IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 7, 2020, accepted July 26, 2020, date of publication July 31, 2020, date of current version August 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3013328

The ProcessPAIR Method for Automated Software Process Performance Analysis

MUSHTAQ RAZA[1,2] AND JOÃO PASCOAL FARIA[1,3], (Member, IEEE)
[1]Institute for Systems and Computer Engineering, Technology and Science (INESC TEC), 4200-465 Porto, Portugal
[2]Department of Computer Science, Abdul Wali Khan University Mardan, Mardan 23200, Pakistan
[3]Faculty of Engineering, University of Porto, 4200-465 Porto, Portugal

Corresponding author: Mushtaq Raza (mushtaq.raza@fe.up.pt)

This work was supported in part by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalization COMPETE 2020 Programme under Project POCI-01-0145-FEDER-006961, and in part by the National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project UIDB/50014/2020 and Project SFRH/BD/85174/2012.

ABSTRACT High-maturity software development processes and development environments with automated data collection can generate significant amounts of data that 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. In this article, we present ProcessPAIR, a novel method with tool support designed to help developers analyze their performance data with higher quality and less effort. Based on performance models structured manually by process experts and calibrated automatically from the performance data of many process users, it automatically identifies and ranks performance problems and potential root causes of individual subjects, so that subsequent manual analysis for the identification of deeper causes and improvement actions can be appropriately focused. We also show how ProcessPAIR was successfully instantiated and used in software engineering education and training, helping students analyze their performance data with higher satisfaction (by 25%), better quality of analysis outcomes (by 7%), and lower effort (by 4%), as compared to a traditional approach (with reduced tool support).

INDEX TERMS Process improvement, performance analysis, performance model, software process.

I. INTRODUCTION

According to Boehm [1], the top two software engineering challenges are the increasing emphasis on rapid development and adaptability, and the increasing software criticality and need for assurance. The need to ensure the quality of software products in a cost-effective way drives companies and organizations to seek to improve their software development process, as it is becoming more and more accepted in industrial production in general and in the software industry in particular that the quality of the process directly affects the quality of the product [2].

Process improvement initiatives, either at the individual or organizational level, should be driven by performance data and performance objectives. As noted in [3], a paradigm shift is needed from ''process improvement leads to performance improvement'' to ''performance is the primary driver of process improvement''. In fact, analyzing process performance data to determine potential areas for improvement is a required practice for organizations that implement the industry-standard CMMI model at maturity levels 4 and 5 [4]. On the other hand, the increasing adoption of sensor-based data collection in modern development environments [5], and the growing aggregation of performance data in cloud-based project management and application lifecycle management tools, open new opportunities for applying performance analysis practices in broader contexts, not limited to high-maturity organizations.

A. PROBLEM STATEMENT

Performance data, collected in the scope of high-maturity software development processes and development environments with sensor-based data collection, can be periodically analyzed by developers and organizations to identify performance problems, determine their root causes and devise improvement actions [6]. However, the manual analysis

The associate editor coordinating the review of this manuscript and approving it for publication was Resul Das.

of performance data for identifying performance problems, determining their root causes, and devising improvement actions is challenging because of the amount of data to analyze [6], the effort and expertise required, and the lack of benchmarks for comparison. Although tools exist to automate data collection and help in problem identification and interactive performance analysis (e.g., [5], [7]), practically no tool support exists for automated comparison with benchmarks and root cause analysis.

B. OBJECTIVE

To overcome that problem, our main research goal is to develop methods and tools for automating the analysis of performance data produced in the context of software development processes for determining performance problems and their root causes and devising improvement actions. Our research hypothesis is that, by taking advantage of performance models derived from the performance data of many process users, it is possible to automatically analyze the performance data of individual developers and identify and rank performance problems and their root causes, reducing manual effort and errors in performance analysis, and improving user satisfaction.

C. CONTRIBUTION

This article presents the results of such research work: a novel method with tool support, named ProcessPAIR (short-name for Process Performance Analysis and Improvement Recommendation), designed to help developers and organizations analyze their performance data with less effort, by automatically identifying and ranking performance problems and potential root causes, so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused. The analysis is based on a performance model (PM) that is structured manually by experts in the process under consideration and calibrated automatically from the performance data of many process users. The ProcessPAIR method is supported by the ProcessPAIR tool, freely available at `https://blogs.fe.up.pt/processpair/`. To our knowledge, the automated comparison with benchmarks and the automated causal analysis, in the context of process performance analysis, are novel features of our approach.

In previous publications, we presented the derivation of an example performance model [8] (without automatic calibration), a previous tool prototype [9], and a validation experiment [10]. The *specific contribution* of this article is the presentation of an *inside view* of the ProcessPAIR method, detailing its main steps (model definition, model calibration, and performance analysis), with the help of (meta)models and examples.

D. OUTLINE

The rest of the article is organized as follows. Background and related work are presented in Section 2. Section 3 presents the overall approach. Sections 4, 5, and 6 explain the three steps of the method (model definition, model calibration, and performance analysis). Experimental results regarding the effectiveness of the ProcessPAIR method and tool are reviewed in Section 7. Section 8 concludes the paper.

II. BACKGROUND AND RELATED WORK

In this section, we start by reviewing some performance measurement and analysis approaches in software engineering. Afterwords, we review specific techniques that can be used for the identification, visualization, and ranking of performance problems and their root causes for processes and performance improvement.

A. PERFORMANCE MEASUREMENT AND ANALYSIS IN SOFTWARE ENGINEERING

The industry-standard CMMI V2.0 maturity model [3] acknowledges a paradigm shift from ''process improvement leads to performance improvement'' to ''performance is the primary driver of process improvement'', i.e., process improvement actions should be based on performance measurement and analysis. Amongst the set of practices proposed within the ''Managing Performance and Measurement (MPM)'' practice area, the following practices at levels 4 and 5 are closely related to our work:

- MPM 4.3 Use statistical and other quantitative techniques to: develop and analyze process performance baselines and keep them updated;
- MPM 4.4 Use statistical and other quantitative techniques to develop and analyze process performance models and keep them updated;
- MPM 5.2 Analyze performance data using statistical and other quantitative techniques to determine the organization's ability to satisfy selected business objectives and identify potential areas for performance improvement.

In our work, we aim at developing methods and tools to support these practices in a semi-automated way.

Performance measurement and analysis practices can be applied not only at the organizational level, but also at the personal and team level, as illustrated by the Personal Software Process (PSP) [11] and the Team Software Process (TSP) [12]. Although the ProcessPAIR method and tool are process independent, in this article we illustrate their application for the PSP, for the following reasons: it has a well-defined measurement framework; there are curated data sets available with historical PSP data; periodical performance analysis is a recommended practice in the PSP; existent tools support automated metrics collection (e.g., [5]) and interactive performance analysis (e.g., [7]), but do not provide automated comparison with benchmarks and causal analysis.

Agile methods are the dominant paradigm in software engineering industry and education, often combined with elements from more classical methods [13]. An example of metrics collection and analysis in agile project courses is described in [14]. Some metrics are introduced to measure the success of three key workflows (merge management,

continuous integration and continuous delivery), giving instructors and project leaders a quick overview of the project status and problems which they can react upon. The main differences with respect to our research work are: in our research work the goal is to retrospectively analyze completed projects; we try to identify and rank potential causes; for problem identification, we use thresholds derived from large data sets. Nevertheless, the metrics introduced by the authors could be adapted and explored in our approach.

B. PROBLEM IDENTIFICATION TECHNIQUES

1) CONTROL CHARTS

In Statistical Process Control (SPC), control charts are a kind of run charts used to graphically represent the behavior over time of variables that characterize process performance (i.e., process performance indicators [15], [16]) and help to assess process stability and capability. Stability has to do with the level of variability in the variable under analysis, and is assessed with the help of lower and upper control limits. Capability has to do with the ability to meet desired performance levels [17], usually depicted by means of lower and upper specification limits. In our approach, we use run charts to display the series of data points for each performance indicator (PI) under analysis, together with thresholds derived from training data, used as specification limits (see example in Figure 8).

2) BENCHMARK BASED SOFTWARE EVALUATION

In order to enable the automated identification of performance problems associated with relevant PIs, one has to decide on the relevant thresholds. One approach for defining such thresholds is the benchmark-based approach described in [18], [19] to rate the maintainability of software products. The authors claim that the effective use of software metrics is hindered by the lack of meaningful thresholds. They also note that thresholds have been proposed for a few metrics only, mostly based on expert opinion and a small number of observations, or systematically derived based on unjustified assumptions about the statistical properties of the metrics (such as normality). Consequently, they propose a method to empirically derive in a systematic way metric thresholds from measurement data (benchmarks), in order to determine risk profiles and maintainability ratings for products under analysis. They propose a discrete rating schema (from 1 to 5 stars), based on thresholds that correspond to the 20%, 40%, 60% and 80% quantiles.

We adapted their approach for process evaluation (instead of product evaluation), using a similar rating (or classification) scheme, but with three levels only (and thresholds for the 33.(3)% and 66.(6)% quantiles), corresponding to the green, yellow and red colors. In our case, benchmarks are derived from the performance data of a large community of process users.

C. ROOT CAUSE IDENTIFICATION AND RANKING TECHNIQUES

After identifying performance problems, it is important to find their root causes, so that improvement actions can subsequently be defined to address the relevant causes. In this section, several techniques that can be used for identifying root causes are presented.

1) FISHBONE DIAGRAMS

Fishbone diagrams [20] (also called Cause-and-Effect diagrams) are a classic technique for helping in manual causal analysis and for visualizing relationships between causes and effects. In our approach, we present diagrammatically the relationships between causes and effects in the ''Cause-Effect View''.

2) DEFECT CAUSAL ANALYSIS

An example of applying Fishbone techniques in software engineering is Defect Causal Analysis (DCA). Many process improvement approaches (e.g., Six Sigma [21] or FMEA [22]) described in the literature and practiced in the industry include causal analysis activities for determining the causes of defects and other problems [23]. However, most of the techniques are essentially manual. Defect Causal Analysis [24] is one of the prominent methods for analyzing defects and identifying root causes for improvement in software engineering. Furthermore, the learning capability of DCA from defects enables improvement of processes and products, which is a significant benefit in the context of continuous improvement strategies [25].

The DCA process involves 6 steps to be performed in DCA workshops [24]: (1) select problem sample; (2) classify selected defects (e.g., using ODC [26]); (3) identify systematic errors (e.g., with Pareto charts); (4) determine main causes (e.g., using Fishbone diagrams); (5) develop action proposals; (6) document meeting results.

The main problem of DCA is that it is basically a manual process, and our goal is to automate, at least partially, the root causes identification of performance problems. The idea is to automatically drill down from performance problems to causes up to the level permitted by the data available. Manual analysis may still be required for identifying deeper causes not apparent in the available data. Next, we investigate a technique that can help in automatic causal analysis.

3) PROCESS PERFORMANCE MODELS

In order to be able to automatically identify and rank (prioritize) the causes of performance problems, we need to have some quantitative relationship between factors (causes) and outcomes (effects). For that purpose, the CMMI proposes the usage of process performance models (PPM) [4].

In the context of the CMMI process improvement framework, a PPM is a description of the relationship among attributes of a process or sub-process and its outcomes,

developed from historical process performance data and calibrated using collected process and product measures [4].

In the case of continuous variables, a PPM often takes the form of a regression equation, relating controllable or uncontrollable factors (x) with outcomes (y), together with an indicator of the degree of variability in the model, such as the R^2 statistic. In the case of discrete variables, PPMs may be based on Bayesian networks [27].

PPMs are useful tools for project management and process management and improvement. In the latter case, PPMs help organizations identify and leverage important relationships among process factors and outcomes, and estimate (predict) the effects of alternative process changes. The creation of a PPM usually involves the following steps, among others: (1) decide what outcomes to analyze; (2) hypothesize factors to investigate; (3) select the modeling techniques to use; (4) obtain relevant data; (5) fit the model to the data and evaluate the degree of fitness according to statistical and business criteria [27], [28].

PPMs can be applied in our work with adaptations. All the steps discussed above for the creation of PPMs will be applied, with some adaptations and choices: in step 2, we follow a hierarchical approach (factors that can, in turn, be affected by other factors, like in Fishbone diagrams); in step 3, we use a regression model in case of a statistical relationship between factors and outcomes, and an exact formula when there is an algebraic relationship. The types of regression models applicable to our work are discussed in section V-C.

As mentioned before, PPMs can be used to estimate (predict) the effects of alternative process changes. In our case, PPMs can be used to rank the factors, according to the effect of changes in single factors on the outcome under analysis. Since our goal is just to rank the factors, sensitivity analysis techniques can be used. Sensitivity analysis techniques are discussed in section V-C.

4) REGRESSION MODELS

Regression analysis is a statistical process that attempts to determine the strength of the relationship between a dependent variable and one or more independent variables [29]. Traditionally, regression techniques are categorized as linear or nonlinear. Generally, when using linear regression models, it is simple to interpret the relationship between dependent variable and predictors and analyze the correlation among predictors [30]. Nonlinear regression models [31] are expressed by functions that are not linear in the independent variables. Common models of this type include neural networks, Support Vector Machines (SVM), Multivariate Adaptive Regression Splines (MARS), and tree-based models [32]. In our approach, we use tree-based models.

III. METHOD OVERVIEW

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

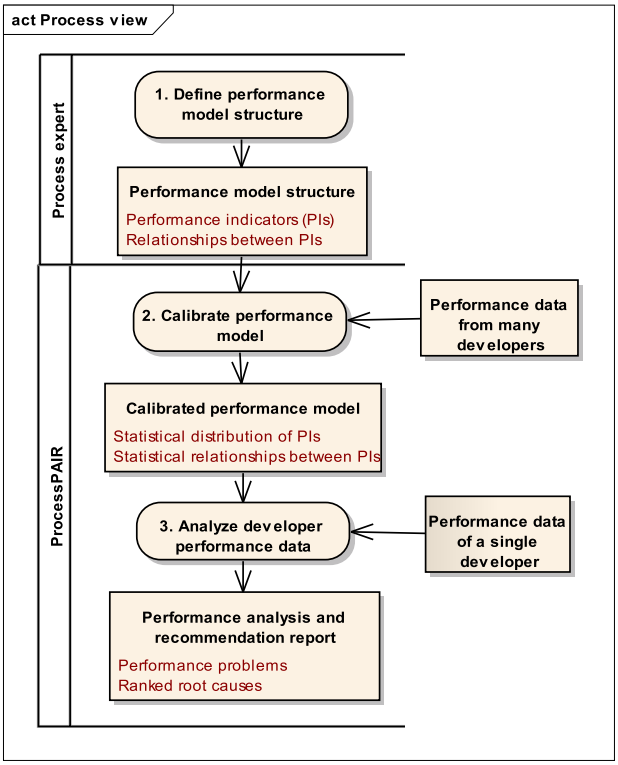

FIGURE 1. UML activity diagram depicting the main activities and artifacts in the ProcessPAIR method.

1) *Define*: Process experts define the structure of a 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.
2) *Calibrate* (or *Learn*): The PM is automatically calibrated by ProcessPAIR based on the performance data of many process users (developers). The statistical distribution of each PI and statistical relationships between PIs are computed from the calibration data set.
3) *Analyze*: Once a PM is defined and calibrated, the performance data of individual developers can be automatically analyzed by ProcessPAIR, to (a) identify performance problems (in top-levels PIs), (b) identify potential root causes (related with child PIs), and (c) rank those potential root causes.

The ProcessPAIR method is supported by the ProcessPAIR tool, with a core framework (process independent) and extensions for the processes of interest. An extension for the PSP, containing the definition of performance models for the PSP and data loaders from project management tools used by PSP developers, was developed for education and training environments, but extensions for other processes can be easily developed.

The three steps of our approach are generically interrelated as described in the next sub-sections.

A. PROBLEM IDENTIFICATION APPROACH

In order to enable the automated identification of performance problems in step 3, one has to first decide on the relevant (top-level) PIs and recommended performance ranges. The relevant top-level PIs and corresponding optimal values are identified by the process expert in step 1. In most cases, the optimal value is implicit in the definition of the PI (e.g., 0 is the optimal value for defect density). The recommended ranges of each PI are determined automatically in step 2, based on its statistical distribution in the calibration data set, according to the following criteria: the $\frac{1}{3}$ values closest to the optimal value (for each side) correspond to ''good'' performance (no performance problem); the $\frac{1}{3}$ values farthest from the optimal value correspond to ''bad'' performance (clear performance problem); the $\frac{1}{3}$ values in between correspond to intermediate performance (potential performance problem).

B. ROOT CAUSE IDENTIFICATION APPROACH

In order to enable the automated identification of root causes of performance problems in step 3, one has to first decide on the relevant cause-effect relationships. In our approach, lower-level PIs that may affect, directly or indirectly, top-level PIs according to a cause-effect relationship are specified by the process expert in step 1. Then, in step 3, it is possible to recursively drill down from problematic top-level PIs (with clear or potential performance problems) to problematic lower-level PIs (with clear or potential performance problems). PIs that can not be further drilled down indicate the potential root causes (as far as the performance data allows us to determine).

C. ROOT CAUSE RANKING APPROACH

When multiple potential root causes (problematic child PIs) are identified for a performance problem in a top-level PI Y, it is important to rank (prioritize) them.

Let $X_1, \ldots, X_n$ be a set of lower-level PIs that affect the value of a higher-level PI Y, according to a function $Y = f(X_1, \ldots, X_n)$, representing an exact formula for deriving Y or a regression formula derived from the calibration data set. We rank the factors X_i according to the values of a *ranking coefficient* ρ_i that represents a cost-benefit estimate of improving each factor X_i whilst keeping the other factors unchanged.

The benefit on Y of a change in the value of a factor X_i can be expressed by the resulting relative variation in the value of Y, i.e., $\Delta Y/Y$.

As for the cost of changing the value of a factor X_i, intuitively, the closest the value is to the optimal value, in terms of percentiles, the more difficult (and less important) it is to improve it. Let us denote by $P_i(X_i) = F_i(X_i) - F_i(o_i)$ the *percentile distance* of X_i to the optimal value, where F_i represents the approximate cumulative distribution function of X_i, and o_i represents the optimal value of X_i. Our base heuristic is that equal relative variations in the $P_i's$ have similar costs. So, we take as cost estimate the relative variation $\Delta P_i/P_i$.

We approximate the cost-benefit ratio using partial derivatives (for small variations) to derive a ranking coefficient (ρ_i):

$$\frac{\frac{\Delta Y}{Y}}{\frac{\Delta P_i}{P_i}} = \frac{\Delta Y}{\Delta X_i}\left(\frac{X_i}{Y}\right) \times \frac{\Delta X_i}{\Delta P_i}\left(\frac{P_i}{X_i}\right) \approx \frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right) \times \frac{\partial X_i}{\partial P_i}\left(\frac{P_i}{X_i}\right).$$

Hence, we can express the ranking coefficient as the product

$$\rho_i = \sigma_{X_i \to Y} \times \pi_{X_i} \tag{1}$$

with

$$\sigma_{X_i \to Y} = \frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right) \tag{2}$$

$$\pi_{X_i} = \frac{\partial X_i}{\partial P_i}\left(\frac{P_i}{X_i}\right) = \frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)} \tag{3}$$

The first factor ($\sigma_{X_i \to Y}$) is a *sensitivity coefficient* [33] that computes the impact of small variations in the value of a factor X_i on the value of Y, whilst keeping all the other factors unchanged.

The second factor (π_{X_i}), which we call a *percentile coefficient*, computes the impact of small variations in the current percentile distance (P_i) of X_i to the optimal value (o_i) on the value of X_i. We denote by $F_i'(X_i)$ the first derivative of $F_i(X_i)$, representing the probability density function.

The optimal value of each PI is provided by the process expert in step 1. The approximate statistical distribution of each PI is automatically computed from the calibration data set in step 2. Regarding the sensitivity coefficients, in case parent and child PIs have an exact relationship, the sensitivity coefficient is provided by the process expert in step 1; in case parent and child PIs are statistically related, a regression model and corresponding sensitivity coefficients are computed automatically in step 2 from the calibration data set.

The concepts and procedures involved in each step of the ProcessPAIR method are detailed in the next sections with the help of meta-models and examples.

IV. PERFORMANCE MODEL DEFINITION

A. PERFORMANCE MODEL CONTENTS

A performance model for a development process under consideration is defined by means of the following data (see also Figure 2):

- set of relevant base measures generated by the development process under consideration, at the project level, with the attributes listed in class *Measure* of Figure 2: short name, long name, description, measurement units, minimum value of the scale, maximum value of the scale, and number of decimal digits (precision digits);
- set of relevant top-level PIs, described by the same attributes as the base measures, plus the optimal value (usually implied by the definition of each PI) and formula for computation from base measures;

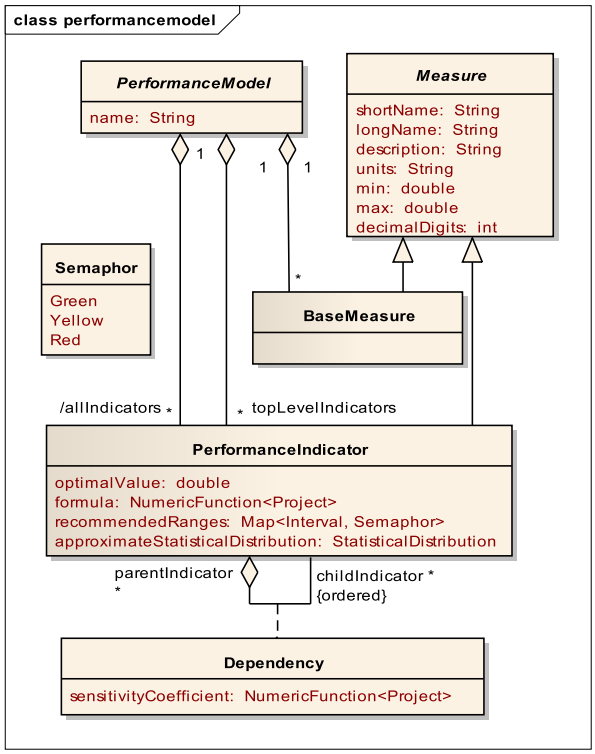

FIGURE 2. UML class diagram depicting the main concepts involved in model definition and calibration.

- child PIs that affect directly or indirectly (by a cause-effect relationship) the top-level PIs according to a formula or statistical evidence;
- sensitivity coefficients $\sigma_{X_i \to Y}$, $i = 1, \ldots, n$, for each PI Y that is affected by child PIs $X_1, \ldots, X_n$ according to an exact formula $Y = f(X_1, \ldots, X_n)$.

B. EXAMPLE OF PERFORMANCE MODEL DEFINITION

In this section, we exemplify the derivation of a PM for the PSP, partially depicted in 3 (more details in [34]).

Beforehand, it is worth noting that the following base measures are collected in the PSP:

- the estimated ($\hat{T}_k$) and actual time (T_k) (effort) spent per process phase k (Plan, Design, Design Review, Code, Code Review, Compile, Unit Test, and Postmortem), and corresponding totals over all phases ($\hat{T}$ and T);
- the number of defects injected and removed per process phase;
- the estimated ($\hat{S}$) and actual size (S) of the deliverable, measured in an appropriate size unit, such as lines of code (LOC) or function points (FP).

Since the scope of the PSP is the development of small programs or components of larger programs, the Requirements, High-Level Design, and System Testing phases are absent (they can be found in the more complete TSP [12]). In some programming environments, the Compile phase may be absent. The phases may be performed iteratively.

For top-level PIs, we choose three PIs related to predictability, quality and productivity.

The major **predictability** PI in the PSP is the *Time Estimation Accuracy*, which we measure as the ratio between actual and estimated values. Because in the PSP's PROBE method [11] a time estimate is obtained based on a size estimate of the deliverable (in added or modified size units) and a productivity estimate (in size per time units), we indicate in Figure 3 that the *Time Estimation Accuracy* (*TimeEA*) is affected by the *Size Estimation Accuracy* (*SizeEA*) and the *Productivity Estimation Accuracy* (*ProdEA*).

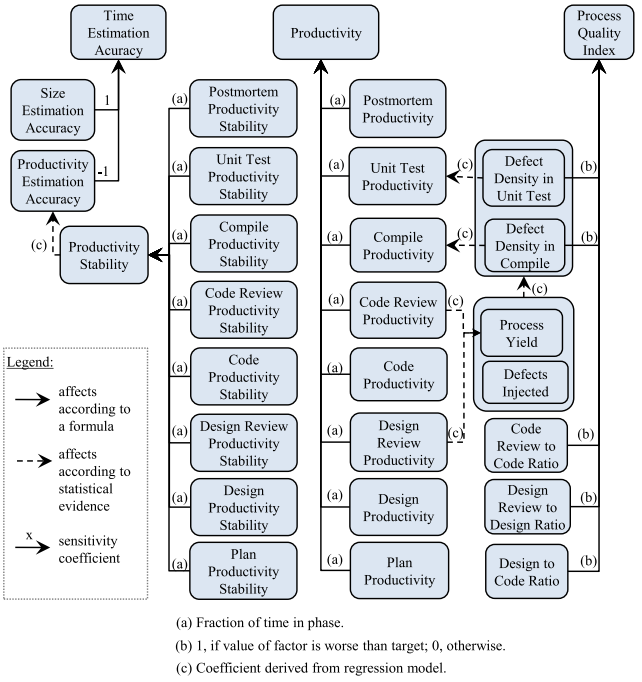

FIGURE 3. Outline of an example performance model, with performance indicators, dependencies and sensitivity coefficients.

These PIs are related by the formula $TimeEA = SizeEA/ProdEA$. Because they are multiplicative factors with powers of 1 and -1, the corresponding sensitivity coefficients are also 1 and -1, as displayed in Figure 3. This means that the *TimeEA* is equally sensitive to *SizeEA* and *ProdEA* (although in opposite directions).

Because in the PROBE method productivity estimates are based on historical productivity [11], we indicate in Figure 3 that the *Productivity Estimation Accuracy* is affected by the *Productivity Stability*.

Since in the PSP time is recorded per process phase, when a productivity stability problem is encountered, one can analyze the productivity stability per phase, in order to determine the problematic phase(s). Hence, we indicate in Figure 3 a set of PIs for the productivity stability per phase, which together affect the overall productivity stability. Similarly to the productivity (to be discussed next), the overall productivity stability is more sensitive to the productivity stability of the more time-consuming phases.

As top-level quality indicator, we choose an aggregated quality measure—the *Process Quality Index* (PQI)—that constitutes an effective predictor of post-delivery defect density [11], [35] (a common product quality measure [36]).

The PQI is computed from five components, depicted in Figure 3 as factors that affect the PQI according to a formula: the ratio of design time to coding time (an indicator of design quality), the ratio of design review time to design time (an indicator of design review quality), the ratio of code review time to coding time (an indicator of code review quality), the ratio of compile defects to a size measure (an indicator of code quality), and the ratio of unit test defects to a size measure (an indicator of program quality). The components are normalized to the [0, 1] interval such that 0 represents poor practice, and 1 represents desired practice. The sensitivity of the PQI to its components is equal to 1 for components with value worse than the desired value, and 0 for components with a value equal to or better than the desired value (calculation details can be consulted in [34]).

In turn, both the *Defect Density in Compile* and *Unit Test* are affected by the total density of *Defects Injected* (and found) and the percentage of defects removed before compile and test (called *Process Yield* in the PSP). In fact, high defect densities in compile and test may be caused by a large number of defects injected (due to poor defect prevention) or a large number of defects escaped from previous defect filters (due to poor design and code reviews).

According to several studies [11], [28], [37], the time spent in reviewing a work product in relation to its size is a leading indicator of the review yield (percentage of defects found) and consequently of the process yield. Hence, we indicate in Figure 3 that the *Process Yield* is affected by the *Design Review* and *Code Review Productivity* (measured as size to time ratios).

In the PSP, **productivity** is measured in size units delivered per time units consumed. Any size measure can be used (function points, LOC, etc.) as long as it correlates with effort and is objectively measurable [11]. Because in the PSP time is recorded per process phase, when a productivity problem is encountered, one can analyze the productivity per phase, in order to determine the problematic phase(s). Hence, we indicate in Figure 3 a set of PIs for the productivity per phase, which together affect the overall productivity.

From the definitions of the overall productivity (P) and productivity per phase (P_k) as size to time ratios ($P = S/T$ and $P_k = S/T_k$, respectively), and the relation $P^{-1} = \sum_k {P_k}^{-1}$, we derive the sensitivity coefficients $\sigma_{P_k \to P} = T_k/T$ (fraction of project time spent in phase k). This means that the overall productivity is more sensitive to the productivity of the more time consuming phases. E.g., if a developer spends 40% of the project time in Test, and reduces the time in Test by 20% (whilst keeping the time spent in the other phases unchanged), the overall productivity will improve by $20\% \times 40\% = 8\%$.

In turn, the time spent in the compile and test phases (including defect fixing) may be affected by the number of defects to fix, so we indicate in Figure 3 that the *Compile* and the *Unit Test Productivity* may be affected by the *Defect Density in Compile* and *Unit Test*, respectively.

V. PERFORMANCE MODEL CALIBRATION

The PM is automatically calibrated by ProcessPAIR from training data sets, containing values of base measures for many developers and projects. The user has just to select the performance model to be calibrated, the data set to be used for calibration, the filtering criteria (if desired), and the XML file for saving the calibration results.

The following data is generated by the calibration process (also visible in Figure 2): noitemsep

- approximate statistical distribution of each PI, represented by a cumulative distribution function;
- recommended performance ranges for each PI, represented by a map from performance ranges (intervals) to colors (semaphores);
- regression models and sensitivity coefficients between related PIs, but not by an exact formula.

A. APPROXIMATE CUMULATIVE DISTRIBUTION FUNCTIONS

The approximate cumulative distribution function of each PI could be obtained by computing a theoretical distribution that best fits the training data, or by linear interpolation between a few percentiles computed from the training data. Since different PIs may follow different types of continuous distributions or may even follow a hybrid continuous-discrete distribution, with non-zero probability at the ends of the scale, we opted for the second method.

To balance fit, smoothing and storage space, we sample the cumulative frequency distribution of the training data for the following relative frequencies: 0%, 1%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 95%, 99%, 100%, maximum relative frequency for which the observed value equals the minimum of the scale (if any observed), and minimum relative frequency for which the observed value equals the maximum of the scale (if any observed). The approximate cumulative distribution function of the PI under consideration is a piecewise linear function that connects the sampling points, as illustrated in Figure 4. The only calibration information that needs to be stored are the sampling points (illustrated by red circles in Figure 4).

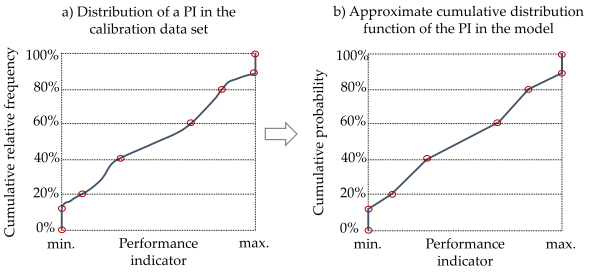

FIGURE 4. Illustration of the procedure for obtaining an approximate cumulative distribution function of a PI (example with hybrid continuous-discrete distribution, and sampling at 20% intervals).

B. PERFORMANCE RANGES

Performance ranges are needed for classifying values of each PI of a subject under analysis into three colors: noitemsep

- green - no performance problem;
- yellow - a possible performance problem;
- red - a clear performance problem.

Such ranges are determined automatically from the approximate cumulative distribution function computed in the previous step, so that there is an even distribution of training data points by the colors (1/3 of data points per color). In case the optimal value is located in one of the extremes of the scale, the 'green' range is also located in the same extreme of the scale, the 'red' range in the other extreme, and the 'yellow' range in the middle. In that case, thresholds are computed from the cumulative distribution function based on terciles and subsequently rounded to the number of precision digits specified for the PI under consideration. In case the optimal value is located in the middle of the scale, we split the intervals to the left and the right of the optimal value based on terciles, in order to derive the performance ranges, as illustrated in Figure 5. Denoting by p the cumulative probability corresponding to the optimal value, the terciles to the left of the optimal value are spaced at intervals of *1/3p*, whilst the terciles to the right of the optimal value are spaced at intervals of *1/3(100%-p)*. Although in our work we considered an even distribution of training data points by the 3 colors, other distributions could also be easily supported, depending on the performance goals.

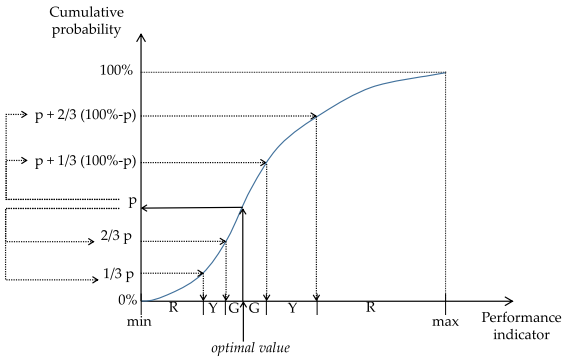

FIGURE 5. Illustration of the procedure for determining the green (G), yellow (Y) and red (R) ranges from the cumulative distribution function of a PI, in case the optimal value lays in the middle of the scale.

C. REGRESSION MODELS AND SENSITIVITY COEFFICIENTS

Sensitivity coefficients between PIs not related by an exact formula are computed by first determining a regression model from the calibration data set and subsequently computing the corresponding sensitivity coefficients.

Assume that in step 1 the process expert indicated that a performance indicator Y is affected by $X_1, \ldots, X_n (n \geq 1)$, but Y is not determined by those factors according to an exact formula. E.g., in Figure 3, we indicate that *Process Yield* is affected by *Code Review Productivity* and *Design Review Productivity* based on statistical evidence from literature [37].

In such case, we first compute a regression model $\hat{Y} = f(X_1, \ldots, X_n)$ from the calibration data set for predicting the value of Y from the values of $(X_1, \ldots, X_n)$.

Because the relationship between the PIs involved may be nonlinear, instead of computing a global linear model, we have the option to compute a piecewise linear model organized as a regression tree [38]. The type of model to use (global or piecewise linear model) is selected by the process expert in the first step of our method. A regression tree [38] is a binary decision tree; at each non-leaf node, the value of one of the factors (X_i) is compared to a so-called split value in order to decide the sub-tree to follow. Hence, each leaf will correspond to a subset (n-dimensional cube or cell) of the space of possible values of $X_1, \ldots, X_n$. Since we are interested in computing a piecewise linear model, each leaf node k has an associated linear model $\hat{Y} = \beta_{k,0} + \beta_{k,1}X_1 + \ldots + \beta_{k,n}X_n$, computed from the points in the calibration data set that fall in that cube by linear regression.

In the recursive tree construction process, we use the following split criterion: as suggested in [39], we select the split variable (X_i) that provides the highest decrease of the tree SSE (sum of squared errors); for each tentative split variable X_i, we do a split as closest as possible to a binary split that guarantees disjoint X_i values on both segments (this is important to handle repeated values). We use the following stop criterion: a cell size cannot contain less than a specified minimum number of training data points (100 for inner cells, and a number between 100 and 25 for border cells, depending on the number of borders).

An alternative to regression trees are multivariate adaptive regression splines (MARS) [40], which also handle non-linearities and interactions between factors, and generate more compact models than regression trees. In the experiments conducted, we obtained lower mean absolute errors (MAE) and root mean squared errors (RMSE) with regression trees as compared to MARS, so we use the described regression trees, in spite of the higher storage space required.

In each leaf node k we only have to store the coefficients $\beta_{k,0}, \beta_{k,1}, \ldots, \beta_{k,n}$ of the model. In each non-leaf node, we have to store the index and value of the split variable. During performance analysis (step 3), once determined the cell k corresponding to a data point $(X_1, \ldots, X_n, Y)$ under consideration, the sensitivity coefficients are simply computed as:

$$\sigma_{X_i \to Y} = \frac{\partial \hat{Y}}{X_i}\left(\frac{X_i}{Y}\right) = \beta_{k,i}\frac{X_i}{Y} (i = 1, \ldots, n). \quad (4)$$

D. DATASET FILTERING

Instead of using the full data set for calibration, we have the option to use only the data points most similar to a given user profile. This requires that each data point in the calibration data set, besides values for the base measures, also contains values for the variables that can be used for filtering. Those

variables need to be specified by the process expert together with the PM.

We compute similarity with the Gower similarity coefficient [41], because it provides a measure of proximity adequate for mixed data types, combining numeric and categorical data.

Let n be the number of variables under consideration (the ones constrained in the supplied profile), and let p and t be vectors that contain the values of those variables in the supplied profile and in a data point in the training data set, respectively.

In the case of a categorical variable k (e.g., programming language), the similarity s_k between t and p regarding $k(1 \leq k \leq n)$, is a number between 0 (least similar) and 1 (most similar) computed as [41]:

$$s_k = \begin{cases} 1 & \text{if } t_k = p_k, \\ 0 & \text{otherwise .} \end{cases} \tag{5}$$

In the case of a numerical variable with a finite range (e.g., programming experience in years), the similarity is computed as [41]:

$$s_k = 1 - \frac{|t_k - p_k|}{r_k} \tag{6}$$

where r_k is the range of values for the k-th variable (distance between minimum and maximum values).

However, many numerical variables of interest have positive values spread along a wide range, with many small values and a few large values. Using the above formula, the similarity would be close to 0 for most of the pairs of data points. In such cases, the process expert may specify a monotonic scale transformation function to be applied prior to using the similarity formula. E.g., the function $1 - e^{\frac{-x}{\tau}}$, where x is the variable of interest and τ is a constant, transforms the $[0, +\infty[$ interval to the [0,1[interval.

Another possibility (that can be selected by the process expert) is to compute the similarity based on the ranking of x in the training data set (i.e., the relative position of x in the sorted multiset of values in the training dataset).

By default, all the variables are given equal weights, so the similarity s between t and p (considering all the variables) becomes a simple average:

$$s = \frac{1}{n} \sum_{k=1}^{n} s_k \tag{7}$$

Regarding the number of most similar data points to select, we use the following criteria: for statistical significance, at least 50 data points are selected; additionally, all data points with a similarity to the given profile greater or equal than 0.9 are selected. These numbers can be configured. Data points with undefined values for any of the variables under consideration are discarded.

E. EXAMPLE OF MODEL CALIBRATION

To calibrate the previously described performance model for the PSP, we used a curated PSP data set from the Software Engineering Institute (SEI), referring to 31,140 projects concluded by 3,114 engineers during 295 classes of the PSP for Engineers I/II training courses running between 1994 and 2005. In this training course, targeting professional developers, each participant develops ten small projects. In this data set, size is measured in source lines of code, excluding comments and blank lines.

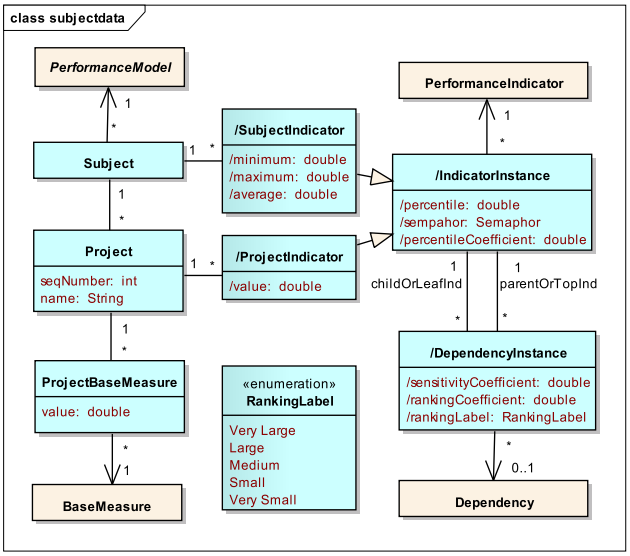

FIGURE 6. UML class diagram depicting the main concepts involved in the analysis of subject data.

Calibration results for the *Time Estimation Accuracy* can be viewed in Figure 8, namely the cumulative distribution function (bottom left) and performance ranges (middle right). In this case, the green range corresponds to the [0.87, 1.20] interval, that is, an estimation error between -13% (overestimation) and 20% (underestimation), which matches what is usually considered an acceptable error [42], [43].

Regarding productivity, the green range obtained corresponds to values above 35 LOC/hour, which also matches recommendations from literature [44].

Regarding defect density in Unit Test and Compile, the green ranges correspond to <= 11 and <= 15 defects/KLOC, respectively, which are less tight than the PSP recommendations (<=5 defects/KLOC for Unit Test and <=10 defects/KLOC for Compile) [11].

VI. PERFORMANCE ANALYSIS

The base performance data of a subject under analysis (developer, team, etc.) that needs to be uploaded by ProcessPAIR, consists of the values of the base measures defined in the selected performance model for a series of projects (see *Subject*, *Project*, and *ProjectBaseMeasure* in Figure 6). Each project is characterized by a sequence number (for ordering purposes) and a name (for identification purposes).

In the sequel, concrete examples of the parameters labeled with encircled numbers can be consulted in Figures 7 and 8.

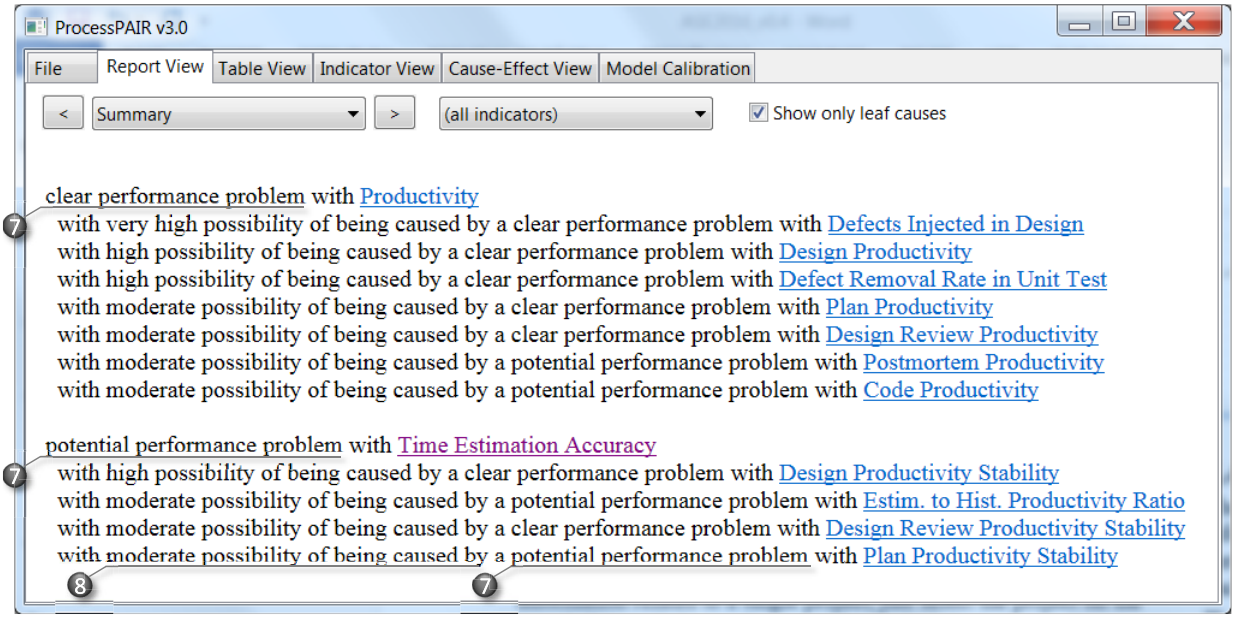

FIGURE 7. Report View example (annotated).

A. PROJECT LEVEL CALCULATIONS

Based on the values of the base measures, ProcessPAIR first computes the following parameters for each PI i and project k (see *ProjectIndicator* and *IndicatorInstance* in Figure 6):

- *value* ($x_i^{(k)}$) ① - computed from the base measures and PI's formula; if this value is undefined, the following parameters are also undefined;
- *normalized percentile* ($N_i^{(k)}$) - computed from the previous value and the statistical distribution of the PI in the PM, normalized so that 100% corresponds to the optimal value and 0% corresponds to extreme values to the left or to the right of the optimal value. Formally, denoting by F_i and o_i the approximate cumulative distribution function and optimal value of the PI under consideration, respectively, we have:

$$N_i^{(k)} = \begin{cases} 1 & \text{if } x_i^{(k)} = o_i, \\ \dfrac{F_i(x_i^{(k)})}{F_i(o_i)} & \text{if } x_i^{(k)} < o_i, \\ \dfrac{1 - F_i(x_i^{(k)})}{1 - F_i(o_i)} & \text{if } x_i^{(k)} > o_i. \end{cases} \quad (8)$$

- *semaphore (color)* ($s_i^{(k)}$) ② - a discretization of the normalized percentile, for user presentation purposes, according to the mapping (m_1):

  $[0, \frac{1}{3}[\rightarrow$ *Red* (clear performance problem)

  $[\frac{1}{3}, \frac{2}{3}[\rightarrow$ *Yellow* (potential p. problem)

  $[\frac{2}{3}, 1] \rightarrow$ *Green* (no performance problem)
- *percentile coefficient* ($\pi_i^{(k)}$) - computed according to equation 3. The percentile coefficient is needed for computing the ranking coefficient but is hidden from the normal user.

For each dependency defined in the PM between PIs i (child) and j (parent) with defined values in project k, the following information is computed (see *DependencyInstance* in Figure 6):

- *sensitivity coefficient* ($\sigma_{i\rightarrow j}^{(k)}$) - computed from the project data and the sensitivity formula defined in the PM. The sensitivity coefficient is needed for computing the ranking coefficient, but is hidden from the normal user;
- *ranking coefficient* ($\rho_{i\rightarrow j}^{(k)}$) - computed as the product $\sigma_{i\rightarrow j}^{(k)} \times \pi_i^{(k)}$, according to equation 1;
- *ranking label* ($r_{i\rightarrow j}^{(k)}$) - a discretization of the ranking coefficient in terms of T-shirt sizes, for user presentation purposes, according to the mapping (m_2):

  $]-\infty, 0.01[\rightarrow$ *VerySmall*
  $[0.01, 0.1[\rightarrow$ *Small*
  $[0.1, 1[\rightarrow$ *Medium* (or *Moderate*)
  $[1, 10[\rightarrow$ *Large* (or *High*)
  $[10, +\infty[\rightarrow$ *VeryLarge* (or *VeryHigh*)

  Such thresholds may be interpreted as follows: considering an improvement by 10% in the child PI (reduction of the percentile distance to the optimal value), the estimated improvement in the parent PI will be $\leq$ 0.1% (very small), 0.1%-1% (small), 1% - 10% (medium), 10%-100% (large), $\geq$ 100% (very large).

The following information is also computed between child indicators i (at any level) and top-level indicators j with defined values in project k:

- *composite sensitivity coefficient* ($\sigma'^{(k)}_{i\rightarrow j}$) - recursively computed from the (elementary) sensitivity coefficients, as explained in section VI-C;

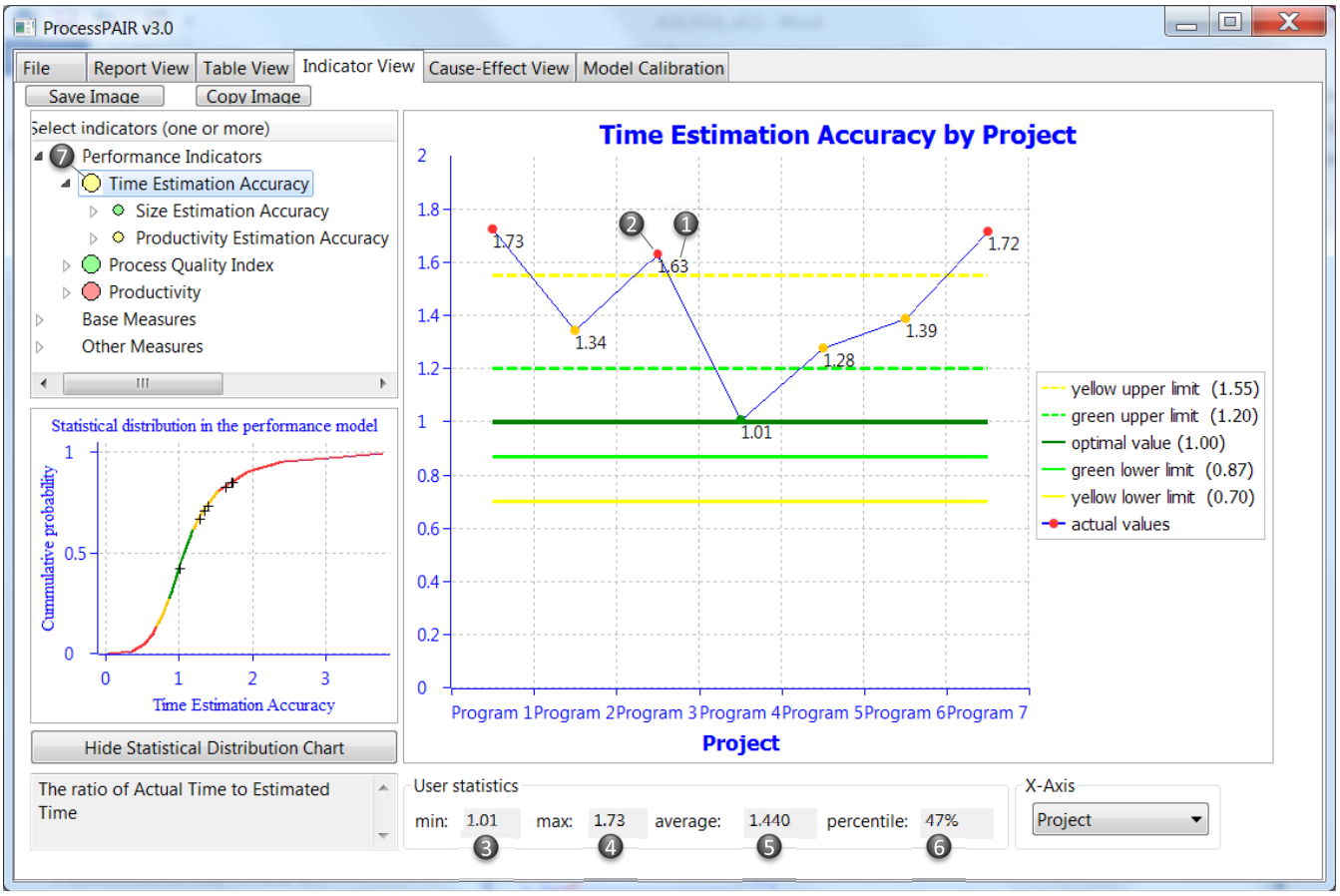

FIGURE 8. **Indicator View example (annotated).**

- *composite ranking coefficient* ($\rho'^{(k)}_{i\to j}$) - computed as the product $\sigma'^{(k)}_{i\to j} \times \pi_i^{(k)}$;
- *composite ranking label* ($r'^{(k)}_{i\to j}$) - computed as the mapping $m_2(\rho'^{(k)}_{i\to j})$.

B. SUBJECT LEVEL CALCULATIONS

For aggregating data, we compute a weighted average of the values calculated at the project level, using an exponentially decaying weight for older projects, based on a configurable memory retention factor f ($0 < f < 1$, *e.g.*, 0.85). Formally, denoting by $def(a)$ a predicate that indicates if a is defined, and by $b^{(1)}, \ldots, b^{(m)}$ the values to be aggregated, the weighted average is computed as:

$$wavg(b^{(1)}, \ldots, b^{(m)}) = \frac{\sum_{k\in\{1,\ldots,m\}\wedge def(b^{(k)})} f^{\text{m-k}} b^{(k)}}{\sum_{k\in\{1,\ldots,m\}\wedge def(b^{(k)})} f^{\text{m-k}}} \quad (9)$$

The following summary (aggregate) information is computed for each PI i at the subject level (see *SubjectIndicator* and *IndicatorInstance* in Figure 6):

- *minimum* ③, *maximum* ④, *average* ⑤ - simple statistics calculated from the values computed at the project level;
- *aggregated normalized percentile* (N_i) ⑥ - computed as $wavg(N_i^{(1)}, \ldots, N_i^{(k)})$.
- *aggregated semaphore* (s_i) ⑦ - computed by the same mapping used at the project level, as $m_1(N_i)$.

The following aggregated information is computed at the subject level, for direct dependencies between PIs i (child) and j (parent):

- *aggregated ranking coefficient* ($\rho_{i\to j}$) - computed as $wavg(\rho^{(1)}_{i\to j}, \ldots, \rho^{(m)}_{i\to j})$.
- *aggregated ranking label* ($r_{i\to j}$) - computed by the same mapping used at the project level, as $m_2(\rho_{i\to j})$.

The following aggregated information is also computed at the subject level, for indirect dependencies between a child PI i (at any level) and a top-level PI j:

- *aggregated composite ranking coefficient* ($\rho'_{i\to j}$) - computed as $wavg(\rho'^{(1)}_{i\to j}, \ldots, \rho'^{(m)}_{i\to j})$.
- *aggregated composite ranking label* ($r'_{i\to j}$) ⑧ - computed as $m_2(\rho'_{i\to j})$.

C. COMPOSITE SENSITIVITY COEFFICIENTS

In the general case, the root causes to be ranked $X_1, \ldots, X_n$ may be indirect children of the problematic top-level PI under consideration (Y). In this case, the function f that relates the involved PIs may be expressed as a composite function, based on the elementary functions that relate each PI with its

direct children. As a consequence, the sensitivity coefficients may be computed using the laws of partial derivatives of composite and multivariate functions.

In general, denoting by $C(X_j)$ the direct child of a PI X_j, the composite sensitivity coefficient σ' between any two PIs can be computed recursively from the elementary sensitivity coefficients (σ) as follows:

$$\sigma'_{X_i \to X_j} = \begin{cases} \sigma_{X_i \to X_j} & \text{if } X_i \in C(X_j), \\ 1 & \text{if } i = j, \\ \sum_{k \in C(X_j)} \sigma'_{X_i \to X_k} \sigma'_{X_k \to X_j} & \text{otherwise.} \end{cases} \quad (10)$$

D. ROOT CAUSE SELECTION AND SORTING

In this section, we explain how the relevant root causes of an identified performance problem are selected and sorted based on the ranking coefficients and colors, for presentation to the user (as in Figure 7).

Assume that a problematic top-level PI Y was identified (with red or yellow color) for a specific project or overall for a subject under analysis, with a set of direct and indirect child PIs. The selection of the relevant root causes is performed in three steps:

1) Cut the child PIs (and their descendants) that have a green semaphore or a ranking coefficient to parent below a specified threshold (0.1 by default, to eliminate nodes with ranking labels 'very small' or 'small');
2) Repeatedly cut the leaves that have a composite ranking coefficient to root (Y) below the specified threshold, until there are no leaves with that condition;
3) Select the remaining leaves as the relevant root causes to be presented to the user, sorted according to the ranking coefficient to root.

E. ANALYSIS VIEWS

The results of performance analysis are presented to the user through multiple views, some of which are illustrated in Figures 7 and 8 for a real (anonymized) user. The user has just to select a previously calibrated performance model for a process under consideration and provide a file with his/her base performance measures for a set of projects.

The **Report View** (Figure 7) shows in a simple way, overall (''Summary'') or project by project, the most relevant top-level performance problems (colored red or yellow in other views) and potential root causes properly prioritized (according to the ranking coefficients, previously explained).

Intermediate causes can be consulted by unchecking the ''Show only leaf causes'' checkbox. Comboboxes allow selecting information for specific projects and/or PIs. The links skip to the Indicator View, for detailed information about each PI.

In the example of Figure 7, there is a clear performance problem (signaled with the red color in other views) with the ''Productivity'' PI, and a potential performance problem (signaled with the yellow color in other views) with the ''Time Estimation Accuracy'' PI. The top 3 potential root causes for the first problem are related with problems in the design phase (''Defects Injected in Design'' and ''Design Productivity'') and the unit test phase (''Defect Removal Rate in Unit Test'').

The **Indicator View** (Figure 8) shows the behavior of each PI throughout the series of projects under analysis, together with model calibration information (description, optimal value, recommended performance ranges and statistical distribution). It also supports interactive analysis.

In the bottom left, the statistical distribution of the PI in the data set used for calibrating the model is presented. The colors correspond to the performance ranges. The actual values in the file under analysis are also shown, marked with the ''+'' symbol, for benchmarking purposes.

In the tree-view on the top-left, the user can easily drill down from problematic top-level PIs (with clear or potential performance problems, colored read and yellow, respectively) to problematic lower-level PIs (with clear or potential performance problems).

At the bottom, it are presented several statistics, computed as explained in previous sections.

In the example shown in Figure 8, the yellow circle in the first PI signals a potential performance problem with the ''Time Estimation Accuracy''. The run chart on the right depicts the values of this PI for a sequence of 7 projects (program 1 to 7), showing a tendency for underestimation, with an initial trend for improvement followed by a trend in the opposite direction.

VII. EXPERIMENTAL RESULTS

In this section, we summarize the results of two experiments conducted in collaboration with Instituto Tecnológico de Monterrey (ITM) in Mexico to assess the effectiveness of the ProcessPAIR method and tool. In both experiments, we used the already described PSP model (with minor modifications) and PSP data set (for model calibration).

In the ''Software Quality and Testing'' course at ITM, master students develop a sequence of 6 or 7 small projects, collecting several performance measures and following increasingly elaborated processes (PSP0 to PSP2.1). By the end of the course, students are requested to analyze their personal performance data collected throughout those projects and document their findings and improvement proposals in a final report (Performance Analysis Report). To guide their analysis, students are asked to address 30 questions organized in four categories: size estimating accuracy, time estimating accuracy, defect and yield analysis, and quality analysis. Students use Process Dashboard [7] for data collection and visualization.

A. POSTMORTEM STUDY

In the first experiment [34], we conducted a postmortem study based on the performance data and reports produced by 27 students that attended the 2015 edition of the course.

We fed their performance data to ProcessPAIR and compared the results of automated analysis (with ProcessPAIR) and manual analysis (documented by the students in their final reports), to assess whether ProcessPAIR is able to accurately identify performance problems (RQ1.1) of individual developers and their root causes (RQ1.2).

Regarding problem identification (RQ1.1), for the cases (302) in which students explicitly characterized their performance (regarding a specific PI and a specific project or all projects), we compared the student assessment with the tool-based assessment, and got the following overall results:

- in 96% of the cases, the results of manual and automatic analysis matched, i.e., both the student and the tool indicated good performance or bad performance;
- in 3% of the cases, the tool indicated a potential problem, but the developer indicated no performance problem (*false positives*);
- in 1% of the cases, the tool indicated no performance problem, but the developer explicitly indicated a performance problem, based on abnormally tight thresholds (*false negatives*).

Hence, we conclude that the automatic analysis produced similar results as the manual analysis, with 96% accuracy, and a very small number of false positives (3%).

Regarding root cause identification (RQ1.2), for the cases (53) of performance problems in top-level PIs identified both in manual and automated analysis, we compared the respective root causes and obtained the following overall results:

- in 21% of the cases, the tool pointed out the same causes as the developer;
- in 53% of the cases, the tool pointed out intermediate causes, and the developer pointed out deeper causes;
- in 26% of the cases, the results are inconsistent, because of faults in the manual analysis.

Hence, we conclude that, in the cases in which the manual analysis was not erroneous, the tool-based analysis was able to accurately identify either the same causes or causes in the same direction as in the manual analysis.

These results suggest that ProcessPAIR may help reducing errors and effort in manual analysis, by providing a good starting point for subsequent manual analysis, focused on the identification of deeper causes and improvement actions.

B. CONTROLLED EXPERIMENT

In the second experiment [10], we conducted a controlled experiment with 61 students that attended the 2016 edition of the course, to assess if students that use ProcessPAIR for performing their final report assignment are more satisfied with the tool support (RQ2.1), produce higher quality reports (RQ2.2), and spend less time (RQ2.3) than students that perform that assignment in a traditional way (using Process Dashboard only).

To that end, half of the students (selected randomly) used ProcessPAIR in their final report assignment, and the other half performed it in a traditional way (using Process Dashboard only). To measure user satisfaction, we prepared a web-based survey with 14 questions that was answered by the students of both groups after concluding the assignment.

We obtained the following overall results:

- User satisfaction (RQ2.1): average score of 4.78 in a 1-5 scale for ProcessPAIR users, against 3.81 for non-ProcessPAIR users, representing a 25% improvement (difference statistically significant);
- Quality of analysis reports (RQ2.2): average grade of 88.1 in a 0-100 scale for ProcessPAIR users, against 82.5 for non-ProcessPAIR users, representing a 7% improvement (difference statistically significant);
- Time spent (RQ2.3): average of 252 minutes for ProcessPAIR users, against 262 minutes for non-ProcessPAIR users, representing a 4% reduction (difference not statistically significant).

These results show that ProcessPAIR provided significant benefits in terms of user satisfaction and quality of analysis outcomes, and (to some surprise) marginal benefits only in terms of the time required to do the analysis. However, the students that used ProcessPAIR spent a significant part of their time performing repetitive tasks (copying charts and data from ProcessPAIR to the analysis report) that we automated in a subsequent version of the tool.

VIII. CONCLUSION

The results achieved show that ProcessPAIR is able to automatically analyze the performance data of individual developers in order to identify and rank performance problems and potential causes, and, consequently, reduce errors and effort and improve user satisfaction in performance analysis, by taking advantage of performance models derived from the performance data of many process users.

The only manual work needed is the definition, by a process expert, of the relevant PIs and dependencies for the process under consideration (only once per process).

Currently, ProcessPAIR is available as a standalone Java application. As future work, we intend to deploy ProcessPAIR as a service available on the cloud (SaaS) and integrate it with a cloud-based application lifecycle management tool. This will increase tool accessibility, facilitate metrics collection, and enable the automatic recalibration of the performance models based on the performance data of the users.

Currently, ProcessPAIR is successfully used by PSP users. In the future, we intend to expand the user base by tailoring ProcessPAIR for other processes, namely, agile and hybrid methods [13], [14], [45].

Machine learning techniques are increasingly used in software engineering for performance analysis [46] and process improvement [47], [48]. Hence, we intend to explore machine learning and process mining techniques to further automate the ProcessPAIR method, e.g., to automatically calibrate optimal values of PIs, identify relationships between PIs, and, in general, automate the generation of performance models. By combining artificial intelligence techniques with

human expertise (crowdsourcing), we will extend ProcessPAIR with the capability of recommending detailed improvement actions for the identified causes of performance problems.

ACKNOWLEDGMENT

The authors would like to acknowledge the SEI and Tec de Monterrey for facilitating the access to the PSP data for performing this research and AWKUM for their partial initial grant.

REFERENCES

[1] B. Boehm, "Some future software engineering opportunities and challenges," in *The Future of Software Engineering*. Springer, 2011, pp. 1–32.
[2] Ron S Kenett and Emanuel Baker, *Software Process Quality: Management and Control*. Boca Raton, FL, USA: CRC Press, 1999.
[3] *CMMI Model V2.0*, CMMI Isntitute, Pittsburgh, PA, USA, 2018.
[4] M. B. Chrissis, M. Konrad, and S. Shrum, *CMMI for Development: Guidelines for Process Integration and Product Improvement*, London, U.K.: Pearson, 2011.
[5] *Hackystat*. Accessed: Aug. 20, 2019. [Online]. Available: https://hackystat.github.io/
[6] D. Burton and W. Humphrey, "Mining PSP data," in *Proc. TSP Symp.* San Diego, CA, USA: Carnegie Mellon Univ., Software Engineering Institute, 2006, pp. 1–28.
[7] Tuma Solutions. *The Software Process Dashboard Initiative Home Page*. Accessed: Aug. 20, 2019. [Online]. Available: http://www.processdash.com/
[8] M. Raza and J. P. Faria, "A model for analyzing performance problems and root causes in the personal software process," *J. Software: Evol. Process*, vol. 28, no. 4, pp. 254–271, Apr. 2016.
[9] M. Raza and J. P. Faria, "ProcessPAIR: A tool for automated performance analysis and improvement recommendation in software development," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. ASE*, 2016, pp. 798–803.
[10] M. Raza, J. A. P. Faria, and R. Salazar, "Assisting software engineering students in analyzing their performance in software development," *Softw. Qual. J.*, vol. 27, no. 3, pp. 1209–1237, 2019.
[11] W. S. Humphrey, *PSP (SM): A Self-Improvement Process for Software Engineers*. Reading, MA, USA: Addison-Wesley, 2005.
[12] W. Humphrey and J. Over, *Leadership, Teamwork, and Trust: Building a Competitive Software Capability*. Reading, MA, USA: Addison-Wesley, 2010.
[13] M. Kuhrmann, E. Hanser, C. R. Prause, P. Diebold, J. Münch, P. Tell, V. Garousi, M. Felderer, K. Trektere, F. McCaffery, and O. Linssen, "Hybrid software and system development in practice: Waterfall, scrum, and beyond," in *Proc. Int. Conf. Softw. Syst. Process ICSSP*, 2017, pp. 30–39.
[14] L. Alperowitz, D. Dzvonyar, and B. Bruegge, "Metrics in agile project courses," in *Proc. 38th Int. Conf. Softw. Eng. Companion - ICSE*, 2016, pp. 323–326.
[15] A. del-Río-Ortega, M. Resinas, C. Cabanillas, and A. Ruiz-Cortés, "On the definition and design-time analysis of process performance indicators," *Inf. Syst.*, vol. 38, no. 4, pp. 470–490, Jun. 2013.
[16] A. del-Río-Ortega, M. Resinas, A. Durán, B. Bernárdez, A. Ruiz-Cortés, and M. Toro, "Visual ppinot: A graphical notation for process performance indicators," *Bus. Inf. Syst. Eng.*, vol. 61, no. 2, pp. 137–161, Apr. 2019.
[17] W. Navidi, *Statistics for Engineers and Scientists*, 5th ed. New York, NY, USA: McGraw-Hill, 2020.
[18] T. M. L. Alves, "Benchmark-based software product quality evaluation," Ph.D. dissertation, School Eng., Univ. Minho, Braga, Portugal, 2012.
[19] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.
[20] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow, *Ishikawa Diagram*. Betascript Publishing, 2010.
[21] Y. H. Kwak and F. T. Anbari, "Benefits, obstacles, and future of six sigma approach," *Technovation*, vol. 26, nos. 5–6, pp. 708–715, May 2006.
[22] J. Campos, "Risk management and failure mode and effect analysis for product development," *Rapid Innov. LLC*, 2012.
[23] M. Kalinowski, D. N. Card, and G. H. Travassos, "Evidence-based guidelines to defect causal analysis," *IEEE Softw.*, vol. 29, no. 4, pp. 16–18, Jul. 2012.
[24] D. N. Card, "Defect-causal analysis drives down error rates," *IEEE Softw.*, vol. 10, no. 4, pp. 98–99, Jul. 1993.
[25] D. N. Card, "Defect analysis: Basic techniques for management and learning," *Adv. Comput.*, vol. 65, pp. 259–295, 2005.
[26] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification—A concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.
[27] B. Stoddard and D. Zubrow, "A tutorial for building CMMI process performance models," Softw. Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep., Apr. 2010. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/a558352.pdf
[28] S. Tamura, "Integrating CMMI and TSP/PSP: Using TSP data to create process performance models," DTIC, Document, Fort Belvoir, VA, USA, Tech. Rep. CMU/SEI-2009-TN-033, 2009. [Online]. Available: https://resources.sei.cmu.edu/asset_files/TechnicalNote/2009_004_001_15089.pdf
[29] S. Chatterjee and A. S. Hadi, *Regression Analysis by Example*. Hoboken, NJ, USA: Wiley, 2015.
[30] M. Kuhn and K. Johnson, *Applied Predictive Modeling*. New York, NY, USA: Springer, 2013.
[31] G. A. Seber and C. J. Wild, *Nonlinear Regression*. New York, NY, USA: Wiley, 1989.
[32] C. C. Aggarwal, *Data Classification: Algorithms and Applications*, 1st ed. London, U.K.: Chapman & Hall, 2014.
[33] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola, *Global Sensitivity Analysis: Primer*. Hoboken, NJ, USA: Wiley, 2008.
[34] M. Raza, "Automated software process performance analysis and improvement recommendation," Ph.D. dissertation, Dept. Inform. Eng., Univ. Porto, Porto, Portugal, 2017.
[35] T. Daughtrey, *Fundamental Concepts for the Software Quality Engineer*. Milwaukee, WI, USA: ASQ Quality Press, 2002.
[36] C. Jones, *Software Assessments, Benchmarks, and Best Practices*. Reading, MA, USA: Addison-Wesley, 2000.
[37] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on PSP data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 534–550, Jul. 2009.
[38] L. Breiman, "Classification and regression trees," in *The Wadsworth Statistics/Probability Series*. Kennett Square, PA, USA: Wadsworth International Group, 1984.
[39] C. Shalizi. *Classification and Regression Trees*. Accessed: Dec. 2015. [Online]. Available: http://www.stat.cmu.edu/~cshalizi/ADAfaEPoV/
[40] J. H. Friedman, "Multivariate adaptive regression splines," *Ann. Statist.*, vol. 19, no. 1, pp. 1–67, Mar. 1991.
[41] J. C. Gower, "A general coefficient of similarity and some of its properties," *Biometrics*, vol. 27, no. 4, pp. 857–871, Dec. 1971.
[42] N. Fenton and J. Bieman, *Software Metrics: A Rigorous And Practical Approach*. Boca Raton, FL, USA: CRC Press, 2014.
[43] S. McConnell, *Software Estimation: Demystifying the Black Art*. Redmond, WA, USA: Microsoft Press, 2006.
[44] L. Prechelt and B. Unger, *A Controlled Experiment on the Effects of PSP Training: Detailed Description and Evaluation*. Karlsruhe, Germany: Univ., Fak. Für Informatik, 1999.
[45] M. Kuhrmann, P. Diebold, J. Munch, P. Tell, K. Trektere, F. McCaffery, V. Garousi, M. Felderer, O. Linssen, E. Hanser, and C. R. Prause, "Hybrid software development approaches in practice: A European perspective," *IEEE Softw.*, vol. 36, no. 4, pp. 20–31, Jul. 2019.
[46] L. Bodo, H. C. D. Oliveira, F. A. Breve, and D. M. Eler, "Performance indicators analysis in software processes using semi-supervised learning with information visualization," in *Proc. 13th Int. Conf. Inf. Technol., New Generations*, Las Vegas, NV, USA. Cham, Switzerland: Springer, Apr. 2016, pp. 555–568.
[47] Q. Song, X. Zhu, G. Wang, H. Sun, H. Jiang, C. Xue, B. Xu, and W. Song, "A machine learning based software process model recommendation method," *J. Syst. Softw.*, vol. 118, pp. 85–100, Aug. 2016.
[48] S. Dissanayake and M. Ramachandran, "Machine learning as a service for software process improvement," in *Software Engineering in the Era of Cloud Computing*. Cham, Switzerland: Springer, 2020, pp. 299–326.

MUSHTAQ RAZA received the Ph.D. degree in computer science specifically in software engineering from the Faculty of Sciences, University of Porto, Portugal. He was a Researcher with the Institute for Systems and Computer Engineering, Technology and Science (INESC TEC), Porto, Portugal. He is currently an Assistant Professor of computer science with Abdul Wali Khan University Mardan (AWKUM) and a Research Collaborator with INESC TEC. He has published more than ten papers in renowned journals and conferences in software engineering. His research interests include software process improvement, machine learning, big data analysis, and software engineering. He is a Program Committee Member of ICSSP, top conference in software engineering, and a Focal Person of the National Technology Fund at AWKUM.

JOÃO PASCOAL FARIA (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the Faculty of Engineering, University of Porto (FEUP), in 1999. He is currently an Associate Professor with FEUP, a Senior Researcher with the Institute for Systems and Computer Engineering, Technology and Science (INESC TEC), and the President of the Sectorial Commission for Information and Communications Technology (CS/03) in the scope of the Portuguese Quality Institute (IPQ). He has more than 25 years of research and development experience in software engineering, having published more than 60 papers in several journals and conferences, and obtained 4 best paper awards. His current research interests include automated performance monitoring, analysis and improvement recommendation, test automation for distributed, and heterogeneous and time-constrained systems. He is a member of ACM.