Quantitative Quality Analysis of Scientific Software – Case Study

BOJANA KOTESKA, MONIKA SIMJANOSKA and ANASTAS MISHEV, University SS. Cyril and Methodius

In this paper we make quantitative analysis of software for calculating ECG-derived heart rate and respiratory rate. The focus of the paper is to provide insight into the importance of the development process through a quantitative quality prism with the aim to stress the importance of the quality aspect of the code and not just the accuracy aspect. The results confirm that the ad-hoc development of a software leads to non-satisfactory results for the quality metrics and attributes of the software. Some of the main weaknesses of the code refer to cyclomatic complexity, blocks duplication, and consecutive failures.

1. INTRODUCTION

Scientific software is usually of multidisciplinary nature solving the problems in various fields by applying computational practices. In this paper, we analyze the quality of scientific software for calculating ECG-derived heart rate and respiratory rate.

The aim of this software is to aid the triage process in the emergency medicine which is crucial for ranging the priority of the injured victims in mass casualty situations [Hogan and Brown 2014], whether they are man-made, natural or hybrid disasters. An efficient triage process takes less than 30 seconds when performed manually and the challenge is to reduce this time and the medical persons needed in order to optimize the process when there are hundreds of injured people. This optimization can be achieved by using the benefits of the biosensor technologies to extract the vital signs needed for the triage. Following this idea, we developed scientific application that extracts heart rate and respiratory rate from ECG signal. The accuracy of the results was crucial whereas the optimization and parallelization of the code was not of our concern, meaning we prioritized the quality of the algorithm before the quality of the code.

Therefore, in this paper we perform quantitative quality evaluation of the application by using formal methods and software engineering practices. This is an important step if we want improve the quality and to integrate the application as a part of a larger medical system. The method for quality assessment relies on quantitative method for evaluating the well-known metrics and attributes, providing an easy comparison with other similar applications.

In [Adewumi et al. 2016], the authors made a review of 19 open-source software quality assessment models. A special attention is put on the quality models for computation-dominant software. A model for evaluating the quality of scientific software that checks only the critical modules and it satisfies

Author's address: Bojana Koteska, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: bojana.koteska@finki.ukim.mk; Monika Simjanoska, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: monika.simjanoska@finki.ukim.mk; Anastas Mishev, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: anastas.mishev@finki.ukim.mk.

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

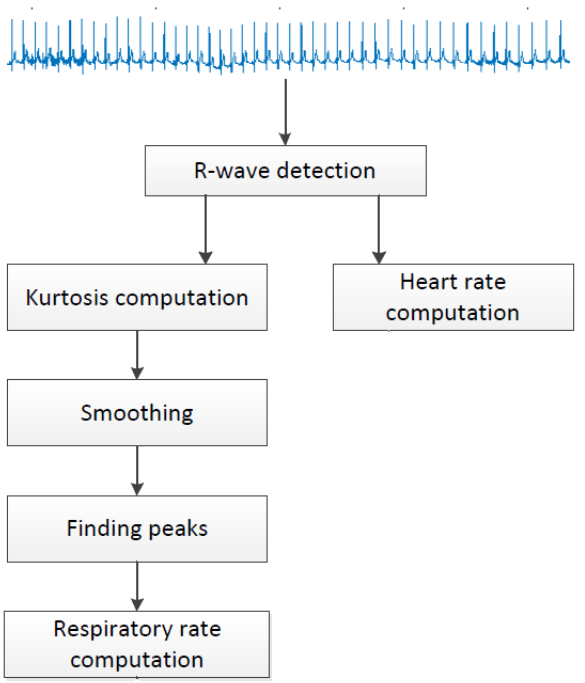

Fig. 1: The algorithm.

the requirements of the IEEE 12207 standard [Maheswari and Prasad 2011]. Quality metrics important for for scientific software are proposed in [Neidhardt 2017]. Most of them measure quantitative statements about the source code. The other ones are related to the software and its dynamics. A probabilistic approach for computing high-level software quality characteristics by aggregating low-level attributes to higher levels is elaborated in [Bakota et al. 2011].

The paper is organized as follows. In Section 2 we provide description of the algorithm. Section 3 provides a comprehensive explanation of the quality evaluation methodology whose results are presented in Section 4. The final Section 5 concludes the paper.

2. SOFTWARE DESCRIPTION

The purpose of developing the software is to take part in the triage procedure for determining a patient's condition [Gursky and Hrečkovski 2012]. The idea is to automatically determine HR and RR parameters which are the crucial part for accurate triage. The software aims to use the leverage of wearable biosensor technology and to use only ECG data to calculate HR and RR parameters. The accuracy of the algorithm is already proven in our previous research [Simjanoska et al. 2018] by using both publicly available databases and our own database.

The reason for developing the software whose quality is tested in this paper is to achieve real-time processing of large amounts of data and to be efficient in terms of the power-demanding Bluetooth connection with the biosensor due to the constant data transmission.

Generally, the algorithm encompasses four methods, one is needed to estimate HR and four consequentive methods are neccessary to estimate the RR parameter. As depicted in 1, the input is a raw

ECG signal. The initial step and the only method for direct calculation of the HR is the R-peak detection performed by using the the Pan Tompkins algorithm [Pan and Tompkins 1985]. Hereupon, the HR is calculated according to the following equation:

$$HR = (\frac{signal\_length}{number\_of\_R\_peaks})^{-1} * 60 \tag{1}$$

In the subsequent part of the algorithm, the obtained R-peaks are used to enable the derivation of the RR. Followed-up by the kurtosis computation technique for measuring the peakedness of the signal's distribution, the locations of the local maxima are obtained which are needed for the smoothing method upon which a peak finder method is applied to find the local maxima (peaks) of the ECG signal. Those peaks are the number of respirations according to which the respiratory rate is calculated:

$$RR = (\frac{signal\_length}{number\_of\_respirations})^{-1} * 60 \tag{2}$$

The implementation of the proposed algorithm is realized into the C programming language. It has 1148 lines of code in total and is organized into 26 logical units. When developing the code, no software engineering practices are used. It is developed ad hoc, directly starting from the programming phase.

3. METHODOLOGY

To analyze the quality of the software for calculating ECG-derived heart rate and respiratory rate we used the quality model for scientific applications proposed in [Koteska et al. 2018]. According to the model, three quality attributes were taken into consideration: maintainability, portability and reliability. The quality attributes were then represented by other quality attributes (yellow rectangles) and metrics that can be quantified (blue rectangles). The selection of the quality attributes and metrics is shown in Fig. 2. The reason for choosing these attributes is the nature of scientific applications. Maintainability is an important aspect of scientific software quality, as scientific applications should be structured to be easily changed. In some cases, changes are necessary to increase the efficiency of applications because of limited memory resources and time. Regarding portability, certain parts of an application may be reused, especially in the same scientific domain. Reliability is critical since scientific simulations may take a long time, and the application must remain in a stable condition [Koteska et al. 2018].

The source code volume metric was measured by calculating MY (man years) and by determining the category of the software according to the programming languages productivity table [Jones 1996]. A possible categorization is the following: 0-8 MY, 8-30 MY, 30-80 MY, 80-160 MY, >160 MY, where the first category is the best. [Heitlager et al. 2007].

The degree of source code duplication is measured as a percentage of duplicated lines of code (blocks of code with at least 6 repeated code lines). If the percentage of duplicated lines of code is less than 10%, then the results are satisfactory [Mguni and Ayalew 2013] [Hegedűs et al. 2010].

The complexity of each software units was calculated by using the following equation:

$$G = E - N + P \tag{3}$$

where E is the number of edges in the control flow graph generated from the source code, N is the number of vertices and P is the number of graph connected components. Software Engineering Institute [ORegan 2015] has defined the following risk categories for cyclomatic complexity of software units: (1-10) - simple, without much risk; (11-20) - a little complex, moderate risk; (21-50) complex, high risk; and (> 50) - without the possibility of testing, very high risk.

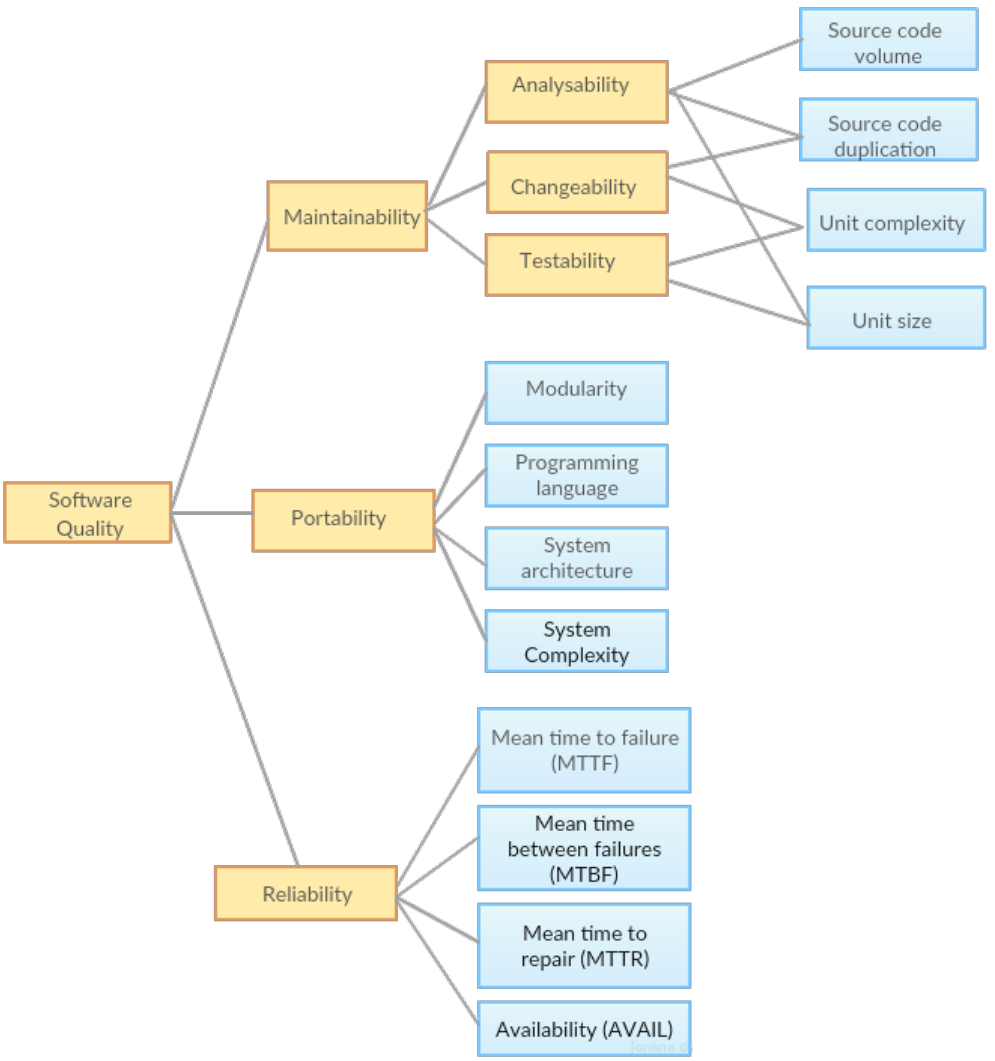

Fig. 2: Attributes and metrics determining the scientific software quality

Logical unit size metric for measuring the size of logical units is the number of lines of code (LOC) [Manoharan 2014] [Heitlager et al. 2007].

Modularity metric is calculated as:

$$Mo = Mn * 0.5 + 1/Ms * 0.5 \tag{4}$$

Mn is number of modules in the system and Ms is the average module size in LOC [Oman and Hagemeister 1992] [Johnson et al. 2015].

Programming language (Pl) metric is calculated by applying the following rules [Oman and Hagemeister 1992] [Johnson et al. 2015]:

—If the programming language Java, C, C ++, or Python the result is 2;

—If another high-level programming language is used, the result is 1.

The following rules are applied for the system architecture metric Oa:

—For an independent platform programming language the results is 1;

—For a non independent platform programming language the following three parameters were considered: the separation between the platform-dependent code and platform-independent code, the usage of standardized application program interfaces (APIs), and standard models for displaying data and usage of platform-dependent libraries. Each parameter is weighted equally.

The system architecture metric range is 1 to 5, (1 means best) [Oman and Hagemeister 1992] [Johnson et al. 2015].

The system complexity metric (Co) is calculated as [Johnson et al. 2015]:

$$Co = G * 0.5 + Cp * 0.5 \tag{5}$$

Cp denotes the average module coupling and G denotes the average unit cyclomatic complexity. The module coupling can be calculated as follows [Pressman 2005]:

$$C = 1 - 1/(d_i + 2 * c_i + d_o + 2 * c_o + g_d + 2 * g_c + w + r) \tag{6}$$

d_i represents the number of input data parameters, and c_i denotes the number of input control parameters. Similarly, d_o and c_o are the numbers of the output data and control parameters; g_d denotes the number of global data variables, and g_c the number of global control variables. For a specific module, w represents the number of modules called in the module, and r represents the number of modules from which the module is called.

Mean time to failure metric (MTTF) is the average time between two consecutive failures of the software system [Prakash and Vidyarthi 2015].

Mean time to repair metric (MTTR) is the average time needed system to repair after a failure [MICHLOWICZ 2013].

Mean time between failures (MTBF) is defined as [Kaur and Bahl 2014]:

$$MTBF = MTTF + MTTR \tag{7}$$

The availability of the system represents the reliability of the system and is calculated as:

$$AVAIL = MTBF/(MTBF + MTTR) \tag{8}$$

4. RESULTS

The results for the quality metrics are obtained by analyzing the software source code and performance. To calculate some of the metrics, we used tools for inspecting the source code and some of the metrics were calculated manually.

The results for the metrics are shown in Table I.

Source code volume metric is expressed in MY and it is obtained from the programming language tables [Jones 1996]. Function points (FPs) are calculated when LOC is divided by 128 because the application is written in C programming language. Source code volume in MYs is calculated from the total FPs divided by the number of FPs monthly per programmer, multiplied by 12.

The percentage of duplicated source code blocks was detected by using the tool PMD [InfoEther].

The complexity of each unit and the parameters needed for its calculation according to the eq. 3 is shown in Table III. Then, the average unit complexity is calculated.

Table I. : Quality metrics results

Metric	Result
Source code volume	0.09
Source code duplication	17.25
Average unit complexity	90.77
Average unit size (LOC)	44.12
Modularity	13.011
Programming language	2
System architecture	1
System complexity	45.77
Mean time to failure (minutes)	25.37
Mean time to repair (minutes)	14
Mean time between failures (minutes)	39.37
Availability	0.74

Table II. : Percentage of Source Code According to the Risk Categories for Cyclomatic Complexity

1-10	20.12 %
11-20	2.44 %
21-50	9.67 %
>50	67.68 %

The calculations according to the risk categories for cyclomatic complexity of software units, are shown in Table II.

The software has three modules with cyclomatic complexity greater than 250, which negatively affects the system complexity.

The average unit size is calculated from the second column LOC in Table III.

Software modularity is calculated according to eq. 4 and system complexity is calculated according to eq. 5. The first (complexity) and eighth (coupling) columns from Table III are used to calculate the system complexity. The column coupling is obtained form the other columns as specified in eq. 6.

The metrics: mean time to failure (MTTF), mean time between failures (MTBF), mean time to repair (MTTR), and availability (AVAIL) are calculated for an execution time of 172 minutes.

The metrics results are used for quantification of the quality attributes. All the metrics are weighted equally. For example, the analysability attribute is calculated from the metrics: source code volume, source code duplication and unit size (Fig.2). The analysability attribute value is an average of the three metric values. If a metric affects an attribute negatively, then the inverse values of the metric are used. According to that, the following results are obtained for the quality attributes (Table IV).

5. CONCLUSION

In this paper we analyzed the quality of the software for calculating ECG-derived heart rate and respiratory rate. The analysis are made according to the quality model composed of standardized quality attributes and metrics. Each of the quality attributes is represented by other quality metrics that are quantifiable.

Based on the obtained results and the acceptable ranges for the metrics, we can conclude that some of the results for the quality attributes of the software are not satisfactory. For example, 67.68% of the source code is in the worst risk category for cyclomatic complexity. Another example is that 17.25% of the source code blocks are duplicated. Mean time between two consecutive failures of the system was 39 minutes and this can be a problem when long ECG signal should be processed. This happened be-

Table III. : Results and Parameters for the software units

Software unit	complexity	LOC	input param.	output param.	global var.	modules called	called in modules	coupling
void diff	27	37	1	0	0	1	1	0.67
double skip_to_last_equal_value	14	28	2	1	0	2	1	0.83
void do_vectors	166	155	5	0	0	8	1	0.93
static void findLocalMaxima	339	285	3	0	0	9	1	0.92
static void parse_inputs	36	64	6	0	0	3	1	0.90
void findpeaks	789	238	2	0	0	13	1	0.94
void b_emxInit_real_T	2	11	3	0	0	0	4	0.86
void emxEnsureCapacity	7	21	3	0	0	0	6	0.89
void emxFree_boolean_T	3	7	1	0	0	0	2	0.67
void emxFree_int32_T	3	7	1	0	0	0	3	0.75
void emxFree_real_T	3	7	1	0	0	0	6	0.86
void emxInit_boolean_T	2	12	2	0	0	0	2	0.75
void emxInit_int32_T	2	12	2	0	0	0	2	0.75
void emxInit_real_T	2	12	2	0	0	0	9	0.91
void HRRRshort_initialize	23	2	0	0	0	1	0	0.00
void HRRRshort	888	99	4	0	0	7	0	0.91
double nanmean	7	18	1	1	1	1	1	0.80
void rt_InitInfAndNaN	22	8	1	0	6	6	1	0.93
boolean_T rtIsInf	2	2	1	1	2	0	3	0.86
boolean_T rtIsNaN	2	6	1	1	0	0	6	0.88
real_T rtGetInf	5	31	0	1	0	1	1	0.67
real32_T rtGetInfF	1	4	0	1	0	0	2	0.67
real_T rtGetMinusInf	4	31	0	1	0	1	1	0.67
real32_T rtGetMinusInfF	1	4	0	1	0	0	2	0.67
real_T rtGetNaN	7	31	0	1	0	1	1	0.67
real32_T rtGetNaNF	3	15	0	1	1	0	2	0.75

Table IV. : Quality attributes results

Analyzability	Changeability	Testability	Portability	Reliability
20.48	54.01	67.44	13.01	19.87

cause the software was developed ad-hoc without using any software development model and starting directly from the coding phase. For some of the metrics we did not have the acceptable ranges and this is a possible future extension of the paper. Another future work is to develop the same software using a specific software development model that we will find as most suitable and to make a comparison between these results and the quality results for the new developed software.

ACKNOWLEDGMENT

This work was supported, in part, by the European Unions Horizon 2020 research and innovation programme, project Virtual Research Environment for Regional Interdisciplinary Collaboration in Southeast Europe and Eastern Mediterranean VI-SEEM [675121].

REFERENCES

Adewole Adewumi, Sanjay Misra, Nicholas Omoregbe, Broderick Crawford, and Ricardo Soto. 2016. A systematic literature review of open source software quality assessment models. *SpringerPlus* 5, 1 (2016), 1936.

Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. 2011. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 243–252.

Elin A Gursky and Boris Hrečkovski. 2012. *Handbook for Pandemic and Mass-casualty Planning and Response*. Vol. 100. IOS Press.

György Hegedűs, György Hrabovszki, Dániel Hegedűs, and István Siket. 2010. Effect of object oriented refactorings on testability, error proneness and other maintainability attributes. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems*. ACM, 8.

Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE, 30–39.

David E Hogan and Travis Brown. 2014. Utility of vital signs in mass casualty-disaster triage. *Western journal of emergency medicine* 15, 7 (2014), 732.

InfoEther. PMD duplicate code detector. https://pmd.github.io/pmd-5.4.1/usage/cpd-usage.html. (????).

Chris Johnson, Ritesh Patel, Deyanira Radcliffe, Paul Lee, and John Nguyen. 2015. *Establishing Qualitative Software Metrics in Department of the Navy Programs*. Technical Report. DTIC Document.

Capers Jones. 1996. Programming Languages Table, Release 8.2. *Software Productivity Research, Burlington, MA* (1996).

Gurpreet Kaur and Kailash Bahl. 2014. Software reliability, metrics, reliability improvement using agile process. *International Journal of Innovative Science, Engineering and Technology* 1, 3 (2014), 143–7.

Bojana Koteska, Anastas Mishev, and Ljupco Pejov. 2018. Quantitative Measurement of Scientific Software Quality: Definition of a Novel Quality Model. *International Journal of Software Engineering and Knowledge Engineering* 28, 03 (2018), 407–425.

G Uma Maheswari and VV Rama Prasad. 2011. Optimized software quality assurance model for testing scientific software. *International Journal of Computer Applications* 36, 7 (2011).

Thilagavathi Manoharan. 2014. Metrics Tool for Software Development Life Cycle. *vol* 2 (2014), 1–16.

Kagiso Mguni and Yirsaw Ayalew. 2013. An Assessment of Maintainability of an Aspect-Oriented System. *ISRN Software Engineering* 2013 (2013).

Edward MICHLOWICZ. 2013. TPM method in the analysis of flow in the cold rolling mill. In *Conference proceedings 22nd International Conference on Metallurgy and Materials METAL*. 291–296.

Alexander Neidhardt. 2017. *Applied Computer Science for GGOS Observatories: Communication, Coordination and Automation of Future Geodetic Infrastructures*. Springer.

Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 337–344.

Gerard ORegan. 2015. Software Engineering Institute (SEI). In *Pillars of Computing*. Springer, 195–205.

Jiapu Pan and Willis J Tompkins. 1985. A real-time QRS detection algorithm. *IEEE Trans Biomed Eng* 3 (1985), 230–236.

Shiv Prakash and Deo Prakash Vidyarthi. 2015. Maximizing availability for task scheduling in computational grid using genetic algorithm. *Concurrency and Computation: Practice and Experience* 27, 1 (2015), 193–210.

Roger S Pressman. 2005. *Software engineering: a practitioner's approach*. Palgrave Macmillan.

Monika Simjanoska, Bojana Koteska, Ana Madevska Bogdanova, Nevena Ackovska, Vladimir Trajkovik, and Magdalena Kostoska. 2018. Automated triage parameters estimation from ECG. *Technology and Health Care* 26, 2 (2018), 387–390.