Automatic Calibration of Performance Indicators for Performance Analysis in Software Development

Mushtaq Raza
INESC TEC, Porto, Portugal/ Department of Computer Science
Abdul Wali Khan University Mardan
Mardan, Pakistan
mushtaq.raza@fe.up.pt

João Pascoal Faria
INESC TEC/ Faculty of Engineering, University of Porto
Rua Dr. Roberto Frias
Porto, Portugal
jpf@fe.up.pt

***Abstract*—ProcessPAIR is a novel method and tool for automating the performance analysis in software development. Based on performance models structured by process experts and calibrated from the performance data of many developers, it automatically identifies and ranks potential performance problems and root causes of individual developers. However, the current calibration method is not fully automatic, because, in the case of performance indicators that affect other indicators in a conflicting way, the process expert has to manually calibrate the optimal value in a way that balances those impacts. In this paper we propose a novel method to automate this step, taking advantage of training data sets. We demonstrate the feasibility of the method with an example related with the Code Review Rate indicator, with conflicting impacts on Productivity and Quality.**

***Index Terms*—automatic performance analysis, personal software process, Performance analysis tool**

I. Introduction

Process and product data produced in software development projects can be periodically analyzed to identify performance problems, determine their root causes and devise improvement actions. However, conducting the analysis manually is challenging because of the potentially large amount of data to analyze, the effort and expertise required, and the lack of benchmarks for comparison.

ProcessPAIR is a novel method and tool for automated performance analysis and improvement recommendation in software development [8]. Based on performance models defined by process experts and calibrated from the performance data of many projects, it automatically identifies and ranks potential performance problems and root causes of individual entities (developers, teams or organizations), so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused. ProcessPAIR was successfully applied in education and training environments [9]. ProcessPAIR is also of interest to high-maturity organizations (CMMI maturity levels 4 and 5), because it facilitates the implementation of practices of the Organizational Process Performance (ML4) and Organizational Performance Management (ML5) process areas [3].

However, the current calibration method used by ProcessPAIR is not fully automatic, because the optimal value of each performance indicator must be provided by the process expert. In many cases, the optimal value follows directly from the definition and is located in one extreme of the scale (minimum or maximum). But in the case of performance indicators that affect other indicators in a conflicting way, an intermediate optimal value that balances those impacts need to be manually calibrated by the process expert. An example is the Code Review Rate (size unit reviewed per time unit). If code reviews are performed too fast, quality of reviews (Code Review Yield) is negatively affected, but if they are performed too slow, Productivity is negatively affected, and it is not easy to choose a review rate that balances these two conflicting impacts.

In this paper we propose a novel method to automate this step, and hence fully automate the model calibration process, taking advantage of training data sets. We show the feasibility of the method with a data set that includes Code Review Rate, Productivity and Code Review Yield data.

The article is organized as follows. Section II presents background information about ProcessPAIR. Related work is presented briefly in Section III. Section IV presents the proposed method and feasibility study. Section V concludes the article and points directions for future work.

II. Background

A. The ProcessPAIR Approach

The ProcessPAIR approach involves three main steps (see Figure 1):

1) *Define:* Process experts define the structure of a performance model (PM) suited for the development process under consideration. In our approach, a PM comprises a set of top-level and child performance indicators (PIs), organized hierarchically by cause-effect relationships [7].
2) *Calibrate (or Learn):* The PM is automatically calibrated by ProcessPAIR based on the performance data of many process users. The statistical distribution of each PI and statistical relations between PIs are computed from the training dataset, taking advantage of statistical and machine learning techniques [7].
3) *Analyze:* Once a PM is defined and calibrated, the performance data of individual entities can be automatically

DOI reference number: 10.18293/SEKE2019-188

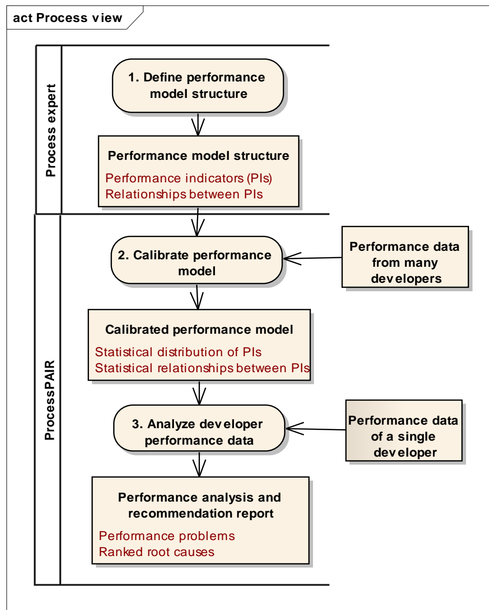

Fig. 1. UML activity diagram depicting the main activities and artifacts in the ProcessPAIR method.

analyzed by ProcessPAIR, to identify performance problems (in top-levels PIs), identify potential root causes (related with child PIs), and rank those potential root causes.

The ProcessPAIR approach is supported by the ProcessPAIR tool, freely downloadable from https://blogs.fe.up.pt/processpair/. The tool is implemented as a standalone Java application, in order to protect the users data. It has a core framework and extensions for the processes of interest. An extension for the Personal Software Process (PSP), containing the definition of performance models for the PSP and data loaders from the most relevant project management tools used by PSP developers, was developed for education and training environments, but other extensions can be easily developed for other processes and contexts.

Further details about each step are given next.

B. Model Definition

The first step in our approach is the definition of the following elements of the PM:

- list of relevant PIs, including formulas for their computation from base measures, and the definition of the optimal value of each PI;
- subset of top-level PIs;
- cause-effect relationships between PIs, determined by a formula or statistical evidence;
- sensitivity coefficients [10] between PIs related by a formula (needed for ranking the identified root causes in the performance analysis step).

C. Model Calibration

The PM is automatically calibrated by ProcessPAIR from training data sets, generating the following data:

- approximate statistical distribution (cumulative distribution function) of each PI in the training data set;
- recommended performance ranges for each PI, needed for classifying values of each PI of a subject under analysis into three semaphores: green - no performance problem; yellow - a possible performance problem; red - a clear performance problem. Such ranges are calibrated automatically from the training data, so that there is an approximately even distribution of data points by the semaphores. In particular, the green range corresponds to the 1/3 data points closest to the optimal value, and the red range corresponds to the 1/3 data points farthest to the optimal value;
- regression models and sensitivity coefficients between PIs not related by a formula. Sensitivity coefficients between PIs not related by a formula are computed by first determining a regression model from the calibration dataset (a piecewise linear model organized as a regression tree [1]), and subsequently computing the corresponding sensitivity coefficient.

Some results of model calibration can be consulted in Figure 2. The example refers to the Code Review Rate, here named as Code Review Productivity. The approximate statistical distribution (cumulative distribution function) of this performance indicator, calibrated automatically by ProcessPAIR based on a training data set, is shown on the bottom left side. The 'green' and 'yellow' performance ranges are shown on the right; these ranges are calibrated automatically by ProcessPAIR, based on the commutative distribution function and the optimal value (calibrated manually by the process expert). The data points in the chart on the right show the values of this performance indicator for a series of projects under analysis. Different performance indicators defined in the performance model can be consulted in the tree view on the top left side. The Code Review Productivity has a green semaphore because its values lie mostly inside the green range.

D. Performance Analysis

Having defined and calibrated the PM, the performance data of individual entities (developers, teams or organizations) can be automatically analyzed by ProcessPAIR, to identify and rank performance problems and potential causes [7].

To rank the identified causes (child PIs) of performance problems in top-level PIs, it is used a ranking coefficient, that combines a sensitivity coefficient (measuring the impact of improving child PIs on top-level PIs) and a so-called percentile coefficient (measuring the difficulty of improving the child PIs).

The percentile coefficient is computed based on the distance of the observed values to the optimal value of each PI.

Hence the choice of optimal value has impact on both problem identification and root cause identification and prioritisation.

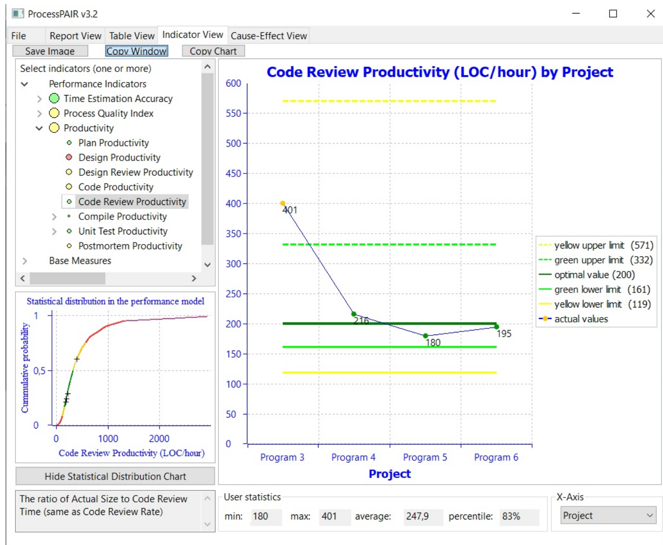

Fig. 2. ProcessPAIR user interface.

III. Related Work

A. Optimal Code Review Rate

According to [4] [12], the time spent in reviewing a work product in relation to its size (review rate) is a leading indicator of the review yield (percentage of defects found).

In a published study [5], the recommended review rate of 200 lines of code (LOC) per hour or less was found to be an effective rate, identifying nearly two-thirds of the defects in design reviews and more than half in code reviews.

A team using the Team Software Process (TSP) obtained a process performance model (PPM) for establishing a target code review rate (number of lines of code reviewed per hour), based on the predicted impact on the code review yield (percentage of defects found in reviews), characterized as [12]:

- Regression equation: $CodeReviewYield = 146 - 0.364 \times CodeReviewRate$
- $R2 = 94.1\%, p - value = 0.000$

According to this regression equation, the smaller the code review rate, the higher is the predicted code review yield (that, anyway, by definition, cannot exceed 100).

However, the quantitative impact on overall productivity was not analysed in those studies.

B. Productivity Measurement

Software development productivity is usually measured in function points per time unit or lines of code (LOC) per time unit [13] [6] [11]. However, both productivity measurement techniques have some limitations. On one hand, the measurement of function points remains subjective even after the completion of the software development project. On the other hand, productivity measures based on LOC have limitations due to the lack of counting standards and the dependence on the programming language [2].

In the data set we will explore for automatic calibration of the optimal value, there is no information about function points, only size and time. The training data set contains data from more than 3000 individuals that developed the 10 projects of the standard PSP training (the same projects for all individuals, but with varying programming languages). Hence, we will take the average effort per project as a proxy for the functional complexity of each project, and calculate the individual productivity as the ratio between the functional complexity of the projects and the actual time (hours) spent by that individual.

IV. Proposed Method and Results

A. Method

Let us assume that a child PI X (such as the Code Review Rate) has conflicting impacts on two or more parent PIs Y_1, Y_2, ..., Y_n (such as the Code Review Yield and Productivity).

The first step is to analyse the impact of the child PI X on a parent PI Y_i at a time, represented as a function f_i from X to Y_i. In order to arrive at a smooth function, we derive that function as follows: for each candidate optimal value x of X, we compute the mean value of Y in the data points that have the value of X within the green range corresponding to x.

Formally, denoting by S the training data set, p a data point in S, $Y_i(p)$ the value of Y_i in p, $X(p)$ the value of X in p, F the cumulative distribution function of X in S, and F^{-1} the inverse of F,

$$f_i(x) \triangleq mean\{Y_i(p)|p \in S \cdot X(p) \in Green(x)\}$$

with

$$Green(x) = [F^{-1}(\frac{2}{3}F(x)), F^{-1}(\frac{2}{3}F(x) + \frac{1}{3})]$$

In the second step, we compute a combined impact function f_c, as a normalized average of the previous functions:

$$f_c(x) \triangleq mean\{\frac{f_i(x)}{max(f_i)}|i = 1, ..., n\}$$

The values of this function are adimensional values in the 0-1 scale.

Finally, we choose the value x of X that maximizes $f_c(x)$.

All the filtering procedures and calculations can be fully automated.

We implemented the calculations in a prototype tool taking advantage of evolutionary algorithms (genetic algorithms) to solve the optimization problem in a way that can scale to large data sets.

B. Results

In this study, for automatic calibration, we used a PSP data set available from the Software Engineering referring to 31,140 projects concluded by 3,114 engineers during 295 classes of the classic PSP for Engineers I/II training courses running between 1994 and 2005. In this training course, targeting professional developers, each engineer develops 10 small projects, following increasingly sophisticated process

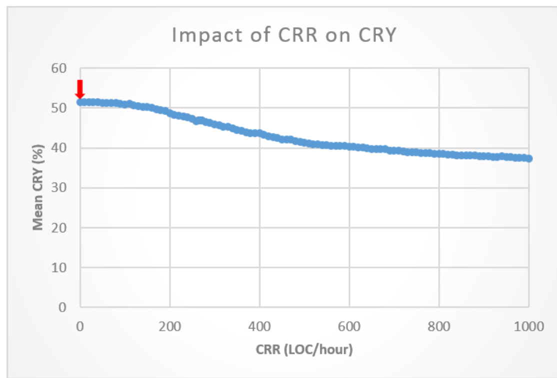

Fig. 3. Impact of CRR on CRY

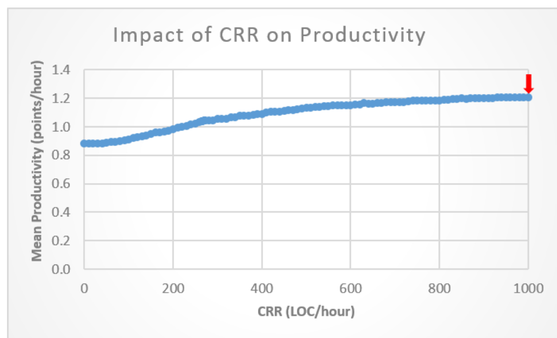

Fig. 4. Impact of CRR on Productivity

variants (PSP0, PSP1, etc.). Since code reviews are introduced only in the third project, we excluded the data points with zero time spent in Code Reviews. Since the Code Review Yield is undefined in case of 0 defects entering the Code Review phase, we also excluded data points with undefined Code Review Yield. In the end, we selected 9,650 data points (each corresponding to a project developed by a developer). Based on the selected data points, we computed the impact functions for the case of Code Review Rate, impacting Productivity and Code Review Yield. The resulting curves are presented in Figures 3, 4 and 5.

Figure 3 shows that, as expected, higher values of Code Review Rate are associated with lower values of Code Review Yield, which declines more significantly in the 200-500 LOC/hour range.

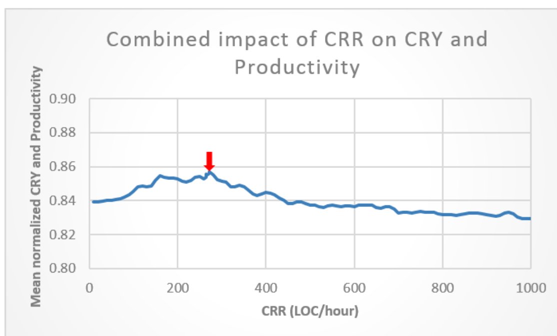

Fig. 5. Combined impact of CRR on CRY and Productivity

Figure 4 shows that, as expected, higher values of Code Review Rate are associated with higher values of Productivity, with a more significant increase in the 100-600 LOC/hour range.

Figure 5 shows the combined impact of Code Review Rate on Code Review Yield and Productivity. The combined curve has some oscillations due to the close symmetry of the two component curves, with a peak value at 270 LOC/hour.

Hence, the computed optimal value is 270 LOC/hour. This value is a bit higher than the literature recommendation of 200 LOC/hour, but perhaps closer to common practice when productivity impact is also important.

V. Conclusions

The method proposed in this paper worked successfully for the case study presented, allowing the automatic calibration of the optimal value and range of a PI (Code Review Rate) with conflicting impacts on other PIs (Code Review Yield and Productivity). The derived optimal value (270 LOC/hour) is a bit higher than the literature recommendation (200 LOC/hour), which is justified by the fact that we are quantitatively analyzing not only the impact on review effectiveness (yield), but also on productivity.

As future work, we intend to implement the calibration method in the ProcessPAIR tool in order to automatically calibrate all the PIs with conflicting impacts on high-level PIs.

References

[1] L. Breiman. *Classification and Regression Trees*. The Wadsworth statistics/probability series. Wadsworth International Group, 1984.

[2] David N Card. The challenge of productivity measurement. In *Pacific Northwest Software Quality Conference*, pages 1–10, 2006.

[3] Mary Beth Chrissis, Mike Konrad, and Sandra Shrum. *CMMI for development: guidelines for process integration and product improvement*. Pearson Education, 2011.

[4] Watts S Humphrey. *Psp (sm): a self-improvement process for software engineers*. Addison-Wesley Professional, 2005.

[5] Chris F Kemerer and Mark C Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE transactions on software engineering*, 35(4):534–550, 2009.

[6] Katrina D Maxwell and Pekka Forselius. Benchmarking software development productivity. *Ieee Software*, 17(1):80–88, 2000.

[7] M. Raza and J. P. Faria. A model for analyzing performance problems and root causes in the personal software process. *J. Softw. Evol. Process*, 28(4):254–271, April 2016.

[8] Mushtaq Raza and João Pascoal Faria. Processpair: A tool for automated performance analysis and improvement recommendation in software development. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 798–803, New York, NY, USA, 2016. ACM.

[9] Mushtaq Raza, João Pascoal Faria, and Rafael Salazar. Assisting software engineering students in analyzing their performance in software development. *Software Quality Journal*, pages 1–29, 2019.

[10] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[11] Goparaju Purna Sudhakar, Ayesha Farooq, and Sanghamitra Patnaik. Measuring productivity of software development teams. 2012.

[12] Shurei Tamura. Integrating cmmi and tsp/psp: using tsp data to create process performance models. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2009.

[13] Stefan Wagner and Melanie Ruhe. A systematic review of productivity factors in software development. *language*, 1989, 1980.