A systematic process to define expert-driven software metrics thresholds

Renata Saraiva [1], Mirko Perkusich[1], Hyggo Almeida[1], and Angelo Perkusich[1]

[1]Embedded and Pervasive Computing Laboratory, Federal University of Campina Grande, Campina Grande, Brazil

Abstract

Software metrics are usually used for quantification, not giving the necessary support for decision making. To increase their usefulness, it is necessary to give them meaning through the definition of significant thresholds. Despite its importance, the state of the art on threshold derivation is mostly based on data-driven approaches. This paper presents a systematic approach to define thresholds for metrics in the absence of data and based on eliciting knowledge from experts. The proposed approach is based on identifying context factors that influence the thresholds for a given metric and is supported by fuzzy logic concepts to model the crisp value (i.e., collected data) into a linguistic variable (i.e., interpreted information). We present context factors elicited from three experts for the metrics code coverage, static code analysis warnings count and defect count. Further, we present cases on how to implement the proposed approach. As a result, we conclude that the approach is promising.

***Keywords*—Software metrics; Software thresholds; Fuzzy logic.**

I. Introduction

Despite their potential advantages, software metrics are usually used only for quantification purposes, not giving adequate support for decision-making [1]. For this purpose, it is necessary to give meaning (i.e., semantics) to the metrics through the definition of reference values (i.e., thresholds). Thresholds are values used to set ranges of desirable and undesirable states and indicate anomalies. The absence of thresholds for many software metrics is one of the main reasons for them to not be effectively used in the industry [4].

There are several solutions to calculate software metrics thresholds proposed in the literature [1], [18], [17], [8], [4], [23], [15], [13], [14], [5]. We summarize their characteristics and limitations in Section II. As far as our knowledge, all the proposed solutions are data-driven and focus on source code metrics. For instance, recently, two empirical studies were published comparing the existing data-driven thresholds models to predict faults on open source software [2] and fault proneness [3]. Therefore, if an organization needs to define thresholds for popular management metrics such as *team velocity*, *build status* and *lead time* [9], the literature is scarce in guiding them on how to define representative thresholds for their context. For this purpose, in practice, organizations use expert knowledge to define thresholds using ad hoc processes. For instance, for the metric *code coverage*, many development groups require 85% coverage to achieve quality targets as the status quo [20]. On the other hand, there are several factors that might influence coverage target for a given project, such as product complexity, project criticality, and cost of evolution.

To complement the current state of the art on software metrics threshold derivation, we present a systematic approach to define thresholds for metrics in the absence of data and based on eliciting knowledge from experts. The goal is to support managers when making decisions regarding the interpretation (i.e., semantics) of the collected metrics by developing models that mimics the thought process of humans when making decisions regarding the thresholds.

The proposed approach is based on identifying context factors that influence the thresholds for a given metric and is supported by fuzzy logic concepts to model the crisp value (i.e., collected data) into a linguistic variable (i.e., interpreted information). For instance, consider that the metric *code coverage* is used to decide if enough tests have been executed and, given that the *defect count* is

DOI reference number: 10.18293/SEKE2019-217

low enough, the product can be delivered to the customer. The crisp value of *code coverage* lies in $[0, 100]$. So, for instance, we could map the value 85 to the linguistic variable *OK*, which means that, given this metric, the product should be released. Conversely, we could map the value 50, meaning that the product should not be released.

The research question that we address in this paper is:

How can we define thresholds for software metrics in the absence of data and when the organization context cannot be reduced to an experimental setup?

Our contributions include: (1) a systematic approach for deriving software metrics thresholds in the absence of data; (2) an empirical cyclic process to continuously refine thresholds; and (3) context factors that influence the definition of popular metrics such as *code coverage*, *static code analysis warnings count* and *defect count*. Our systematic approach is demonstrated through the definition of thresholds for popular software metrics.

The remaining of the paper is organized as follows: in Section II, we present works related to deriving thresholds and discuss them in light of our approach, the proposed process; in Section III, we present the proposed approach; and in Section IV, we present our final remarks and future work.

II. Related work

The effective use of software metrics is hampered by the lack of significant thresholds [1]. In the literature, few metrics have defined thresholds. Furthermore, many researchers have proposed different approaches to define them [1], [4], [5], [13], [15], [17], [18], [23].

Alves *et al.* [1] present a method that determines threshold empirically from measurement data (i.e., benchmarking). The method is based on statistical properties of the metric such as scale and distribution. To evaluate their approach, they collected data from 100 object-oriented software systems to calculate thresholds, which were successfully used to assist on software analysis, benchmarking and certification. The main risk of such a solution is to use thresholds to assist decision-making that were calculated for a different context.

In the works of Oliveira et al. [15], [13], the concept of relative thresholds is proposed as well as a tool for extracting these thresholds. Their approach handles the heavy-tailored distribution of source code metrics by complementing absolute thresholds with a percentage of software code entities that must follow it. The technique is validated with an industrial case study. As Alves et al. [1], its limitation is that the calculated threshold and percentage might be dependent on the context.

Ferreira et al. [4] used the EasyFit tool to define the probability distribution with the best fit for the distribution for a given metric. Therefore, if the defined probability distribution had a representative mean value, it was used as the reference value. Otherwise, the distribution is quantified as bad, good or moderate. By analyzing data of forty open source projects, they defined the thresholds for six metrics: LCOM (Lack of Cohesion of Methods), DIT (Depth in Tree), COF (Coupling Factor), afferent couplings, number of public methods, and number of public fields.

In Foucault *et al.* [5], a solution based on statistical methods was presented. This approach is based on (i) *double sampling* [19] to randomly selects projects samples, and (ii) *bootstrap* to estimate the thresholds based on quartiles. Despite the potential of this approach, the validation process was limited to a test to identify the best configuration for the approach itself since, according to the authors, the two statistical methods are widely used.

In Shatnawi [17], a solution based on logarithmic transformation was presented. In this approach, initially, the data is transformed using the natural log, leaving the symmetric data thus closer to a normal distribution. Afterward, a temporary reference value (T') is collected using the mean (M) and standard deviation (SD) so that $T' = M + SD$ or $T' = M - SD$. Finally, the T' is converted to the original distribution by using the exponent function of T', generating the final reference value.

All the presented studies are data-driven and most of them focus on source code metrics. The motivation of our work is to define a systematic process that guides engineers in defining metrics thresholds in the absence of data. In this context, Marinescu [10] presents a guideline to define semantical filtering to support the analysis of source code metrics in the context of detecting design flaws based on the derivation of thresholds. They define two types of thresholds-related filters: marginal and interval. For a marginal filter, it is necessary to define the threshold value and direction, which specifies whether the threshold value is an upper or lower bound. The thresholds are described as design rules or heuristics (e.g., a class should not be coupled with more than 6 other classes). Interval filters are defined as an interval such as "between 20 and 30". Even though Marinescu [10] describes the use of thresholds as a key component on the proposed solution, he does not present a systematic approach to define it. As shown in Section III, we use the classification of thresholds presented in Marinescu [10] and complement their work by proposing a systematic approach to define them.

III. Threshold definition strategy

In this section, we present the proposed approach with running examples. The examples are a result of a pilot study executed at a Brazilian software development

company in which we collected data from 3 project managers, all of them with over 5 years of experience managing projects with support of software metrics. The most important decision they had was defining if a version of the product had enough quality to be released. The three main metrics they used for this purpose were *code coverage*, *static code analysis warnings count* and *defect count*. *Code coverage* was used to indicate if enough tests were performed, which gave them confidence that few defects would be detected only in operation. *Static code analysis warnings count* was used as a measure of the internal quality of the product. The *defect count*, which was only representative if *code coverage* was high enough, gave a snapshot of the current quality of the product. As a status quo on the company, all projects had a lower bound threshold of 80% for *code coverage*, 10 for *static code analysis warnings count*, and 5 for *defect count*.

The main goal of the proposed approach is to provide software engineers with a systematic mechanism to enable them to work with software metrics in a more abstract level through the definition of thresholds in the absence of historical data. Since the goal of using metrics is to support decision-making, this is closer to the real intention in using metrics. An assumption of the proposed approach is that the metrics are valid for their intended purposes [12].

The proposed expert-driven threshold definition strategy is cyclic and composed of three main steps, namely: (i) thresholds characterization, (ii) thresholds modeling, and (iii) thresholds evaluation. An overview of the approach is shown in Figure 1. Our approach can be used to elicit data from a single or multiple experts. There are two main roles: threshold designer and domain experts. The threshold designer is responsible for leading the planning and execution of the threshold definition process by the elicitation of knowledge from the domain experts. The domain experts are responsible for actively participating in the process of defining the thresholds models (steps i and ii) and evaluating the models (steps iii). The domain experts might include the project manager, development lead, test lead, product manager or quality assurance manager.

In what follows, we present details regarding each of the steps of the proposed approach.

A. Step i: thresholds characterization

On the first (i) step, the goal is to characterize the thresholds through the identification of the relevant context factors and the type of the threshold, as defined in Marinescu [10]. First, it is necessary to define the decision scale to be used.

Definition 1 (Metric semantics scale) A metrics semantic scale *is the scale to be used to represent the semantics of a given metric.*

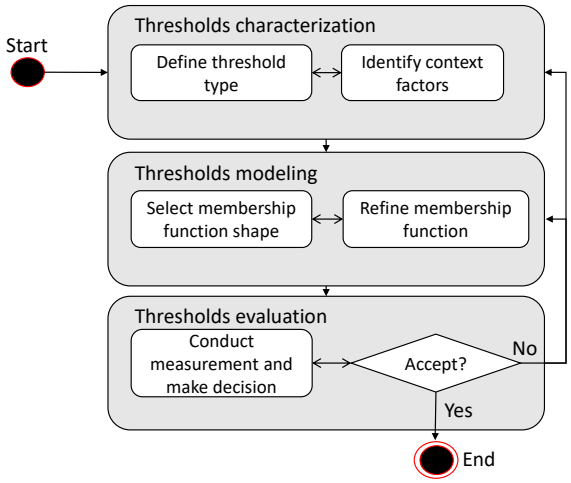

Fig. 1. Proposed approach overview.

The possible types of scale are Boolean and ordinal. For instance, one may use a Boolean scale for *code coverage* with the values: *OK* and *NOT OK*. Another possibility is to use an ordinal scale: *Bad*, *Moderate* and *Good.*

Afterwards, the type of threshold must be defined: marginal or interval. If a marginal threshold is selected, there are two possible types: **HigherThan**(Θ) and **LowerThan**(Θ), where Θ is the reference value. If an interval threshold is selected, by definition is of the type *Between*(α,β), which is equivalent to **HigherThan**(α) $\wedge$ **LowerThan**(β), where α is the lower bound and β is the upper bound.

Next, the context factors must be identified. Context factors are attributes from key entities of the process that might influence the threshold. There are two types of context factors: diminishing factors and enhancing factors. The diminishing factors influence the thresholds values to be lower, and the enhancing factors influence the thresholds values to be higher. For instance, for *code coverage* metric, we elicited the following factors from the experts: *product complexity* and *project criticality* as enhancing factors, while *team experience* and *impact on evolution cost* as diminishing factors. For *static analysis warnings count* and *defect count*, *project criticality* is the enhancing factor. Finally, the factors are ranked in order of relative magnitude of their influence on the thresholds' values.

It is important to notice that the context factors are factors that influence the semantics of the metrics and not the value itself. For instance, one might reason that the factor "volatile requirements" influences the threshold of "defect

count", because the more volatile are the requirements, less time will be available for testing and, consequently, lower will be the "defect count" in the testing phase (not necessarily, the "defect leakage"). On the other hand, even though this might be true, the reference value for the given metric should not be lower, which discards this factor.

In the case that multiple domain experts are involved in the process, the threshold designer must execute the described tasks with each one individually to avoid bias. Afterwards, with all the domain experts together, he presents the identified possibilities of threshold types and context factors and executes a meeting like the Planning Poker [7], in which each expert holds two cards: Agree and Disagree. The threshold designer mediates the meeting by enabling a structured discussion regarding a consensus of the threshold types and context factors to be considered in the models. As in a Planning Poker meeting, candidate solutions must be individually voted by experts, by turning their card simultaneously, until a consensus is reached by all the experts.

B. Step ii: thresholds modeling

After defining the type of threshold and rank the context factors, the experts will have a better understanding regarding the semantics of the given metrics. Since our approach is based on concepts of fuzzy logic, the thresholds are modeled as a linguistic variable, which is "a variable whose values are words or sentences in a natural or artificial language" [22]. For this purpose, we map the metric semantic scales defined in step i as the term set to be used for the linguistic variable. So, for instance, we could have the linguistic variable *code coverage* (c) composed of the terms *{OK, Not OK}*, in the case of a Boolean scale.

The main goal of this step is to fuzzify the crisp values of a metric into fuzzy linguistic terms. For this purpose, it is necessary to define the membership functions. There are several types of membership functions that can be used. In Figure 2, we show six popular types of functions.

A membership function must be defined for each term in the given linguistic variable. So, for a metric with a Boolean scale, two membership functions must be defined.

The main challenge is to define the parameters for the membership functions. For this purpose, given the type of function and the magnitude of the impact of the context factors identified in step i, the threshold designer might show the experts possible shapes for the membership functions to guide them. For instance, if the threshold is marginal, probably the experts could choose a z-shape, sigmoid, or s-shape as a reference. If the threshold is interval, the experts could choose triangular, trapezoidal or Gaussian. For instance, for the *code coverage*, we could use the Gaussian shape.

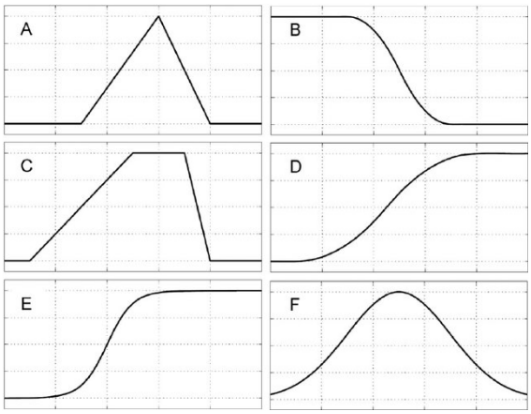

Fig. 2. Six types of fuzzy membership functions: (A) triangular, (B) z-shape, (C) trapezoidal, (D) s-shape, (E) sigmoid and (F) Gaussian, [6].

TABLE I. "What if" scenarios for *Code coverage* **with a verbal scale**

Code coverage crisp value	Not OK	OK
10	Certain	Impossible
20	Certain	Impossible
30	Certain	Impossible
40	Probable	Improbable
50	Expected	Uncertain
60	Fifty-fifty	Fifty-fifty
70	Uncertain	Expected
80	Improbable	Probable
90	Uncertain	Expected
100	Probable	Improbable

Afterwards, the experts should use their experience from past projects to define "what-if" scenarios to guide them in configuring the functions, in which for a set of values, they would indicate the probability of it being mapped to each of the possible terms. For this purpose, instead of directly defining the probabilities, they could use a verbal scale such as the one presented in Renooij and Witteman [16]. For instance, for *code coverage*, the scenarios shown in Table I could be used:

Afterwards, the verbal scale is converted to a numerical scale following the rules presented in Renooij and Witteman [16]. As a result, we have the values presented in TableII.

Given this, an algorithm can be used to fit the data elicited from the experts for each linguistic term into the appropriate distribution. For instance, in Figure 3, we present a fit for the membership function for the term *Not OK* using the Akima Cubic Spline. Finally, the expert can analyze visually the resulting distribution and judge if the reflects his intuition (i.e., face validity). Otherwise, they must reflect on the inconsistencies and the step should restart.

TABLE II. "What if" scenarios for *Code coverage* with a numerical scale

Code coverage crisp value	Not OK	OK
10	100	0
20	100	0
30	100	0
40	85	15
50	75	25
60	50	50
70	25	75
80	15	85
90	25	75
100	85	15

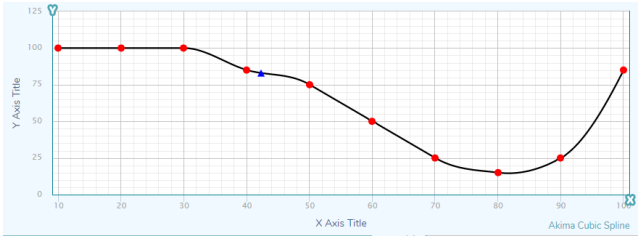

Fig. 3. Curve fit for *Code Coverage*'s linguistic term *Not OK*.

As a rule of thumb, it is preferable to set thresholds that are more conservative, since it is better to get more false positive results, rather than missing an important issue due to a very strict threshold value. The threshold can be refined during the empirical cycle which is executed on step iii.

As in step ii, this step should be, initially, executed individually with each expert. Afterwards, consensus must be achieved between all the experts in a Planning Poker-style meeting.

C. Step iii: thresholds evaluation

At this point, the thresholds models are defined, but as in other expert-driven processes [11], [21] it is necessary to have an empirical cycle in which decisions based on the thresholds are analyzed to evaluate the models. For this purpose, the threshold designer should schedule meetings according to the project's context. For agile projects with short-term releases, a meeting could be held every iteration or two. During the meeting, along with the threshold designers, the domain experts that participated on steps i and ii should participate to discuss the results of using the models.

Assuming that the metrics are valid for their intended purposes, if the model's results are not consistent with the reality, there are three possible outcomes: (1) exception case, (2) scope limitation, and (3) model needs refinement. For the first outcome, it is possible that, for instance, the developed model indicates with 90% that the product should be released, but the release is a failure. This might occur due to a rare case for the given organization, which might be the case of an unexpected success of the product causing overload on the server. In this case, the experts could decide that the model is reliable, and the bad decision was caused by the uncertainty inherent in the process.

For the second possible outcome, a bad decision of releasing a product might have been caused by a failure of requirements elicitation such as missing an important non-functional requirement (e.g., number of requests per second). For this case, the experts can assume that the decisions based on the model assume that the product's requirements are complete and that this case is out of the scope of the model.

For the third case, the experts might decide that the model needs refinement, because they failed to, for instance, consider an important context factor or a better suited membership function. Independent of the outcome, the execution of step iii should be considered a mandatory activity in the measurement program.

IV. Conclusions

In this paper, we presented a systematic process to define thresholds for metrics in the absence of data and based on eliciting knowledge from experts. The proposed approach is based on identifying context factors that influence the thresholds for a given metric and is supported by fuzzy logic concepts to model the crisp value (i.e., collected data) into a linguistic variable (i.e., interpreted information). It is cyclic and composed of three main steps, namely: (i) thresholds characterization, (ii) thresholds modeling, and (iii) thresholds evaluation.

Our contributions are threefold: (1) a systematic approach for deriving software metrics thresholds in the absence of data; (2) an empirical cyclic process to continuously refine thresholds; and (3) context factors that influence the definition of popular metrics such as *code coverage*, *static code analysis warnings count* and *defect count*.

For further research, we will expand the proposed approach to handle the case of having multiple metrics used for a single decision. Furthermore, we will execute a case study to empirically evaluate the proposed approach in terms of practical utility.

References

[1] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, sep 2010.

[2] O. F. Arar and K. Ayan. Deriving thresholds of software metrics to predict faults on open source software. *Expert Syst. Appl.*, 61(C):106–121, Nov. 2016.

[3] A. Boucher and M. Badri. Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology*, 96:38–67, 2018.

[4] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, feb 2012.

[5] M. Foucault, M. Palyart, J.-R. Falleri, and X. Blanc. Computing contextual metric thresholds. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 1120–1125, New York, New York, USA, mar 2014. ACM Press.

[6] M. U. Guide. Matlab cd-rom, mathworks, 2007.

[7] N. C. Haugen. An empirical study of using planning poker for user story estimation. In *AGILE 2006 (AGILE'06)*, pages 9–pp. IEEE, 2006.

[8] S. Herbold, J. Grabowski, and S. Waack. Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering*, 16(6):812–841, 2011.

[9] E. Kupiainen, M. V. Mäntylä, and J. Itkonen. Using metrics in agile and lean software development–a systematic literature review of industrial studies. *Information and Software Technology*, 62:143–163, 2015.

[10] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE, 2004.

[11] E. Mendes. Expert-based knowledge engineering of bayesian networks. In *Practitioner's Knowledge Representation*, pages 73–105. Springer, 2014.

[12] A. Meneely, B. Smith, and L. Williams. Validating software metrics: A spectrum of philosophies. *ACM Trans. Softw. Eng. Methodol.*, 21(4):24:1–24:28, Feb. 2013.

[13] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik. Rttool: A tool for extracting relative thresholds for source code metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 629–632. IEEE, 2014.

[14] P. Oliveira, M. T. Valente, A. Bergel, and A. Serebrenik. Validating metric thresholds with developers: An early result. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 546–550. IEEE, 2015.

[15] P. Oliveira, M. T. Valente, and F. P. Lima. Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 254–263. IEEE, feb 2014.

[16] S. Renooij and C. Witteman. Talking probabilities: communicating probabilistic information with words and numbers. *International Journal of Approximate Reasoning*, 22(3):169 – 194, 1999.

[17] R. Shatnawi. Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process*, 27(2):95–113, feb 2015.

[18] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1):1–16, jan 2010.

[19] S. K. Thompson. Simple random sampling. *Sampling, Third Edition*, pages 9–37, 2012.

[20] T. Williams, M. Mercer, J. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No. 01CH37179)*, pages 420–424. IEEE, 2001.

[21] B. Yet, Z. Perkins, N. Fenton, N. Tai, and W. Marsh. Not just data: A method for improving prediction with knowledge. *Journal of Biomedical Informatics*, 48:28 – 37, 2014.

[22] L. A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning—i. *Information sciences*, 8(3):199–249, 1975.

[23] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. How Does Context Affect the Distribution of Software Maintainability Metrics? In *2013 IEEE International Conference on Software Maintenance*, pages 350–359. IEEE, sep 2013.