Empirical Evaluation of the ProcessPAIR Tool for Automated Performance Analysis

Mushtaq Raza
INESC TEC/Faculty of Engineering of the University of Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
+351920055092
uomian49@yahoo.com

João Pascoal Faria
INESC TEC/ Faculty of Engineering of the University of Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
+351225081400
jpf@fe.up.pt

Rafael Salazar
Tecnológico de Monterrey
Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849, Monterrey, Nuevo Léon, Mexico
+528183582000
rafael.salazar@itesm.mx

***Abstract*— Software development processes 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 that analysis manually is challenging because of the potentially large amount of data to analyze and the effort and expertise required. ProcessPAIR is a novel tool designed to help developers analyze their performance data with less effort, by automatically identifying and ranking performance problems and potential root causes. The analysis is based on performance models derived from the performance data of a large community of developers. In this paper, we present the results of an experiment conducted in the context of Personal Software Process (PSP) training, to show that ProcessPAIR is able to accurately identify and rank performance problems and potential root causes of individual developers so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused.**

Keywords- Automatic Performance Analysis; Performance Analysis Tool; Personal Software Process; Empiric Assessment.

I. INTRODUCTION

Development processes making intensive use of metrics and quantitative methods, such as the Team Software Process (TSP) [1] and Personal Software Process (PSP) [2], can generate large amounts of data that can be periodically analyzed by developers to identify their performance problems, determine root causes and devise improvement actions [3]. Although tools exist to automate data collection and produce performance charts and reports for manual analysis of TSP/PSP data [4][5][6], practically no tool support exists to automate developer performance analysis. The manual analysis of performance data for determining root causes of performance problems and devising improvement actions is challenging because of the amount of data to analyze [3] and the effort and expertise required.

To address those shortcomings, in previous work [7][8] we developed models, techniques, and tools to automate the analysis of performance data produced in the context of high maturity software development processes. The developed ProcessPAIR tool, available freely in http://blogs.fe.up.pt/processpair/, is able to automatically identify and rank performance problems and potential root causes of individual developers in their performance data.

In the current paper, we focus on the empirical assessment of ProcessPAIR approach and tool, through a case study in which we analyze the performance data and analysis reports produced by several PSP trainees at Tecnológico de Monterrey, Mexico. Specific objectives and research questions are presented in Section III.

Section II presents some background information on our approach and tool. Sections III, IV, V and VI present the case study planning, performance model preparation, performance analysis, and results and discussion. Section VII presents some related work and section VIII presents final conclusions and future work.

II. BACKGROUND

Our 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 (TSP, PSP, or other). In our approach, a PM comprises a set of performance indicators (PIs) organized hierarchically by cause-effect relationships [8]. Examples are given in section IV.

2. *Calibrate*: The PM is automatically calibrated based on the performance data of many process users. The statistical distribution of each PI and statistical relations between PIs are computed from the data set [8].

3. *Analyze*: Once a PM is defined and calibrated, the performance data of individual developers can be automatically analyzed with ProcessPAIR, to identify and rank performance problems and root causes.

DOI reference number: 10.18293/SEKE2016-205

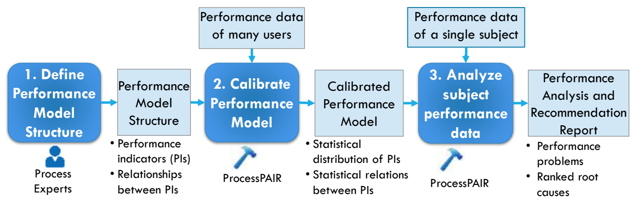

Performance data of many users
Performance data of a single subject
1. Define Performance Model Structure
Process Experts
Performance Model Structure
• Performance indicators (PIs)
• Relationships between PIs
2. Calibrate Performance Model
ProcessPAIR
Calibrated Performance Model
• Statistical distribution of PIs
• Statistical relations between PIs
3. Analyze subject performance data
ProcessPAIR
Performance Analysis and Recommendation Report
• Performance problems
• Ranked root causes

Figure 1. The ProcessPAIR approach.

Further details about each step are given next.

A. *Model definition*

The firsts step in our approach is the definition (as a tool extension) of the following elements of the PM:

- list of relevant PIs, including a formula for its computation from base measures;
- subset of top-level PIs;
- cause-effect relationships between PIs, determined by a formula or statistical evidence;
- sensitivity coefficients [9] between PIs related by a formula.

The performance model defined for analyzing PSP performance data will be presented in section IV.A.

B. *Model calibration*

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

- approximate statistical distribution of each PI, represented by a cumulative distribution function;
- recommended performance ranges for each PI;
- sensitivity coefficients between PIs not related by an exact formula.

The approximate cumulative distribution function of each PI is computed by linear interpolation between a few percentiles computed from the training data.

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

Sensitivity coefficients between PIs not related by an exact formula are computed by first determining a linear regression equation from the training data and subsequently computing the corresponding sensitivity coefficient.

The data set used for calibration in the case study will be described in section IV.B.

C. *Performance analysis*

Having defined and calibrated the performance model, the performance data of individual developers can be automatically analyzed by ProcessPAIR, to identify and rank performance problems and potential causes of individual developers. The results of the analysis are presented in multiple views, as shown in section V.B.

III. CASE STUDY PLANNING

D. *Objectives and research questions*

The overall objective of the case study is to assess whether ProcessPAIR is able to accurately identify performance problems of individual developers and their potential causes in the context of PSP training so that subsequent manual analysis for the identification of deeper causes and remedial actions can be properly focused and effort can be saved.

In PSP training, students develop a sequence of projects, with the stepwise introduction of the following practices: performance measurement (based on size, effort and defects); size and effort estimation; coding standards; design and code reviews; design templates and design verification; quality management [2].

More specifically, the goal of the case study is to answer the following research questions:

- **RQ1** (*problem identification*): Is it possible to automatically analyze the performance data of an individual PSP developer in order to identify performance problems, with similar results but less effort than in manual analysis?
- **RQ2** (*root cause identification*): Is it possible to automatically analyze the performance data of an individual PSP developer in order to determine the root causes of the identified performance problems, with similar results but less effort than in manual analysis?

E. *Performance data under analysis*

The subject data under analysis is based on a data set from Tecnológico de Monterrey, in Mexico, referring to 10 subjects (students) that developed 6 projects each using the PSP, in the scope of the "Software Quality and Testing" course in 2015. The subjects used Process Dashboard (http://www.processdash.com/) for collecting the standard PSP base measures. In the end of the sequence of projects, the subjects analyzed their personal performance in those projects and documented their findings and improvement proposals in a Final Report (written in Spanish).

F. *Performance analysis procedures*

Two of the authors of this paper, not involved in PSP training in Tec de Monterrey, both fluent in English and one with a good reading understanding of Spanish, translated into English and analyzed the final reports (in both English and Spanish), in order to extract relevant information for comparison with the tool-based analysis. Results from the tool-based analysis for each subject were effortlessly obtained by uploading the performance data stored in Process Dashboard to ProcessPAIR. The extracted results from the final reports and from the tool for each subject were then collected into an appropriate table, as illustrated in Section VI. Subsequently, the results were classified according to the categories defined in Section V and statistics were computed shown in VI.

IV. PERFORMANCE MODEL PREPARATION

A. *Performance model definition*

To best fit the specific context of PSP training in Tec de Monterrey, we used the PSP performance model defined in our previous work [8] with minor changes. The full set of top-level and nested PIs can be seen in the first column of Figure 2. We consider three top-level PIs regarding predictability, quality, and productivity.

The major predictability PI in the PSP is the *Time Estimation Accuracy*, which we measure by the ratio between actuals and estimates. Since in the PSP's PROBE estimation method [2], a time (effort) 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 consider that the *Time Estimation Accuracy* is affected by the *Size Estimation Accuracy* and the *Productivity Estimation Accuracy.* Hence, the latter PIs are presented in Figure 2 as child nodes of the *Time Estimation Accuracy*. The rational for further drilling down the *Productivity Estimation Accuracy* can be consulted in [8].

Product quality is usually measured by post-delivery defect density [10]. However, since the scope of the PSP is the development of small programs or components of large programs, post-delivery defects are seldom known. The PSP proposes an aggregated quality measure—the *Process Quality Index (PQI)*—that constitutes an effective predictor of post-delivery defect density [2][11]. Hence, we use the PQI as the top-level quality indicator to analyze. The PQI is computed based on five components, which are presented in Figure 2 as factors that affect the PQI. The exact formula can be consulted in [8], as well as the rational for further drilling down these PIs.

In the PSP, productivity is usually measured in lines of code per hour, in spite of known limitations [10]. Since in the PSP time is recorded per process phase, when a productivity problem is encountered one can analyze the productivity per phase, to determine the problematic phase(s). Hence, Figure 2 shows a set of PIs for the productivity per phase, which together affect the overall productivity. Exact formulas can be consulted in [8], as well as the rational for further drilling down these PIs.

B. *Performance model calibration*

To calibrate the performance model, we used a large PSP data set from the Software Engineering Institute (SEI) 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. The calibration is performed automatically by the tool; the user has just to provide an input file with the data set.

V. PERFORMANCE ANALYSIS

A. *Manual performance analysis*

Regarding RQ1 (problem identification), based on the information available in the final report of each subject, we produced a table with a synthesis of cases in which the subject explicitly indicated bad performance or good performance in a PI for a specific project or overall (summary). Examples are shown in Table III, together with the support citations extracted from the final report.

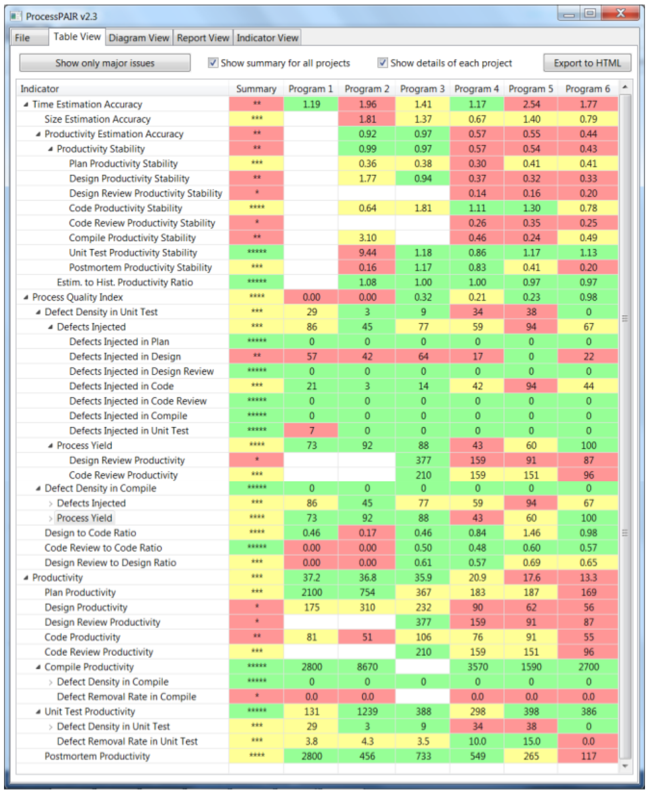
Regarding RQ2 (root cause identification), for each case in which the subject explicitly indicated bad performance and corresponding causes, we filled in an additional column with the causes mentioned by the subject, as illustrated in Table III.

B. *Automatic performance analysis*

The results from the tool-based performance analysis for each subject were effortlessly obtained by uploading the performance data stored in Process Dashboard to ProcessPAIR. The results of the analysis are presented by the tool in multiple views.

The relevant view for problem identification is the *Table View*. This view presents the detailed evaluation of all PIs for all projects of the subject under analysis, as depicted in Figure 2. Each cell is colored green, yellow or red, in case, its value suggests no performance problem, a potential performance problem, or a clear performance problem, respectively. A cell is colored green (red) if its value lies within the range of the best (worst) 1/3 values in the calibration data. Cells with missing data are left blank. For example, the red cells in Figure 2 suggest that the main problems with time estimation accuracy occur in projects P2, P5, and P6. By expanding the nodes in this view, one can drill down to lower level PIs, following the hierarchical structure of the performance model, in order to identify potential causes of performance problems. For example, the red colored cells in Figure 2 suggest that the time estimation problem in P2 is caused by a size estimation problem.

The *Diagram View* (see Fig. 3) helps identifying and prioritizing, project by project, the causes of performance problems. The child indicators are sorted according to the value of a ranking coefficient representing a cost-benefit estimate that relates the cost of improving the value of the child indicator with the benefit on the value of the parent indicator [8]. For example, the diagram of Fig. 3 suggests that the major cause for the poor productivity in project 5 is the poor productivity in the Design phase, followed by the Design Review, Plan, and Code phases.



☑ Show summary for all projects ☑ Show details of each project

Indicator	Summary	Program 1	Program 2	Program 3	Program 4	Program 5	Program 6
Time Estimation Accuracy	**	1.19	1.96	1.41	1.17	2.54	1.77
Size Estimation Accuracy	***		1.81	1.37	0.67	1.40	0.79
Productivity Estimation Accuracy	**		0.92	0.97	0.57	0.55	0.44
Productivity Stability	**		0.99	0.97	0.57	0.54	0.43
Plan Productivity Stability	***		0.36	0.38	0.30	0.41	0.41
Design Productivity Stability	**		1.77	0.94	0.37	0.32	0.33
Design Review Productivity Stability	*				0.14	0.16	0.20
Code Productivity Stability	****		0.64	1.81	1.11	1.30	0.78
Code Review Productivity Stability	*				0.26	0.35	0.25
Compile Productivity Stability	**		3.10		0.46	0.24	0.49
Unit Test Productivity Stability	*****		9.44	1.18	0.86	1.17	1.13
Postmortem Productivity Stability	***		0.16	1.17	0.83	0.41	0.20
Estim. to Hist. Productivity Ratio	*****		1.08	1.00	1.00	0.97	0.97
Process Quality Index	****	0.00	0.00	0.32	0.21	0.23	0.98
Defect Density in Unit Test	***	29	3	9	34	38	0
Defects Injected	***	86	45	77	59	94	67
Defects Injected in Plan	*****	0	0	0	0	0	0
Defects Injected in Design	**	57	42	64	17	0	22
Defects Injected in Design Review	*****	0	0	0	0	0	0
Defects Injected in Code	***	21	3	14	42	94	44
Defects Injected in Code Review	*****	0	0	0	0	0	0
Defects Injected in Compile	*****	0	0	0	0	0	0
Defects Injected in Unit Test	*****	7	0	0	0	0	0
Process Yield	****	73	92	88	43	60	100
Design Review Productivity	*			377	159	91	87
Code Review Productivity	***			210	159	151	96
Defect Density in Compile	*****	0	0	0	0	0	0
Defects Injected	***	86	45	77	59	94	67
Process Yield	****	73	92	88	43	60	100
Design to Code Ratio	****	0.46	0.17	0.46	0.84	1.46	0.98
Code Review to Code Ratio	*****	0.00	0.00	0.50	0.48	0.60	0.57
Design Review to Design Ratio	***	0.00	0.00	0.61	0.57	0.69	0.65
Productivity	***	37.2	36.8	35.9	20.9	17.6	13.3
Plan Productivity	***	2100	754	367	183	187	169
Design Productivity	*	175	310	232	90	62	56
Design Review Productivity	*			377	159	91	87
Code Productivity	**	81	51	106	76	91	55
Code Review Productivity	***			210	159	151	96
Compile Productivity	*****	2800	8670		3570	1590	2700
Defect Density in Compile	*****	0	0	0	0	0	0
Defect Removal Rate in Compile	*	0.0	0.0		0.0	0.0	0.0
Unit Test Productivity	*****	131	1239	388	298	398	386
Defect Density in Unit Test	***	29	3	9	34	38	0
Defect Removal Rate in Unit Test	***	3.8	4.3	3.5	10.0	15.0	0.0
Postmortem Productivity	****	2800	456	733	549	265	117

Figure 2. An example of problem identification (Table View).

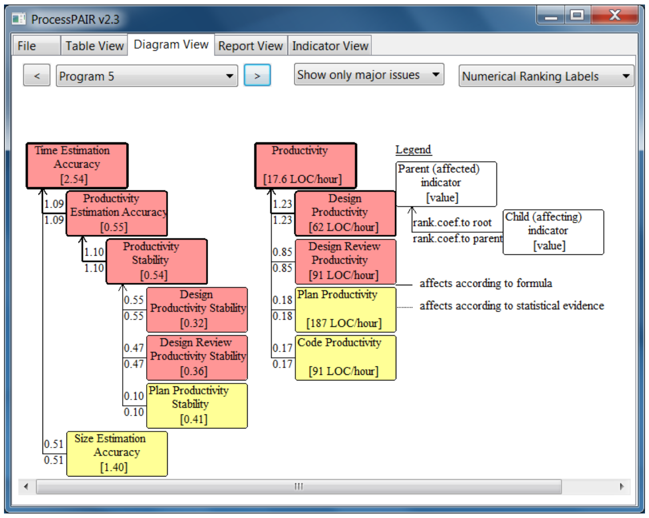

ProcessPAIR v2.3
File
Table View
Diagram View
Report View
Indicator View
<
Program 5
>
Show only major issues
Numerical Ranking Labels
Time Estimation Accuracy [2.54]
1.09
1.09
Productivity Estimation Accuracy [0.55]
1.10
1.10
Productivity Stability [0.54]
0.55
0.55
Design Productivity Stability [0.32]
0.47
0.47
Design Review Productivity Stability [0.36]
0.10
0.10
Plan Productivity Stability [0.41]
0.51
0.51
Size Estimation Accuracy [1.40]
Productivity [17.6 LOC/hour]
1.23
1.23
Design Productivity [62 LOC/hour]
0.85
0.85
Design Review Productivity [91 LOC/hour]
0.18
0.18
Plan Productivity [187 LOC/hour]
0.17
0.17
Code Productivity [91 LOC/hour]
Legend
Parent (affected) indicator [value]
rank.coef.to root
rank.coef.to parent
Child (affecting) indicator [value]
affects according to formula
affects according to statistical evidence

Figure 3. An example of root cause identification (Diagram View).

C. *Comparison*

The results extracted from the performance analysis report (manual analysis) and from the tool (automatic analysis), were compared as illustrated in Table III.

Regarding problem identification (RQ1), we considered that there is a match when the developer explicitly indicated bad performance and the tool indicated a clear (red) or potential (yellow) performance problem. False positives occur when the developer explicitly indicates good performance, but the tool indicates a clear or potential performance problem. False negatives occur when the developer explicitly indicates bad performance, but the tool indicates no performance problem.

Regarding root causes identification (RQ2), we encountered three kinds of situations:

- *same causes*: the developer and tool indicate the same causes;
- *deeper manual analysis*: the tool accurately points out intermediate causes, and the developer points out deeper causes;
- *faults in manual analysis*: the developer overlooked important causes or pointed out erroneous causes.

VI. RESULTS AND DISCUSSION

A. *Statistics*

From the table produced in the previous step (as illustrated by the excerpts in Table I), we computed the statistics shown in Table I and Table II.

Regarding problem identification (RQ1), among the 187 cases analyzed, there are 180 matches (96%), with only 6 false positives (3%) and 1 false negative (1%). The false positives and the false negative correspond to boundary situations similar to the one illustrated in Table I.

Regarding root causes identification (RQ2), Table II is self-explanatory. From the 116 cases in which the developers explicitly indicated bad performance, only in 52 cases (with some examples in Table III) the developers pointed out root causes.

TABLE I. PROBLEM IDENTIFICATION STATISTICS.

		Automatic analysis		
		Green	***Yellow***	***Red***
Manual Analysis	***Bad***	1 **(1% false negative)**	45 (match)	70 (match)
	Good	65 (match)	6 **(3% false positives)**	0

TABLE II. ROOT CAUSES IDENTIFICATION STATISTICS.

Classification	**Absolute frequency**	**Relative frequency**
Same causes *(tool benefit: eliminate manual effort)*	10	19%
Developer pointed out deeper causes (tool pointed out intermediate causes) *(tool benefit: reduce manual effort)*	28	54%
Faults in manual analysis *(tool benefit: prevent user errors)*	14	27%
Total	**52**	**100%**

TABLE III. EXCERPTS OF THE PROBLEM AND ROOT CAUSES IDENTIFICATION AND COMPARISION TABLE.

Data point		Manual performance analysis		Automatic performance analysis		Comparison	
Top-level Indicator	***Subject, Project***	***Problem identification***	***Root causes identification***	***Problem identif-ication***	***Root causes identification (with ranking coefficient)***	***Problem identif-ication***	***Root causes identif-ication***
Time Estimation Accuracy	S1, P2	Bad performance: "there are two that stand out for being very large, the program 2 and (…)"	"I attribute the bad (time) estimation to the (bad) size estimation"	Red	Size Estimation Accuracy (1.0)	Match	Same causes
Time Estimation Accuracy	S1, P1	Good performance: "the closest estimates were the program 1 and 4, with 18.90% and 17.20%"	-	Green	-	Match	-
Defects Injected	S1, Summary	Bad performance: "almost all programs have 50 to 100 errors per KLOC, which can improve"	"most defects injected in Design (mainly of type Function), followed by Code"	Yellow	Defects Injected in Design (12.4), Defects Injected in Code (0.8)	Match	Deeper manual analysis (1)
Size Estimation Accuracy	S2, P6	Bad performance: "the last program fall out of the (desired) range of 10% error"	-	Green	-	False negative (2)	-
Time Estimation Accuracy	S3, P5	Good performance: "for program 4 and up (…) the estimation error is approximately within a range of -10% to 10%, which (…) is a good range"	-	Yellow	-	False positive (3)	-
Productivity	S4, P4	Bad performance: "Program 4 represents a significant (productivity) downward"	"due to some defects (…) not identified in time, resulting in a time consuming testing phase"	Red	Code Productivity (62.7), Code Review Productivity (4.8)	Match	Faults in manual analysis (4)

(1) The manual and automatic analysis coincide regarding the identification and prioritization of the problematic defect injection phases, with the quantitative prioritization in the automatic analysis. Additionally, the developer point out the most problematic defect type (Function).
(2) This false negative corresponds to a boundary situation. The actual size estimation error was approximately -15%, which is the threshold considered by the tool to distinguish green and yellow regarding size estimation. On the other hand, the developer considered an abnormally tight range of +-10%.
(3) This false positive corresponds to a boundary situation. The actual time estimation error was approximately -13%, which is the threshold considered by the tool to distinguish green and yellow regarding time estimation. On the other hand, the developer was 'benevolent' in his analysis, by considering -13% to be approximately within the +-10% range.
(4) Data shows that defects were removed in Code Review, not in Unit Test, and that much more time was spent in Code Review than in Unit Test.

B. *Answers to the research questions*

Regarding RQ1 – "Is it possible to automatically analyze the performance data of an individual PSP developer in order to identify performance problems, with similar results but less effort than in manual analysis", we conclude that, in this case study, the automatic analysis produces similar results (without essentially any manual effort), with very few false positives (3%) and false positives (1%) corresponding to boundary situations.

Regarding RQ2 – "Is it possible to automatically analyze the performance data of an individual PSP developer in order to determine the causes of the identified performance problems, with similar results but less effort than in manual analysis", the results in Table II show that, in the cases in which the manual analysis was not faulty (we found faults in 27% of the cases!), the tool-based analysis was able to point out the same causes as the ones found by the developers in their manual analysis (19% of the cases) or was able to point out intermediate causes in the same direction as the deeper causes identified in manual analysis (54%) of the cases. Hence, regarding RQ2, we conclude that the automatic analysis was able to identify either the same causes or causes in the same direction as the manual analysis.

Overall, the benefits of the tool-based analysis are:

- it can correctly identify the performance problems, saving manual effort;
- it can correctly identify causes for the identified performance problems, so that subsequent manual analysis for searching deeper causes can be properly focused, reducing the overall manual effort needed and the errors in manual analysis.

C. *Limitations and threats to validity*

In the case study presented, the conclusions obtained by the model-based analysis are very close to the ones obtained by the developers in their manual analysis. This suggests that our approach can be helpful in performance analysis and process improvement, by pointing out the areas to focus on manual analysis. However, further experiments need to be conducted to quantify the effort savings that can be achieved by conducting performance analysis with the help of our tool from the beginning.

Although our approach and tool are general and can be instantiated for any development process, the model and experiment described in this paper refer only to PSP performance data. We intend to replicate our approach to other development processes without having such a well-defined measurement framework as the PSP, but we expect to encounter difficulties regarding data availability, data quality, and standardization.

VII. RELATED WORK

Our approach draws inspiration from existing work on process performance models (PPM) [9][12], benchmark-based approaches for software product evaluation [13], and defect causal analysis (DCA) techniques [14].

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 performance data, and calibrated using collected process and product measures [15]. The main difference is that our performance model conveys additional elements needed to identify performance problems (in the outcomes) and rank potential root causes (factors): recommended ranges for each PI; approximate statistical distribution of each PI; sensitivity coefficients (derived from exact or regression equations).

In our approach, in order to enable the automated identification of performance problems, after deciding on the relevant PIs, one has to decide on the relevant ranges. Our approach for defining such ranges draws inspiration from the benchmark-based approach developed by researchers of the Software Improvement Group [13][16] to rate the maintainability of software products, with adaptations for process evaluation instead of product evaluation.

The DCA approach [14] is essentially complementary to our approach. The main advantage of our approach is that it has the potential to identify relevant performance problems and causes in a fully automatic way, so that subsequent manual activities can be conducted in a more focused and efficient way, to further determine root causes and devise improvement actions.

VIII. CONCLUSIONS AND FUTURE WORK

The results of the case study show that the ProcessPAIR tool is able to accurately identify performance problems of individual PSP developers and potential causes for those problems. Hence, subsequent manual analysis for the identification of deeper causes and remedial actions can be properly focused, reducing the overall effort and possible errors in performance analysis.

As future work, we plan to build a comprehensive catalogue of improvement actions to recommend for the highest-ranked causes, build similar models for analyzing performance data produced in the context of other development processes, and conduct further experiments for assessing the effort gains with our tool.

ACKNOWLEDGMENTS

The authors would like to acknowledge the SEI and Tec de Monterrey for facilitating the access to the PSP data for performing this study. This work is partially financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia as part of project UID/EEA/50014/2013 and research grant SFRH/BD/85174/2012.

REFERENCES

[1] Davis, N., and Mullaney, J. 2003. *The Team Software Process (TSP) in Practice: A Summary of Recent Results*. CMU/SEI-2003-TR-014.

[2] Humphrey, W. 2005. *PSP^{sm}: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional.

[3] Burton, D. and Humphrey, W. 2006. Mining PSP Data. In *TSP Symposium 2006 Proceedings*.

[4] The Software Process Dashboard Initiative home page. http://www.processdash.com/.

[5] Philip, J., Kou, H., Agustin, J., Christopher, C., Moore, C., Miglani, J., Zhen, S., Doane, W. 2003. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In *ICSE 2003*. Portland, Oregon.

[6] Shin, H., Choi, H., and Baik, J. 2007. Jasmine: A PSP Supporting Tool. In *Proc. of the Int. Conf. on Software Process* (ICSP 2007), LNCS 4470, Springer-Verlag, 73-83.

[7] Raza, M., Faria, J. 2014. A Model for Analyzing Estimation, Productivity and Quality Performance in the Personal Software Process. In Proc. of the 2014 Int. Conf. on Software and System Process (ICSSP 2014), ACM, 10-19.

[8] Raza, M., Faria, J. 2015. A Model for Analyzing Performance Problems and Root Causes in the Personal Software Process. *Journal of Software: Evolution and Process*, John Wiley & Sons

[9] Saltelli, A., Chan, K., Scott, E. M. 2008. Sensitivity Analysis, Wiley.

[10] Jones, C. 2010. Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies. McGraw-Hill.

[11] Humphrey, W. 2009. *The Software Quality Profile*. White Paper, SEI.

[12] Tamura, S. 2009. Integrating CMMI and TSP/PSP: Using TSP Data to Create Process Performance Models. CMU/SEI-2009-TN-033.

[13] Alves, T., Ypma , C., Visser, J. 2010. Deriving Metric Thresholds from Benchmark Data. In *2010 IEEE International Conference on Software Maintenance (ICSM),* 1-10.

[14] Card, D.N. 2005. Defect Analysis: Basic Techniques for Management and Learning. *Advances in Computers*, vol. 64, 259-295, Elsevier.

[15] Chrissis, M. B., Konrad, M., Shrum, S., 2003. CMMI: Guidelines for Process Integration and Product Improvement, 2nd Edition. Addison-Wesley.

[16] Alves, T. 2012. Benchmark-based Software Product Quality Evaluation. Ph Thesis. U. Minho