded that assessment of software reliability differs significantly from the analogous task for technical systems. At the same time, the averaged characteristic of software reliability may be calculated using known assessments of reliability of software modules the product consists of. Testing of a software component can be carried out using a standard procedure of stochastic testing with the uniform input data distribution law.

The paper is prepared with state support from the Ministry of Education and Science of Russia as part of the Programme for competitive growth of the Samara State Aerospace University among leading world science and education centres for 2013–2020.

References

1. Avakyan AA, Iskandarov RD, Novikov NN et al. The concept of building a highly reliable calculators for aviation and rocketry. Reliability and quality 2001, Proceedings of the International Symposium, Penza, 2001: 33-37. [in Russian]
2. ISO 9126:1991. Information technology. Software product evaluation. Quality characteristics and guidelines for their use. 186 p. [in Russian]
3. Myers, G. Reliability Software. Moscow: Mir, 1980; 360 p. [in Russian]
4. Meyer B, Baudoin C. Programming methods. Vol. 2 Moscow: Mir, 1982; 368 p. [in Russian]
5. Kovartsev AN. Automate development and testing of software. Samara: Samara Aerospace University, 1999; 150 p. [in Russian]
6. Kovartsev AN, Popova-Kovartseva DA, Gorshkova EE. Software testing based on global search of several variables functions discontinuity. CEUR ITNT 2015 Information Technology and Nanotechnology, Samara, 2015: 389-396.
7. Kovartsev A N, Popova-Kovartseva DA. On efficiency of parallel algorithms for global optimization of functions of several variables. Computer Optics, 2011; 35(2): 256-261. [in Russian]
8. Kovartsev AN. A deterministic evolutionary algorithm for the global optimization of morse cluster. Computer Optics, 2014; 39(2): 234-240. [in Russian]
9. Lipaev VV. Reliable software. Moscow: SINTEG, 1998; 232 p. [in Russian]
10. Kovartsev AN. An efficient algorithm for testing the truth of assertions for real numbers expressed in relational signatures. Computer Optics, 2014; 38(3): 550-554. [in Russian]