Journal of Software Engineering Research and Development, 2020, 8:5, doi: 10.5753/jserd.2020.546

Software Operational Profile *vs.* Test Profile: Towards a better software testing strategy

Luiz Cavamura Júnior [**Federal University of São Carlos** | ***luiz_cavamura@ufscar.br***]
Ricardo Morimoto [**Federal University of São Carlos** | ***rmmorimoto@gmail.com***]
Sandra Fabbri [**Federal University of São Carlos** | ***sfabbri@ufscar.br***]
Ana C. R. Paiva [**School of Engineering, University of Porto & INESC TEC** | ***apaiva@fe.up.pt***]
Auri Marcelo Rizzo Vincenzi [**Federal University of São Carlos** | ***auri@ufscar.br***]

Abstract The Software Operational Profile (*SOP*) is a software specification based on how users use the software. This specification corresponds to a quantitative representation of the software that identifies its most used parts. As software reliability depends on the context in which users operate the software, the *SOP* is used in software reliability engineering. However, there is evidence of a misalignment between the software tested parts and the *SOP*. Therefore, this paper investigates a potential misalignment between *SOP* and the tested software parts to obtain more evidence of this misalignment based on experimental data. We performed a set of Experimental Studies – EXS to verify: a) whether there are significant variations in how users operate the software; b) whether there is a misalignment between the *SOP* and the tested software parts; c) whether failures occur in untested *SOP* parts in case of misalignment; d) whether a test strategy based on the amplification of the existent test set with additional test data generated automatically can contribute to reduce the misalignment between *SOP* and untested software parts. We collected data from four software while users were operating them. We analyzed this data to reach the goals of this work. The results show that there is significant variation in how users operate software and that there is a misalignment between *SOP* and the tested software parts after evaluating the four software studied. There is also indication of failures in the untested *SOP* parts. Although the aforementioned test strategy has reduced the potential misalignment, the test strategy is not enough to avoid it, thus indicating a need for specific test strategies using *SOP* as a test criterion. These results indicate that *SOP* is relevant not only to software reliability engineering but also to testing activities, regardless of the adopted testing strategy.

Keywords: *Software Quality, Software Testing, Operational Profile, Test Profile*

1 Introduction

Software users provide relevant data related to the many possible ways they explore a given software feature. We create software based on the expression of the creative nature of our intellect (Assesc, 2012). Using their previous professional experience, this same creative aspect allows software users to adapt to different ways of using the software due to changes in the process initially supported by the program (Sommerville, 1995). This feature makes software functionalities parameterizable to meet specific and particular needs, even if they are designed to meet business rules that are common to many organizations.

The Software Operational Profile (*SOP*) corresponds to the manner in which a given user operates the software. The *SOP* may be quantitatively characterized by assigning a probabilistic distribution to the software operations, showing what users use the most in software (Musa, 1993; Gittens et al., 2004; Sommerville, 1995). A given user may not reproduce the same failure identified by another one. The reason for this is that software can have many different operational profiles and experienced users can adapt how they operate the software. As such, software quality is dependent on its operational use (Cukic and Bastani, 1996).

A survey by Cukic and Bastani (1996) states that information about *SOP* is considered either essential or relevant to issues related to activities inherent to software development. Examples of these questions are: “Which are the most used parts of the software?”; “How do users use the application?”; “What are the software usage patterns?”; and “How does test coverage correspond to the code that was indeed executed by users?”. Additionally, Rincon (2011) analyzed a set of ten open-source software and, in only one of them, the available functional test set reached a code coverage close to 70%. Even if this interval level of code coverage is considered acceptable, there is a significant percentage of untested code which may be related to critical features for the majority of software users. This fact highlights the possibility of a misalignment between the tested parts and the parts that users effectively use. Thus, there are indications of the relevance of *SOP* in ensuring software quality and also in evidencing a possible misalignment between *SOP* and the tested software parts (Rincon, 2011; Begel and Zimmermann, 2014). This misalignment can often lead to failures when operating the software.

The term misalignment refers to the potential dissonance between the software tested parts and the *SOP*, which corresponds to the software parts most used by users. Thus, it represents situations in which the *SOP* or parts of the *SOP* may not have been previously executed by the software test suite, indicating that the adopted test strategy may not be aligned with the user’s interests in terms of software functionality.

Therefore, this study investigates a potential misalignment between the tested software parts and *SOP*. The research results, based on a set of Experimental Studies (EXS), provide the following contributions:

1. Evidence that there are significant variations in how users operate software, even when they perform the same operations, i.e., there are different software usage patterns;
2. Evidence of a possible misalignment between *SOP* and software testing;
3. Evidence that there are faults concentrated on untested parts of the software;
4. Definition and introduction of the term "test profile";
5. Evidence that even when using an automated test generator to extend an existent test set the misalignment between the *SOP* and the tested parts of the software has little improvement.

In addition, the related studies briefly present the results obtained by a Systematic Review of the Literature (*SLR*), which we carried out before the execution of the experimental studies. These results show that, to the best of our knowledge, there is no previous study with the same purpose as this one (Cavamura Júnior et al., 2020). We adapted the methodology proposed by Mafra et al. (2006) to plan and perform the activities described in this paper.

The remaining of this paper is as follows: Section 2 presents concepts related to the definition of *SOP*. Section 3 describes the adopted methodology for this study. Section 4 presents the related studies identified and selected by the *SLR* (Cavamura Júnior et al., 2020). Section 5 describes the results of the experimental studies. Section 6 presents some lessons learned with the results. Section 7 presents threats to validity. Lastly, Section 8 describes the conclusions and future work.

2 Software Operational Profile (SOP)

SOP is a way to obtain a specification of how users operate software (Musa and Ehrlich, 1996; Sommerville, 1995). Musa (1993) proposed one of the most relevant approaches for *SOP* registration. Musa (1993) defines *SOP* as a quantitative characterization based on the way software is operated. This definition corresponds to software operations, to which an occurrence probability is assigned. An operation corresponds to a task performed by the software. We delimit this operation by external factors related to software implementation.

Software operations can present different behavior and, consequently, provide different results. In this way, there are different possible execution paths, depending on the given input data. These different ways of execution are named execution types. In Figure 1 we present an example of software operations and their respective execution types.

Input data, which characterize an execution type, create a data set named input state ("IS" in Figure 1). Input states, associated with execution types, form the software input space. As input states characterize the execution types of an operation, the input space can be fractioned by operations, associating an input state set in each operation, named operation domain. Thus, it is possible to assign an input domain to each software operation ("ID" in Figure 1) that determines how the software executes the operation; i.e., the input do-

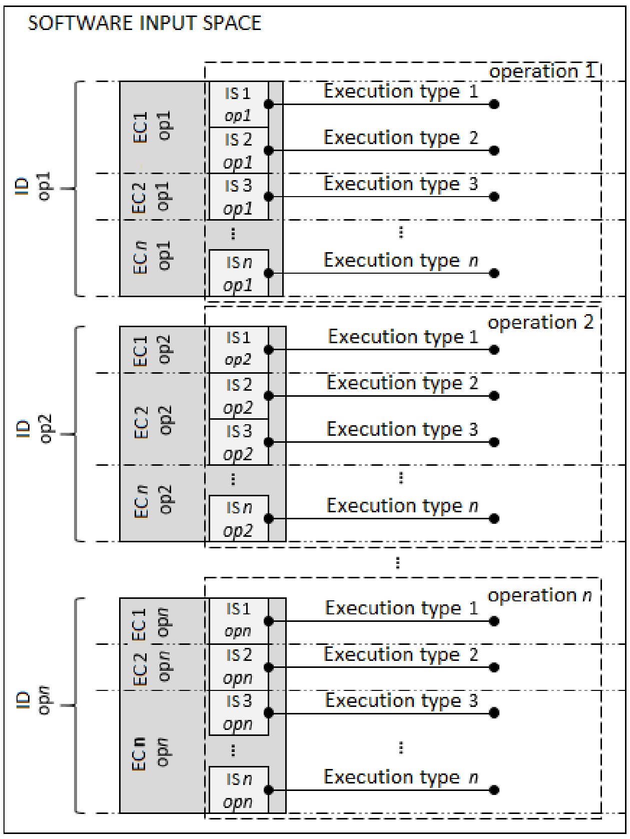


Figure 1. Concepts involved in the definition of the operational profile.

main elements (input states) determine the execution type of an operation.

In Figure 1 are shown: i) the input states, identified by "$IS_1, IS_2, IS_3, \ldots, IS_n$", ii) the software input space, and iii) the input domain of each operation, identified by "$IDop_1, IDop_2, \ldots, IDop_n$".

Although the operation set available in software is finite, the execution types correspond to a set with infinite elements, given that the input domain can be infinite. Thus, assigning an occurrence probability to execution types is possible since we can partition the input domain into sub-domains. Each generated sub-domain corresponds to an execution category. These categories group the execution types whose different input states produce the same behavior in operation.

In Figure 1 we present the execution categories, identified by "$EC_1, EC_2, \ldots, EC_n$", which divide the input domain of each operation and group the execution types with the same behavior. In Figure 1 we present the existing relation between operational concepts, execution types, input state, input space, input domains, and execution categories.

In Musa (1994, 1993) studies, the author assigns an occurrence probability to the execution categories in order to obtain a quantitative characterization of the software corresponding to the operational profile. The data used to get the occurrence probabilities of operation can be obtained from log files generated by previous version of the software or from similar software (Musa, 1993; Takagi et al., 2007). Developer expectations can also determine these probabilities (Takagi et al., 2007).

In the context of this study, the term granularity corresponds to the level of fragmentation (be it conceptual or structural) we use to assign an occurrence probability or execution frequency to the generated software fragments. Then, it is possible to identify the most used software parts when users are operating the software, i.e., the *SOP*. According to the object-oriented programming paradigm, subprograms correspond to the methods implemented in data structures, called classes. Thus, the methods in this paradigm represent actions assigned to the operations performed by the software.

As *SOP* is a software specification based on how users operate software (Musa and Ehrlich, 1996; Sommerville, 1995), showing the software parts most used by users, the *SOP* in the context of this paper corresponds to the frequency of the processed methods while the software is performed by users, thus indicating the most operated software parts.

2.1 The *SOP* and the Software Quality

Pressman (2010) defines software quality as an effective process of creating a valuable product for those who produce it and will use it. Thus, software quality can be subjective in that it depends on the point of view of who is analyzing the software's characteristics. Considering the user's point of view, for example, software of quality is software that meets its needs and is easily operated (Falbo, 2005). However, from a developer's point of view, software of quality is e.g., one that demands less maintenance effort.

Software reliability corresponds to the probability of a software operation occurring without any occurrence of failure in a specified period and in a specific environment (Musa, 1979; Cukic and Bastani, 1996). Thus, as software reliability depends on the context in which software is used, software reliability meaning software maintainability and efficiency (among others) is one of the software's attributes related to software quality, and it represents the user's point of view on software quality (Musa, 1979; Bittanti et al., 1988).

Since the *SOP* represents the way software will be used by its users and considers software reliability as dependent on the context in which users operate the software, *SOP* can support activities related to the reliability of software engineering. Thus, the purpose of *SOP* is to generate test data that reproduces the way software is executed in its production environment, ensuring the validity of reliability indicators (Musa and Ehrlich, 1996).

In the software reliability process, a usage model representing the *SOP* is created to design test cases and perform the test activity. The elements constituting the usage model correspond to the adopted granularity to determine the *SOP*, whose execution frequencies or occurrence probability identify the most used software parts.

In the literature, studies using models representing *SOP* in their testing techniques have classified these techniques as statistical testing, statistical use testing, reliability testing, model-based testing, use-based testing and *SOP*-based testing (Poore et al., 2000; Kashyap, 2013; Sommerville, 2011; Pressman, 2010; Musa and Ehrlich, 1996).

It is worth noting that the frequency with which a fault becomes apparent during the software operation is more significant for users than the remaining faults (Takagi et al., 2007) and a defect affecting reliability for one user may never be revealed to another who has a different work routine (Sommerville, 2011). The use of *SOP* does not guarantee the detection of all faults, but it ensures that the most used software operations are tested (Ali-Shahid and Sulaiman, 2015).

2.2 Problems related to the use of *SOP*

Although the *SOP* can be obtained from log files recording events that occur in the operating software, in previous versions of the software, in similar software and even from the developers' experience (Musa, 1993; Takagi et al., 2007), there are several problems related to the identification of the *SOP* reported in the literature.

In this study, we observed that the use of an instrumented version of the software to identify the *SOP* of the data collected during operation of software by users affects the performance of operating the software and generates a large volume of data. According to Namba et al. (2015), the effort to identify the *SOP* depends on the complexity of the software. Other kinds of problems are also reported in the literature. Thus, reports of difficulties and issues related to *SOP* identified in the literature are relevant and will be addressed in possible test approaches defined according to the results presented in this paper. Table 1 summarizes the main challenges and problems identified.

3 Research Methodology

The results presented in this paper are part of a PhD Project (Cavamura Júnior, 2017) that follows the methodology proposed by Mafra et al. (2006). The methodological steps proposed by Mafra et al. (2006) were instantiated into the context of the research presented in this article. This methodology is an extension of the methodology proposed by Shull et al. (2001) for introducing software processes. The methodology proposed by Mafra et al. (2006) is shown in Figure 2.

We defined five research questions to guide our investigation in this paper:

- RQ_1: Are there other studies with the same goal or similar goals whose results provide the contributions proposed in this paper?
- RQ_2: Are there any relevant variations in how users operate software?
- RQ_3: Is there misalignment between *SOP* and the tested software parts?
- RQ_4: Given the misalignment between *SOP* and the tested software parts, do the failures occur in the untested *SOP* parts?
- RQ_5: Given the misalignment between *SOP* and the tested software parts, can a test strategy including automated test data[1] generator contribute to reduce the misalignment?

To answer RQ_1 and considering the methodology presented in Figure 2, the step "Secondary Study" included a Systematic Mapping Study (*SMS*) and a Systematic Literature Review (*SLR*) to identify studies whose contributions were similar or equivalent to the research contribu-

[1] In the remaining of the paper, we use test data to refer to inputs automatically generated.

Table 1. Problems related to SOP.

Reference	Year of publication	Reported problem
(Cukic and Bastani, 1996)	1996	Identifying the *SOP* is difficult because it requires predicting software usage.
(Leung, 1997)	1997	Estimation errors and *SOP* changes are inevitable when software is operated in a production environment.
(Shukla, 2009)	2009	Studies related to SOP focus on exploring software operations. The parameters of these operations are little explored.
(Sommerville, 2011)	2011	Software reliability depends on the context in which software will be used. Experienced users can constantly adapt their behavior regarding software usage.
(Namba et al., 2015)	2015	*SOP* identification requires a lot of effort, making this activity difficult depending on the complexity of the software.
(Fukutake et al., 2015)	2015	The probability of use decreases when the software usage model has multiple states.
(Bertolino et al., 2017)	2017	SOP-based testing can be saturated and lose effectiveness because it focuses only on failures most likely to occur.

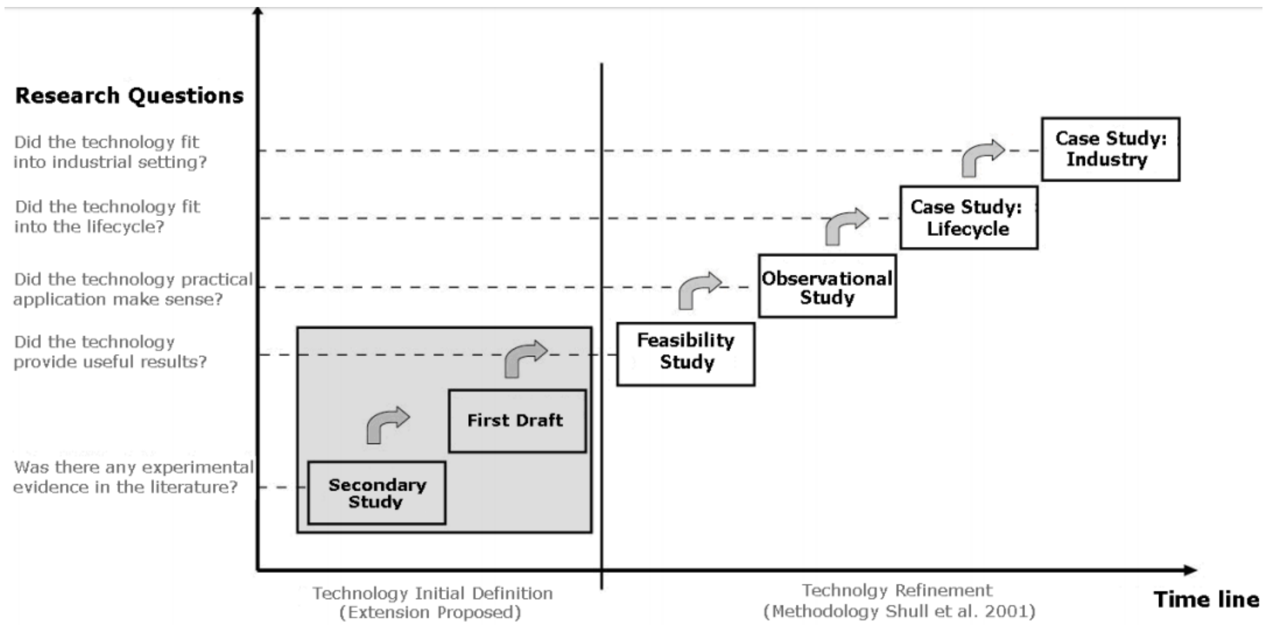

Research Questions
Did the technology fit into industrial setting?
Did the technology fit into the lifecycle?
Did the technology practical application make sense?
Did the technology provide useful results?
Was there any experimental evidence in the literature?
Secondary Study
First Draft
Feasibility Study
Observational Study
Case Study: Lifecycle
Case Study: Industry
Technology Initial Definition (Extension Proposed)
Technolgy Refinement (Methodology Shull et al. 2001)
Time line

Figure 2. Adopted Research Methodology (extracted from Travassos et al. (2008))

.

tions reported in this article and, thus, evaluate its originality. The results obtained from the *SMS* are available elsewhere at `http://lcvm.com.br/artigos/anexos/jserd2020/cap-3-rs-ms.pdf`. Also, a detailed description of the *SLR* can be found elsewhere in (Cavamura Júnior et al., 2020). We present a brief description of the main results of both *SMS* and *SLR* in Section 4.

The "First Draft" stage comprised the planning of the experimental studies presented in this study. We adopted the model proposed by the GQM (Basili et al., 2002)'s technique to guide the planning of this research. The instantiated model for the planning phase is presented in Table 2.

The "Feasibility Study", "Observational Study" and "Case Study: Lifecycle" stages comprised the accomplishment of a set of *EXS* subdivided into four activities (*AT*) associated with the research questions, called $EXS\text{–}AT_1$, $EXS\text{–}AT_2$, $EXS\text{–}AT_3$, and $EXS\text{–}AT_4$. The purpose of each activity and the research questions associated with each one of them are summarized in Table 3.

To perform the *EXS* activities we instrumented four software, S_1, S_2, S_3 and S_4, to collect data that allowed us to identify the *SOP* for each individual user. Table 4 shows the characterizations of used software and associates them with the *EXS* activities.

Table 2. Exploratory Study Planning.

Stage	Analyze	For the purpose of	Focus	Perspective	Context
1 (RQ_1)	studies that addressed the use of *SOP*	Check whether there are researches for the same or similar purposes	Answered base on a previous work	Software test researchers	Software applications users
2 (RQ_2, RQ_3, RQ_4 , RQ_5)	the *SOP*	**(a)** Check if there are significant variations; **(b)** Check if there is a misalignment; **(c)** Show the occurrence of failures; **(d)** Check if the insertion of additional test data, generated automatically by EvoSuite, can contribute to reduce the misalignment	**(a)** The way software is operated by its users **(b)** *SOP* and tested software parts **(c)** *SOP*'s parts not tested **(d)***SOP* and tested software parts	Software test researchers	Software applications users

Table 3. Research Activities.

Activity	Purpose	Question
SMS/SLR	Evaluate research originality (Cavamura Júnior et al., 2020)	RQ_1
EXS–AT$_1$	Check for relevant variations in how the users operate the software	RQ_2
EXS–AT$_2$	Find out through the *SOP* and the software's test suite whether there is a misalignment between *SOP* and the tested parts of the software	RQ_3
EXS–AT$_3$	Once we confirm the misalignment between *SOP* and the tested parts of the software, check if there is any failure in the *SOP*'s parts not tested	RQ_4
EXS–AT$_4$	Check whether a test strategy, based on the amplification of the existent test set with additional test data automatically generated, can contribute to reducing the misalignment between the *SOP* and the tested parts of the software	RQ_5

The "Feasibility Study" stage comprised the accomplishment of *EXS–AT*$_1$. The "Observational Study" stage comprised the accomplishment of *EXS−AT*$_2$, *EXS−AT*$_3$, and *EXS−AT*$_4$ based on operational profiles collected from S_1 and S_2. The "Case Study: Lifecycle" stage comprised the accomplishment of *EXS–AT*$_2$, *EXS–AT*$_3$, and *EXS–AT*$_4$ again but based on operational profiles collected from S_3 and S_4. The "Case Study: Industry" stage is in progress and its results will be published in a future work.

Once the methodology was defined, this study was planned in two stages to provide answers for the research questions. The research questions associated with these stages is shown in the "Stage" column of Table 2.

- *Stage 1*: performing an *SMS* and an *SLR*;
- *Stage 2*: performing the *EXS* composed of four activities: *EXS–AT*$_1$, *EXS–AT*$_2$, *EXS–AT*$_3$, and *EXS–AT*$_4$.

The focus of this paper is on Stage 2 of Table 2, i.e., the set of *EXS* we performed to obtain evidence of the possible misalignment between *SOP* and the tested software parts. The other kinds of experiments were also carried out as part of the ongoing work (Cavamura Júnior, 2017).

In Section 4, we present a brief description of the main findings of the *SLR*. An interested reader can find more information elsewhere (Cavamura Júnior et al., 2020). In Section 5, the *EXS* and their respective results are described.

4 Related Work

We conducted *SMS* and *SLR* (Stage 1 of Table 2) to provide the theoretical basis and evidence of the originality of this study. The *SMS* process together with the *SLR* process consist of the planning, conducting and results publishing phases (Nakagawa et al., 2017). A detailed description of the *SMS*, *SLR* and their respective detailed results can be found at `http://lcvm.com.br/artigos/anexos/jserd2020/cap-3-rs-ms.pdf` and at (Cavamura Júnior et al., 2020), respectively.

We conducted a *SMS* to: i) verify how the distribution of primary studies related to *SOP* in software engineering areas is characterized; ii) acquire knowledge of the contributions provided by the use of *SOP* in the areas of software engineering, focusing on the software quality field. iii) check if the use of *SOP* in quality assurance activities has been a topic of interest to researchers.

The *SMS* found 4726 studies, of which we selected 182 for data extraction. The distribution of the primary studies in software engineering areas is shown in Figure 3. After we

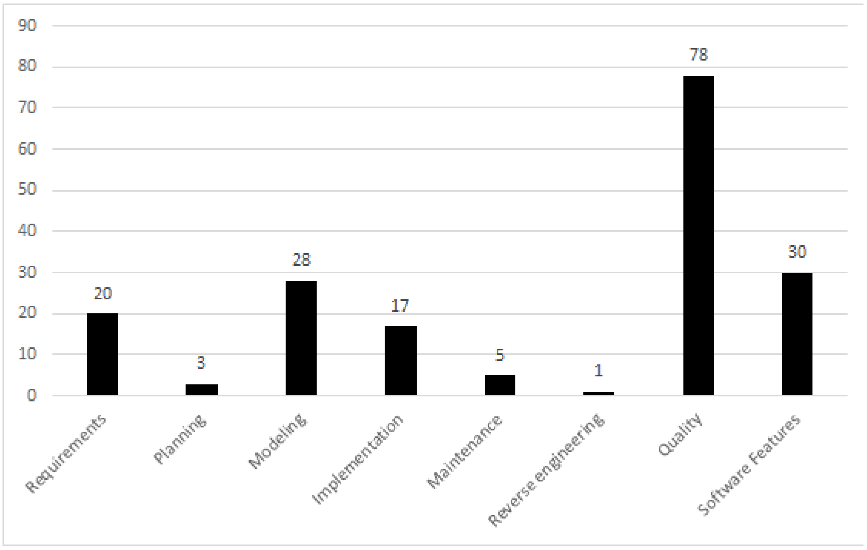


Figure 3. Distribution of the studies in software engineering areas.

analyzed the extracted data, we concluded that software quality is the most explored area by studies that used *SOP* as a resource in the strategies addressed in these studies. Most of these strategies are associated with software reliability. Although software quality is the most approached area, we found some studies related to software testing. Thus, this scenario evidences a gap in the software quality field, mainly in its subareas that are not associated with software reliability. Therefore, the results of the *SMS* motivated us on conducting the *SLR*, whose purpose was to identify, analyze and understand the studies whose contributions are similar or equivalent to the contributions of the research reported in this paper, i.e identify, analyze and understand the studies that used *SOP* as evaluation criteria to check is there a possible misaligned between *SOP* and tested software parts (Cavamura Júnior et al., 2020).

At the end of the *SLR* (Cavamura Júnior et al., 2020), as highlighted in Figure 4, we observed only three studies closest to ours: Bertolino et al. (2017), Chen et al. (2001), and Amrita and Yadav (2015), briefly described next. Figure 4 shows the number of processed studies by *SLR*. The interested reader may find additional information about the complete *SLR* protocol elsewhere (Cavamura Júnior et al., 2020).

Bertolino et al. (2017) mention the test based on the operational profile can suffer saturation and loss of effectiveness since it focuses on the occurrence of most likely failures. Thus, to improve software reliability, the test should also focus on faults with a low probability of occurrence. In this context, Bertolino et al. (2017) present an adaptive and iterative software testing technique based on *SOP*. In the first iteration, the authors selected the test cases following a traditional test based on operational profile, i.e., the authors randomly selected the test cases according to the occurrence probability of each partition of the software input domain under test. In each subsequent iteration, the technique: a) calculates the number of ideal test cases to be selected for each partition, and; b) selects, prioritizes and executes the number of test cases.

Bertolino et al. (2017) obtained a probability calculation to represent how much the partition test will contribute to program reliability. Based on this information, Bertolino et al. (2017) determine the optimal amount of test cases for testing each partition.

In this probability calculation, Bertolino et al. (2017) considered the failure rate and the occurrence probability of each partition. The failure rate is the ratio of the number of failed test cases and the number of test cases assigned to the partition. Thus, Bertolino et al. (2017) obtained the occurrence probability from *SOP*. To select and prioritize test cases, the frequency with which the program parts are exercised when running the tests is obtained from the previous iterations. As the focus of Bertolino et al. (2017)'s approach is to select test cases covering portions of the program that are poorly exercised, test cases associated with the uncovered parts of software have high priority.

We can determine software reliability by the time elapsed between the detected faults. In this way, Chen et al. (2001)'s technique considers the context in which a test suite can overestimate software reliability when it is not able to detect new faults due to the use of an obsolete *SOP*. The more redundant the test cases are about the covered code, the more overestimated will be the reliability of the software. Thus, this technique adjusts the time interval between failures when running redundant test cases. Chen et al. (2001)'s identified the redundant test cases through coverage analysis during the execution of the tests.

According to Amrita and Yadav (2015), researchers have approached the selection of test cases based on *SOP*, but the authors did not find much discussion about the infrequent software parts. Amrita and Yadav (2015) propose a model that provides the flexibility to allocate test cases according to the priority defined by *SOP* and by the experience of the testing team. Based on this information, Amrita and Yadav (2015)'s model selects test cases using *fuzzy* logic.

We observed that Bertolino et al. (2017), Amrita and Yadav (2015) addressed the use of *SOP* in the selection and prioritization of test cases, focusing on those software parts whose operation is infrequent. Chen et al. (2001)'s study addressed the selection of test cases, using *SOP* to identify redundant test cases and treat them in the process of software

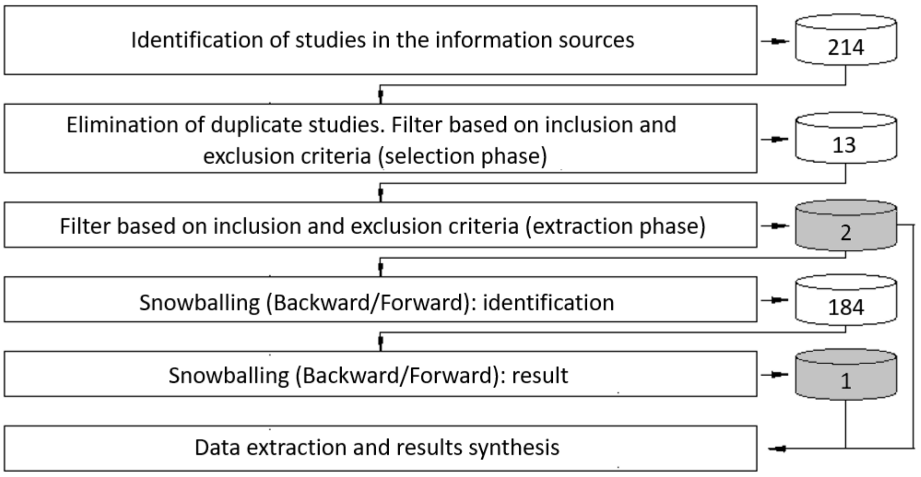

Identification of studies in the information sources
214
Elimination of duplicate studies. Filter based on inclusion and exclusion criteria (selection phase)
13
Filter based on inclusion and exclusion criteria (extraction phase)
2
Snowballing (Backward/Forward): identification
184
Snowballing (Backward/Forward): result
1
Data extraction and results synthesis

Figure 4. Processed studies by *SLR*.

reliability, and thus obtain more accurate reliability. Nevertheless, the studies identified and processed by *SLR* did not directly investigate in their approaches whether there is a misalignment between the existing test suite and *SOP*, thus providing an answer to research question RQ_1. We believe the selection and prioritization activities will not be productive if we do not align test cases with *SOP*.

5 Experimental Studies (EXS)

The studies by Begel and Zimmermann (2014) and Rincon (2011), briefly described in Section 1, provided initial evidence about the possible misalignment between the tested software parts and the *SOP*. We performed the *EXS* to obtain empirical data that, after analyzed, could provide answers to the research questions RQ_2, RQ_3, RQ_4, and RQ_5, thus resulting in more evidence, based on experimental data, on the possible misalignment between the tested software parts and the *SOP*. As described in Section 3, we defined four activities for the *EXS*, named *EXS–*AT_1, *EXS–*AT_2, *EXS–*AT_3, and *EXS–*AT_4. In order to perform these activities, we instrumented four software, S_1, S_2, S_3 and S_4, to collect data that allowed to identify the *SOP* for each software during its operation by users. S_1, S_2, S_3 and S_4 were implemented under the object-oriented programming paradigm. A characterization of the software used and their association to the activities of the *EXS* is presented in Table 4.

During these activities, users had to perform tasks at a given period when they were operating S_1, S_2, S_3, and S_4. Thus, we collected data automatically in an attempt to obtain the operational profile of the software used. In the following subsections, we describe the strategy adopted for the data collection, the activities of the *EXS*, and their results.

5.1 Strategy for data collection

In each activity, we instrumented the S_1, S_2, S_3, and S_4 software to collect data during their operation by the users participating in the activity. We adopted aspect-oriented programming (Ferrari et al., 2013; Laddad, 2009; Rocha, 2005), which allows us to obtain information and to manipulate specific software parts without modifying the implementation of the S_1, S_2, and S_3. For S_4, we developed a monitoring tool using the *javassist* framework. The *javassist* allows for the manipulation of Java bytecode. This feature allowed us to monitor S_4 execution and collect S_4 information while participants were operating it. Although the aspect-oriented paradigm makes it possible to perform the instrumentation without modifying the source code of the software, it requires the created aspects to be compiled together with the software for instrumentation. *Javaassist* was adopted to perform the instrumentation without having to compile the software that is to be instrumented.

We defined the strategy for data collection and applied it at the subprogram level. The developed tool and the instrumentation collect information about the methods execution of S_1, S_2, S_3, and S_4'. From that information, we obtained the execution frequency of the processed methods during the S_1, S_2, S_3, and S_4 software execution in the activities.

5.2 *EXS–*AT_1: Evaluating the variation in how software is operated by users

We performed the *EXS*$-AT_1$ activity to evidence whether there are relevant variations in how users operate the software to carry out the same task. To measure this variation, we obtained the *SOP* used in this activity for each user through data coming from the instrumented S_1 software.

In order to reduce the risks associated with the threats to validity of the activity, 30 undergraduate students of the Computer Science and Computer Engineering courses participated in this activity. These participants had equivalent experience and knowledge. We trained the participants in an attempt to make them familiar with S_1 and the concepts involved with its use. Additionally, we assigned the same task to the participants in this activity. We assigned to each participant the task of inspecting the Java source code of S_1 Project, named *Software Under Inspection* (*SUI*), considering an object-oriented paradigm. We set a time limit for participants to complete the task. The tasks performed within the defined time period were considered successfully completed. Thus, data obtained from all participants were used in the activity.

Table 4. Characterization of the software used in the *EXS*.

Software	Purpose	Source	Methods	Test cases	Origin of test cases	Usage
S_1	Provide software inspection support (*Crista*)	closed source	2749	716	computational tool	$EXS{-}AT_1$, $EXS{-}AT_2$
S_2	Bibliographic reference management (*JabRef*)	open source	7100	514	Community	$EXS{-}AT_2$, $EXS{-}AT_3$, $EXS{-}AT_4$
S_3	Process Automation (developed on demand)	closed source	869	351	Test team	$EXS{-}AT_2$
S_4	CASE tool (*ArgoUml*)	open source	18099	2272	Community	$EXS{-}AT_2$, $EXS{-}AT_3$, $EXS{-}AT_4$

We stored the data collected by the instrumentation of S_1 and we, subsequently, analyzed it. Through this data, we identified the *SOP* of each participant. It is worth noting that all participants have the same goal and artifacts to conclude the task. In the following subsections, we describe the analysis of the collected data and the results obtained by the activity *EXS–AT*$_1$.

5.2.1 *EXS–AT*$_1$: Data Analysis

We grouped the data collected by the $EXS{-}AT_1$ activity according to the participant who originated them; that is, for each participant, we obtained and recorded information about the execution of the S_1 methods, allowing to compute the execution frequency of the methods.

To identify the variations in how users operate S_1, we created a representation of the operational profile of S_1 for each participant. Each representation corresponds to a homogeneous one-dimensional data structure that recorded the execution frequency of each method in S_1 for each participant during the execution of the task. The structure elements represent the methods implemented in S_1, regardless of whether they were executed during the activity or not. Thus, each structure was composed of 2749 elements corresponding to the 2749 methods implemented in S_1 (Table 4). For each of these elements, we assigned the execution frequency of the method when performing the activity. For non-executed methods, we assigned the numeric value 0. Figure 5 presents a graphical representation of the data structure corresponding to a part of the S_1 profile. We show some elements (M_1, M_2, M_3, ..., M_{2749}). Each element corresponds to an implemented method of S_1. The number in the cells represents the execution frequency of a given method for a given participant after concluding an activity. Thus, according to Figure 5, four methods (M_1, M_3, M_{2748}, and M_{2749}) were not executed during the activity, while the remaining ones (M_2, M_{100}, M_{101}, and M_{102}) were executed 500, 10000, 15725 and 87000 times, respectively.

As the variations in how users operate S_1 depends on the processed volume, the processed volume for each participant was measured. The S_1 software is a computational tool that provides support for the inspection activity of source code based on the *stepwise abstraction* reading technique. The purpose of the *stepwise abstraction* reading technique is to determine the program's functionality according to the functional abstractions generated by the source code (Linger et al., 1979).

The S_1 software analyzes the *SUI* and, for each class, generates a treemap visual metaphor providing a simple mode to visualize the source code. The code blocks are represented by rectangles disposed hierarchically. These rectangles are named *declarations* on the tool context. When a *declaration* is selected the respective source code is shown to make the inspection and to register the functional abstraction for that *declaration*. A functional abstraction is an annotation inserted by S_1 user that represents the pseudo-code with respect to the selected *declaration*.

During the S_1 operation, for each inspected class the S_1 user assigns a functional abstraction for each *declaration* identified by the tool in the class, identifying that the *declaration* was inspected. The discrepancies found during the inspection process are recorded in a similar manner in the tool, i.e., assigning the discrepancy to the *declaration*. Figure 6 shows an S_1 user interface during a class inspection.

S_1 provided metrics that allowed us to measure the processing volume generated by each participant. In this activity, the volume of processing corresponds to the number of functional abstractions attributed to each class that structurally composes the *SUI* as well as to the number of discrepancies found in each class. Thus, it was possible to determine which classes and how much of each class were inspected by each participant. It should be noted that the same tool configuration parameters were applied to all participants.

In an attempt to obtain homogeneity in the processing volume generated by each participant, we grouped them according to the generated processing volume. An indicator was calculated to represent the processing volume generated by each participant. The indicator corresponds to the ratio between the sum of abstractions and discrepancies of all classes of one participant by the sum of *declarations* of all classes. For instance, the total of inspected software *declarations* was 1526. Among the participants, the largest amount of the functional abstractions and discrepancies registered by one par-

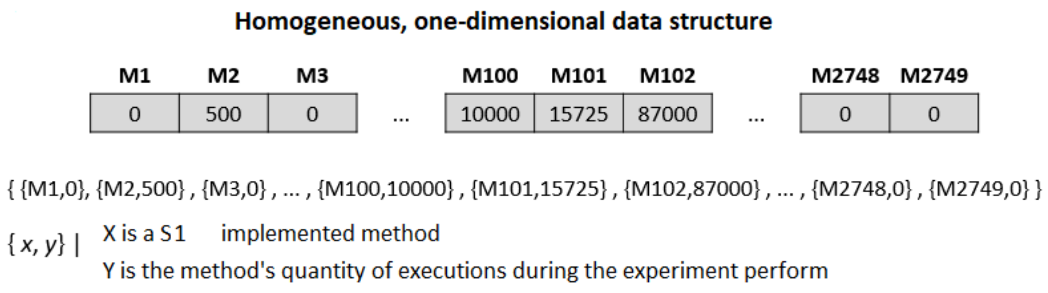
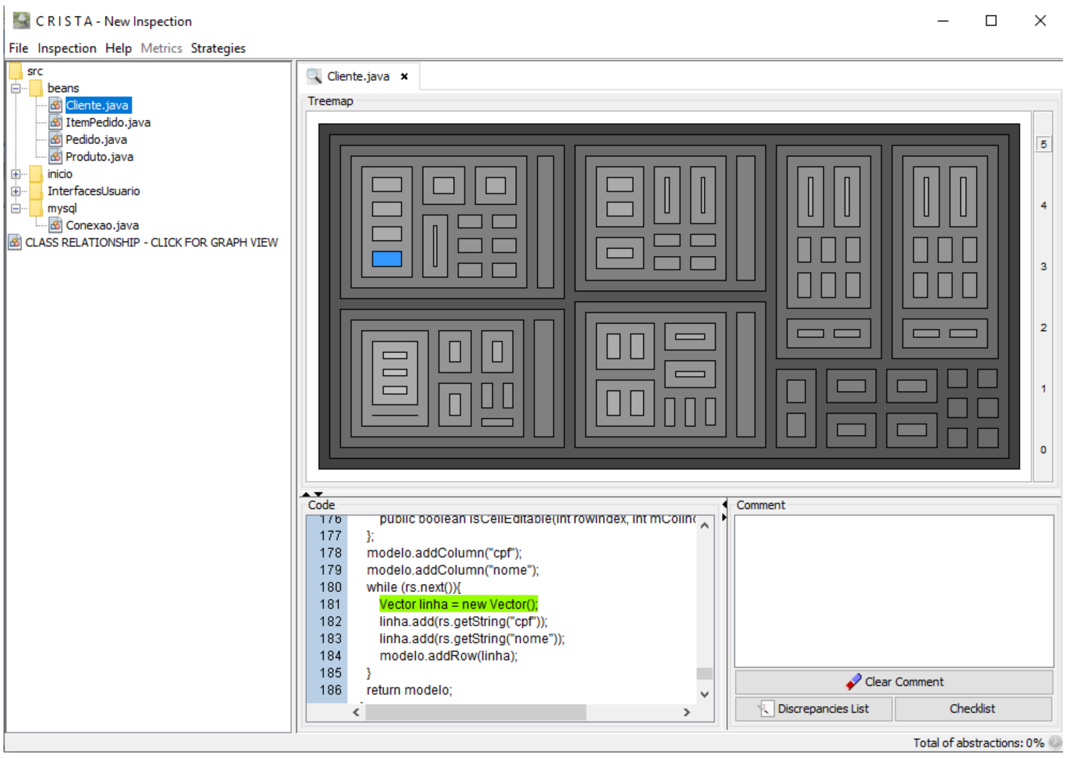

Homogeneous, one-dimensional data structure
M1
M2
M3
M100
M101
M102
M2748
M2749
0
500
0
...
10000
15725
87000
...
0
0
{ {M1,0}, {M2,500} , {M3,0} , ... , {M100,10000} , {M101,15725} , {M102,87000} , ... , {M2748,0} , {M2749,0} }
{ x, y} | X is a S1 implemented method
Y is the method's quantity of executions during the experiment perform

Figure 5. Graphical representation of the data structure.



```
        public boolean isCellEditable(int rowIndex, int mColIn
    };
    modelo.addColumn("cpf");
    modelo.addColumn("nome");
    while (rs.next()){
        Vector linha = new Vector();
        linha.add(rs.getString("cpf"));
        linha.add(rs.getString("nome"));
        modelo.addRow(linha);
    }
    return modelo;
```

Figure 6. S_1 user interface.

ticipant was 284. For this participant the indicator value was 0.186 (284/1526).

The corresponding calculated indicator classified participants. This classification allowed us to identify 3 groups of participants with similar indicator value. In other words, we assigned participants who demanded similar processing volume and resulted in the same group. In Table 5 we show the created groups.

Table 5. Groups of participants in activity $EXS\text{–}AT_1$.

Group	Participants
A	P10, P11, P12, P13, P30
B	P4, P5, P6, P7, P8, P9, P24, P25, P26, P27, P28, P29
C	P1, P2, P3, P15, P16, P17, P18, P19, P20, P21, P22, P23, P14

According to Table 5, 30 individuals participated in the experiment. Group A comprises the data obtained by 5 participants; group B compiles the data obtained by 12 participants, and group C compiles the data obtained by 13 participants.

We compared the representations of the operational profile of S_1 to highlight the variations concerning how the users operate the software. This comparison is possible through the data structures corresponding to these representations. Thus, we considered the same group of participants when we performed this comparison.

As previously described, homogeneous one-dimensional data structures were used to generate the operational profile representations of S_1. The elements that constitute these data structures represent the methods implemented in S_1, and their stored values correspond to the execution frequency. As the number of elements and their association to the methods of S_1 are common to these structures, we compared the data stored in them, that is, the execution frequency of each method of S_1. We compared each element of a data structure to the corresponding element of a different data structure. Thus, each representation contained in a group was compared with all other representations contained in the same group. As an example, we compared the representation of the S_1 operational profile generated by the data collected by participant $P10$ to the ones generated by the participants $P11$, $P12$, $P13$ and $P30$ (Table 5).

We defined an indicator to measure the variations in the execution frequency of each method among the representations. The value of this indicator ranges from 0 to 1. The value of this indicator represents the difference between the method execution frequency, stored in an element of one representation, with the method execution frequency, stored in the respective element in another representation. The indicator is calculated for each comparison made between the elements of one representation with the respective elements of another representation. The indicator value corresponds to the ratio between the difference resulting from the compared frequencies and the highest compared frequency.

In Figure 7 we illustrate the systematic approach to compare the representations of the operational profile of S_1.

Figure 7 evidences that: a) the closer to 1 is the value of the indicator, the higher the difference between the execution frequencies of the evaluated method; b) the closer to 0 the value of the indicator is, the lower the difference between the execution frequencies of the evaluated method. Indicators whose value was equal to 0 denote the participants did not execute that particular method during the accomplishment of the activity. Indicators whose values were equal to 1 denote methods executed by only one participant during the activity.

Table 6 shows the results of the comparison between the operational profile of S_1 for each participant of Group A.

Table 6. Comparison among participants in group A .

ID	P-1	P-2	DMF	IM
01	*P10*	*P11*	59	0.37
02	*P10*	*P12*	42	0.47
03	*P10*	*P13*	39	0.62
04	*P10*	*P30*	77	0.53
05	*P11*	*P12*	45	0.59
06	*P11*	*P13*	80	0.56
07	*P11*	*P30*	68	0.65
08	*P12*	*P13*	73	0.51
09	*P12*	*P30*	57	0.38
10	*P13*	*P30*	92	0.43

The value in the column "ID" in Table 6 corresponds to a comparison identification made between two representations of the operational profile of S_1. The values in the columns "P-1" e "P-2" refer to the identification of the participants whose collected data gave rise to the representations of the operational profile of S_1. The value contained in the "DMF" column refers to the number of methods whose indicator value was equal to 1. The values in column "IM" refer to the average value of the indicators originated by the differences between the execution frequencies recorded in the representations of the operational profile (Figure 7). As an example, the result obtained from the comparison between the representations of the operational profile of S_1 obtained from participants $P12$ and $P13$ (line 08 of Table 6) indicates that 73 methods were performed only by one of the participants, $P12$ or $P13$. The results of the comparisons also indicate that, on average, the execution frequency of the methods differs by 0.51 for the compared participants, i.e., the frequency of these methods is approximately 50% higher for one of the participants.

We created a graphical representation to facilitate the distinction in the operational profile, considering two different participants. As an example, in Figure 8 we illustrate the results from the comparison of the operational profile representations obtained from $P12$ and $P13$. In the graphical representation, each array element represents a method. The information displayed in each element refers to the value obtained for the indicator which quantifies the variation between the execution frequencies of the represented method. Methods whose value is one (1) were registered in only one of the operational profile representations of S_1 (cells painted black in the graphic representation illustrated by Figure 8). The methods whose value obtained by the indicator was between 0.5 (inclusive) and 1 (exclusive) were painted gray in the graphic representations shown in Figure 8. The other methods whose value obtained for the indicator were below 0.5 were painted white in the graphic representation shown by Figure 8.

5.2.2 *EXS–AT*$_1$: Results

We verified significant differences in the execution frequency of methods for S_1 when the participants were operating it. The methods not executed during the activity also had a significant difference between participants. The average value of the indicator used to measure the variations in the execution frequencies of each method was 0.51 for participants of Group A. For this same group, the average value in the number of methods whose execution was registered in only one of the representations of the comparisons was 63.2. These averages for the participants of Group B and Group C were, respectively, 0.5/66.19 and 0.57/43.75. Given the *EXS–AT*$_1$ results, significant variations were verified among the representations of operational profiles, thus providing an answer to research question *RQ*$_2$.

5.3 *EXS–AT*$_2$: *SOP vs.* Test Profile

We performed the *EXS–AT*$_2$ activity to obtain evidence of the possible misalignment between *SOP* and the tested software parts. In an attempt to verify a misalignment between *SOP* and the tested software parts, we evaluated the operational profile of S_1, S_2, S_3, and S_4, along with their test suites. We obtained the operational profile of S_1 during *EXS–AT*$_1$. The same procedure we performed to identify the operational profile of S_1 we also applied for S_2, S_3, and S_4. As stated in Session 5.1, we instrumented S_2 and S_3 to collect data when users operated the software. These data allowed us to identify the *SOP* of S_2 and S_3. The operational profile of the S_4 software was identified with use of a tool to monitor S_4 execution.

Undergraduate students of the Technology in Analysis and Development Systems course participated in the activity as S_2 users. Thus, we trained the participants, who had equivalent experience and knowledge, to use S_2. We repeated the same process above, but now with Postgraduate students of the Web Software Development course, who also had equivalent experience and knowledge to participate in the activity as S_4 users. In addition, public servants participated in the ac-

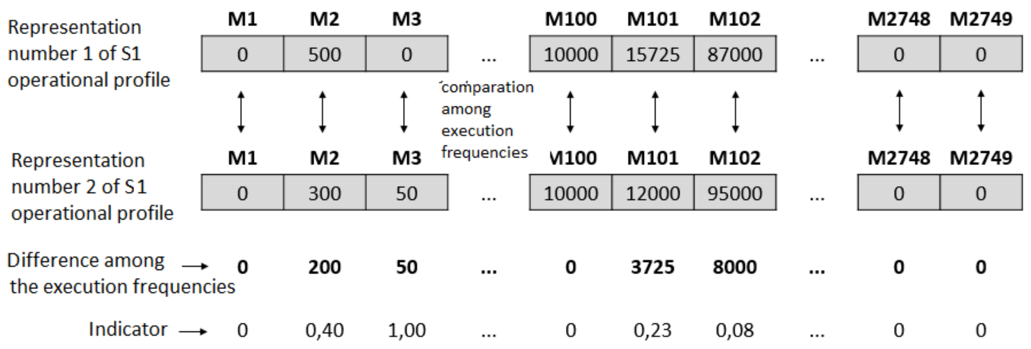

Representation number 1 of S1 operational profile
M1 M2 M3 ... M100 M101 M102 ... M2748 M2749
0 500 0 ... 10000 15725 87000 ... 0 0
comparation among execution frequencies
Representation number 2 of S1 operational profile
M1 M2 M3 ... M100 M101 M102 ... M2748 M2749
0 300 50 ... 10000 12000 95000 ... 0 0
Difference among the execution frequencies
0 200 50 ... 0 3725 8000 ... 0 0
Indicator
0 0,40 1,00 ... 0 0,23 0,08 ... 0 0

Figure 7. Systematic to measure variances.

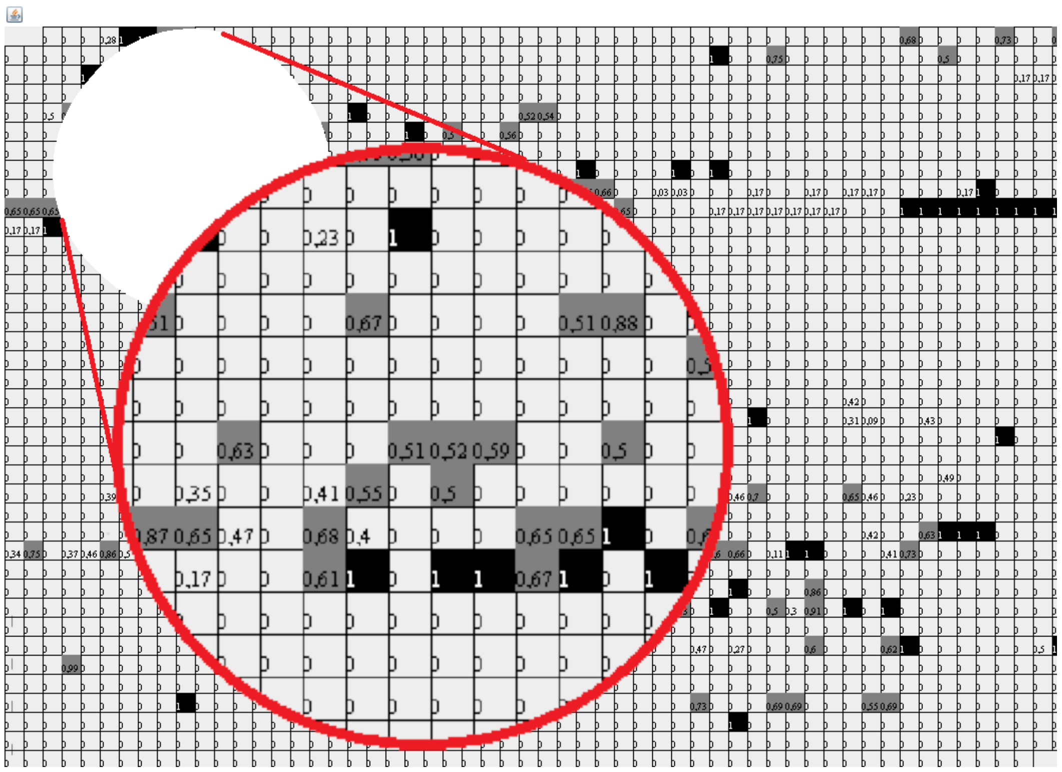


Figure 8. Differences between the $P12$ and $P13$ representations.

tivity as S_3 users performing their daily tasks using the software features. The task assigned to S_2 users was to operate S_2 to record 10 bibliography references. The task assigned to S_4 users was to operate S_4 to create a class diagram from a given software requirement specification. We set a time limit for the S_2, S_3, and S_4 users to perform the task. The tasks performed within the defined period were considered successfully completed, thus data obtained of all participants were used in the activities. S_2, S_3, and S_4 users obtained similar performance and results in their respective performed tasks.

In addition to the data that identified the *SOP* of S_1, S_2, S_3, and S_4, we collected data about the test suite execution of these software to obtain evidence of the mismatch between *SOP* and the tested software parts. The same procedure used to collect the data that provided the *SOP* was used to collect data during the execution of the test suites. These data allowed us to obtain the *test profile* of S_1, S_2, S_3, and S_4. We defined the term *"test profile"* in this paper as the software parts executed after the test suite run.

Note that the test cases of the used software had different origins (as shown in Table 4). We established this characteristic to allow the analysis of *SOP* with test cases defined and created based on different strategies. We compared the *test profile* of S_1, S_2, S_3, and S_4 software to the operational profile of the respective software to verify the mismatch between the *SOP* and the tested software parts. In the following section, we describe the data analysis and the results of the data obtained from these comparisons.

5.3.1 *EXS–AT*$_2$: Data Analysis

We compared the *test profile* of S_1, S_2, S_3, and S_4 to the operating profiles of the respective software in an attempt to find the possible mismatch between *SOP* and the *test profile*. As we already described, in the context of this paper, *SOP* is determined by the frequency of methods execution. We classified the methods implemented in S_1, S_2, S_3, and S_4 based on their processing in *SOP* and the *test profile*. Thus, four classification categories are possible:

- Category 0: method not executed in *SOP* and not executed by the *test profile*;
- Category 1: method executed in *SOP* (by at least 1 participant) but not executed in the *test profile*;
- Category 2: method not executed in *SOP* but executed by the *test profile*;
- Category 3: method executed in both operational and *test profiles*.

As an example, in Figure 9 we show a fraction of the classification table of the methods implemented in S_1. In this example, the *test profile* (0) is compared to the operational profiles of participants 0, 1 and 2. We also classified the methods implemented in S_2, S_3 and S_4, generating a classification table for each software. The complete tables are available at `http://lcvm.com.br/artigos/anexos/jserd2020/tabelas/`.

In Figure 9 we show the classification table of S_1' methods. For each method, we assigned a classification category resulting from the comparison between *SOP* and the *test profile* of S_1. The columns "Participant OP Id./Test Profile", "CL" and "FREQ" refer, respectively, to:

a) the operational profile obtained by participants compared to the *test profile*. The line below column title informs the compared participant and the *test profile*;
b) the classification category assigned to the method;
c) the difference between the execution frequencies obtained in the operational profile of the participant and the *test profile*.

In Figure 10 we show the results from the comparison between *SOP* and the *test profile* for each evaluated software (S_1, S_2, S_3, and S_4).

For each evaluated software (S_1, S_2, S_3 and S_4) shown in Figure 10, the following information is provided:

- $OP \cap TP$: Number of methods processed by at least 1 participant and processed by the *test profile*.
- $OP \not\subset TP$: Number of methods processed by at least 1 participant and not processed by the *test profile*.
- $TP \not\subset OP$: Number of methods processed by the *test profile* and not processed by the participants.

The results show that:

a) 131 out of 280 methods from S_1 processed by at least 1 of the participants were not processed by the *test profile*; 30 methods processed by the *test profile* were not processed by the participants;
b) 313 out of 1308 methods from S_2 processed by at least 1 of the participants were not processed by the *test profile*; 1340 methods processed by the *test profile* were not processed by the participants;
c) 203 out of 437 methods from S_3 processed by at least 1 of the participants were not processed by the *test profile*; 134 methods processed by the *test profile* were not processed by the participants.
d) 4743 out of 8910 methods from S_4 processed by at least 1 of the participants were not processed by the *test profile*; 1319 methods processed by the *test profile* were not processed by the participants.

5.3.2 *EXS–AT*$_2$: Results

For the S_1, S_3 and S_4 software, approximately 50% of the methods processed by *SOP* were not processed by the *test profile*. The S_2's methods processed by *SOP* and not processed by the *test profile* correspond to approximately 25%. It is also possible to verify the occurrence of methods processed by the *test profile* and not processed by *SOP* for S_1, S_2, S_3 and S_4. For S_2, the number of methods processed by the *test profile* and not processed by *SOP* corresponds to approximately 30%. The results show a mismatch between *SOP* and the *test profile* for S_1, S_2, S_3 and S_4. According to Rincon (2011), only one open-source software among the ten open-source software researched by him obtained a coverage code between 70 and 80%. If we considered this interval acceptable, in the best case, we are delivering the software with 20% to 30% of the source code not having been executed during the testing phase. According to Ivanković et al. (2019), the median code coverage for all Google projects with successful coverage computation in the period between 2015 and 2018 varied between 80 and 85%, i.e., an interval between 15 and 20% of the uncovered code. Thus, even if we consider acceptable a percentage range for the misalignment between the *SOP* and the *test profile* that equals the range of uncovered code shown by Rincon (2011) and Ivanković et al. (2019), i.e., between 15 and 30%, the results obtained from *EXS–AT*$_2$ for S_1, S_3 and S_4 are greater than that considered an acceptable range when the methods processed by *SOP* and not processed by the *test profile*. For S_2, the obtained result is equal to the acceptable range considered when it comes to the methods processed in the test profile and not processed by *SOP*. These results show that there may be a misalignment between the *SOP* and tested software parts, providing an answer to question RQ_3.

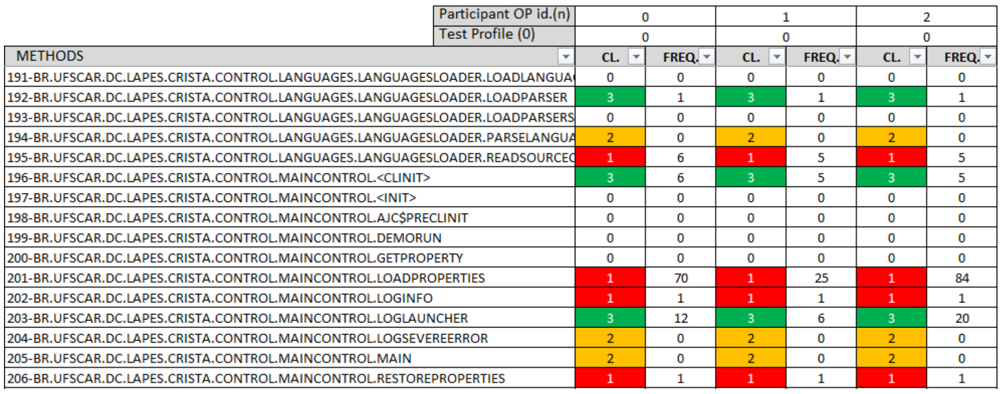
5.4 *EXS–AT*$_3$: Failures in untested *SOP* parts

Bach et al. (2017) investigated the relationship between the coverage provided by a test suite and its effectiveness. The approach adopted in Bach et al. (2017) can also be used as another strategy to get evidence of the possible mismatch between *SOP* and the tested software parts, as well as the relation between this misalignment and software faults. The approach used in Bach et al. (2017) defines two scenarios referring to the hypothesis investigated:

1. Coverage does not influence the detection of future bugs;
2. A high coverage rate can reduce the volume of future bugs.

Bach et al. (2017) analyzed identified faults using the failures reported by software users and the relation of the data obtained by this analysis to the coverage provided by the test suite of the respective software.

In the context of this paper, we assumed that the failures reported by software users occurred in software parts that constitute the *SOP* since such failures occur during the operation of the software by users. As such, the modified software parts resulting from fault corrections constitute the *SOP* and denote the occurrence of failures in the software parts that



	Participant OP id.(n)	0		1		2	
	Test Profile (0)	0		0		0	
METHODS		CL.	FREQ.	CL.	FREQ.	CL.	FREQ.
191-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.LANGUAGES.LANGUAGESLOADER.LOADLANGUA		0	0	0	0	0	0
192-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.LANGUAGES.LANGUAGESLOADER.LOADPARSER		3	1	3	1	3	1
193-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.LANGUAGES.LANGUAGESLOADER.LOADPARSERS		0	0	0	0	0	0
194-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.LANGUAGES.LANGUAGESLOADER.PARSELANGUA		2	0	2	0	2	0
195-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.LANGUAGES.LANGUAGESLOADER.READSOURCEC		1	6	1	5	1	5
196-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.<CLINIT>		3	6	3	5	3	5
197-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.<INIT>		0	0	0	0	0	0
198-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.AJC$PRECLINIT		0	0	0	0	0	0
199-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.DEMORUN		0	0	0	0	0	0
200-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.GETPROPERTY		0	0	0	0	0	0
201-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.LOADPROPERTIES		1	70	1	25	1	84
202-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.LOGINFO		1	1	1	1	1	1
203-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.LOGLAUNCHER		3	12	3	6	3	20
204-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.LOGSEVEREERROR		2	0	2	0	2	0
205-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.MAIN		2	0	2	0	2	0
206-BR.UFSCAR.DC.LAPES.CRISTA.CONTROL.MAINCONTROL.RESTOREPROPERTIES		1	1	1	1	1	1

Figure 9. Classification of software S_1 methods.

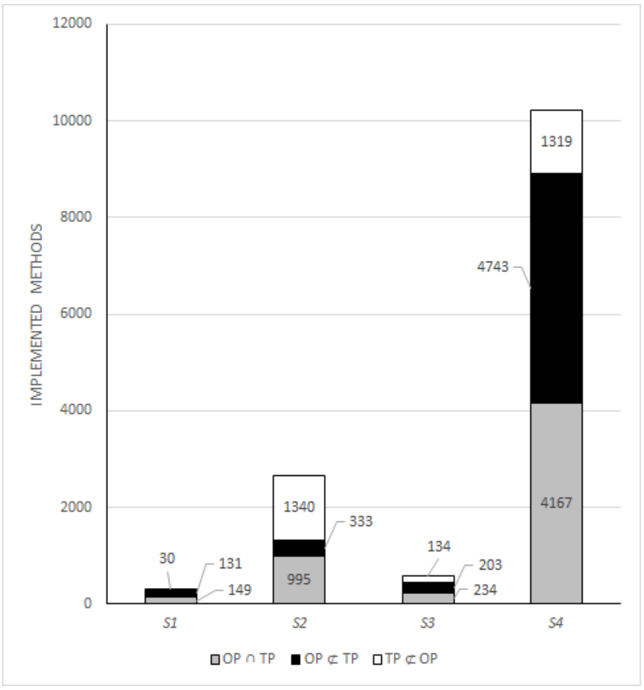


Figure 10. Results.

comprise the operational profile. Given these considerations, the activity $EXS\text{–}AT_3$ serves to verify:

1. If the misalignment between *SOP* and the tested software parts is relevant to software quality (faults not processed by the *test profile* do occur in *SOP* parts);
2. Although there is a misalignment between *SOP* and the tested software parts, this misalignment is irrelevant to software quality (no faults were registered in *SOP* parts not executed by the *test profile*).

We verified the fault history of S_2 and S_4. S_2 and S_4 are open-source software, and their source code is available on a hosting platform providing resources to manage modifications in the source code.

5.4.1 Analyzing failures in the untested SOP of S_2

By means of a pull request, we verified the changes in the S_2's source code classified as bug fix. This verification allowed us to identify the S_2's methods modified for attending a bug fix. We identified 79 methods that have corrections of faults identified by failures reported by users. As we assumed, these methods compose the *SOP* identified through data provided by the software community (bug reports), named SOP_{sup} in this section. We compared the methods comprising SOP_{sup} to the methods processed by S_2's *test profile*, identified in $EXS\text{–}AT_2$. We found that the *test profile* did not execute 49 out of the 79 methods constituting the SOP_{sup}, i.e., SOP_{sup} parts not covered by the test suite where we identified faults.

SOP_{sup} is based on the assumption that the methods corrected due to failures reported by the community constitute the *SOP*. Thus, these failures were not generated by the sporadic actions of users. Based on this assumption, we verified if the SOP_{sup} methods not processed by the *test profile* were contained in the *SOP* obtained by $EXS\text{–}AT_2$ participants. Among these methods, 7 methods were found in the *SOP* obtained by $EXS\text{–}AT_2$ participants. These 7 methods were classified as *SOP* methods not processed by the *test profile*. This indicates that, possibly, if the approach used in the activity is applied to the *SOP* obtained from the real users in a real scenario, the 7 methods contended in SOP_{sup}, i.e., methods presenting defects, would be found and classified as methods in *SOP* and continue untested. Thus, the approach applied in $EXS\text{–}AT_2$ improves new releases of the test suite since it identifies untested and faulty parts of the *SOP*.

5.4.2 Analyzing failures in the untested *SOP* parts of S_4

Unlike the procedure adopted to identify the SOP_{sup} of S_2, we obtained the SOP_{sup}'s methods of S_4 from a bug report available in its official website. For the bugs reported an error log was associated.By utilizing these error logs we could identify 15 methods that revealed failures during their execution. These methods comprise the SOP_{sup} of the S_4.

As with S_2, we compared the SOP_{sup} of S_4 to their *test profile* identified in $EXS\text{–}AT_2$. We found that the *test profile* did not execute 5 out of the 15 methods constituting the SOP_{sup}, i.e., SOP_{sup} parts not covered by the test suite where faults were identified.

5.4.3 *EXS–AT*$_3$: Results

Table 7 summarizes the data obtained from S_2 and S_4 about existing failures in untested *SOP* parts.

The investigation performed in $EXS\text{–}AT_2$ provided evidence of a mismatch between *SOP* and the tested software parts, and that failures occur in *SOP* parts left untested. For S_2, 62.02% of the SOP_{sup} parts in which faults identified

Table 7. S_2 and S_4 SOP_{sup} parts in which faults were identified.

SOP_{sup} parts in which faults were identified		
Software	Identified methods with faults	Identified methods not covered by test
S_2	79	49
S_4	15	5

were not covered by the *test profile*. For S_4 the respective value was 33.33%. This evidence answers research question RQ_4, showing that failures may occur in *SOP* parts not covered by the *test profile*.

5.5 *EXS–AT*$_4$: Attempting to decrease the misalignment between the *SOP* and the Test Profile

We performed the $EXS–AT_4$ activity to assess whether a test strategy based on the use of automated test data generator can contribute to reduce the possible misalignment between *SOP* and untested software parts.

To perform the $EXS–AT_4$ activity, we selected S_2 and S_4 software. The reasons why we selected these software are because we used them in $EXS–AT_2$ and $EXS–AT_3$ and because they are more representative regarding the number of implemented methods.

For each selected software, we generated a test set using an automated tool, named in this section as S_2TC_{tool} and S_4TC_{tool} for S_2 and S_4 software, respectively. The sets of existing test cases for S_2 and S_4 are named in this section as S_2TC_{exis} and S_4TC_{exis} (Table 4). We used *EvoSuite*, an automated generation tool, to write *JUnit* tests for Java software (Fraser and Arcuri, 2011). For the generation of S_2TC_{tool} and S_4TC_{tool}, among the coverage criteria made available by the test generation tool, we adopted the coverage criterion method, given that *SOP* is represented by the execution frequency of the implemented methods in this paper. For S_2 and S_4 were generated 4322 and 2803 test cases respectively. We did not use *SOP* data in the planning and execution of $EXS–AT_4$ test strategy, considering that the *SOP* was unknown for the generation of S_2TC_{tool} and S_4TC_{tool}. Then, we generated automated test cases for all S_2 and S_4 parts.

We incorporated the S_2TC_{tool} and S_4TC_{tool} test cases into S_2TC_{exis} and S_4TC_{exis} respectively, thus obtaining an extended test set resulted for S_2 and S_4 from the union of these sets. We named the extended test sets of S_2 and S_4 as S_2TC_{ext} and S_4TC_{ext}, respectively, in this section. In Table 8 we show the coverage for S_2 and S_4 provided by each set of test cases. The numeric values in percentage are presented in Table 8.

Table 8. S_2 and S_4 software coverage provided by test cases.

Coverage provided by test cases			
Software	TC_{exis}	TC_{tool}	TC_{ext}
S_2	15%	27%	30%
S_4	32%	42%	60%

In Table 8 we show that the S_2TC_{ext} and S_4TC_{ext} test cases increased the coverage of S_2 and S_4 provided by S_2TC_{exis} and S_4TC_{exis} respectively, showing that new parts of S_2 and S_4 were tested and, consequently, extending the S_2 and S_4 *test profiles*. We named the initial *test profiles* obtained from S_2TC_{exis} and S_4TC_{exis} as S_2TP_{ini} and S_4TP_{ini} in this section. Also, we named the extended *test profiles* of S_2 and S_4 in this section as S_2TP_{ext} and S_4TP_{ext}, respectively.

We adopted the same procedure to identify the S_2TP_{ini} and S_4TP_{ini}, described in Section 5.3, to obtain S_2TP_{ext} and S_4TP_{ext}.

The same procedure used to compare the S_2TP_{ini} and S_4TP_{ini} to the S_2's *SOP* and S_4's *SOP* respectively was used to compare the S_2TP_{ext} and S_4TP_{ext} to the S_2's *SOP* and S_4's *SOP* respectively.

5.5.1 *EXS–AT*$_4$: Data Analysis

In Figures 11 and 12 we show, for S_2 and S_4 respectively, the results obtained from the comparison between the *SOP* and the extended *test profile*. Results obtained by comparing the *SOP* of these software to the initial *test profiles* (S_2TP_{ini} and S_4TP_{ini}) are presented again in Figures 11 and 12 to compare them with the results obtained from the S_2TP_{ext} and S_4TP_{ext}.

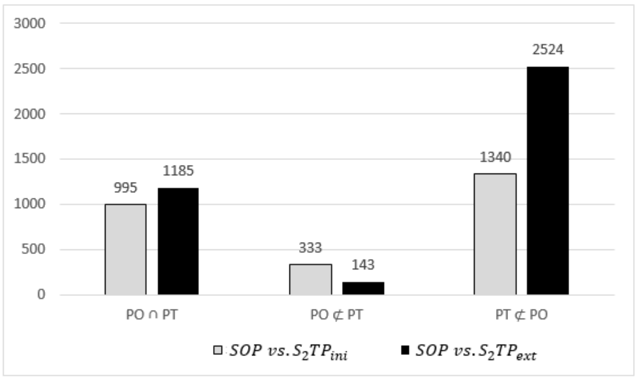


Figure 11. S_2TP_{ini} and S_2TP_{ext} results.

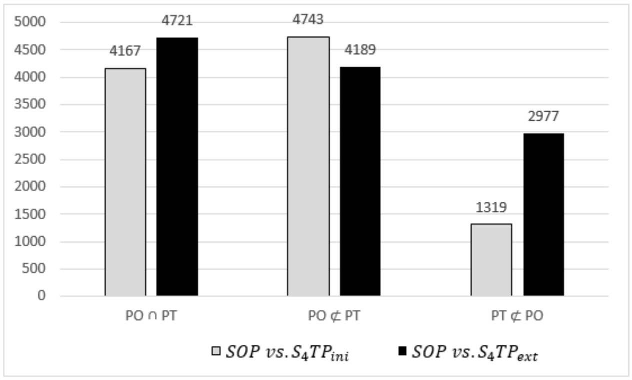


Figure 12. S_4TP_{ini} and S_4TP_{ext} results.

We defined the categories $OP \cap TP$, $OP \not\subset TP$ and $TP \not\subset OP$, shown in Figures 11 and 12, in Section 5.3.1

In Figures 11 and 12, we can see that:

1. 143 out of 1328 methods from S_2 processed by at least 1 of the participants were not processed by the TP_{ext};

2524 methods processed by the *test profile* were not processed by the participants.

2. 4189 out of 8910 methods from S_4 processed by at least 1 of the participants were not processed by the TP_{ext}; 2977 methods processed by the *test profile* were not processed by the participants.

5.5.2 *EXS–AT*$_4$: Results

In Table 9 we show the difference resulted from TP_{ini} and TP_{ext}.

After comparing the results obtained by S_2TP_{ini} and S_4TP_{ini}, the test strategy we adopted in activity *EXS–AT*$_4$ reduced the number of methods processed by *SOP* and not processed by the *test profile* (S_2TP_{ext} and S_4TP_{ext}), being more effective for the S_2 software. However, it is noteworthy that, regarding the number of implemented methods, S_2 is less representative than S_4, for which the adopted strategy reduced the amount of methods processed by *SOP* and not processed by the *test profile* (S_4TP_{ext}) in, approximately, 10% compared to the initial *test profile* (S_4TP_{ini}).

The adopted test strategy also reduced the number of methods constituting the SOP_{sup} of S_2 and S_4 and were not covered by the respective *test profile*, S_2TP_{ini} and S_4TP_{ini}. For S_2, 2 out of 49 methods constituting the SOP_{sup} and were not processed by S_2TP_{ini} were processed by S_2TP_{ext}. For S_4, 1 out of 5 methods constituting the SOP_{sup} and were not processed by S_4TP_{ini} was processed by S_4TP_{ext}.

The adopted test strategy aimed to reduce the misalignment between *SOP* and *Test Profile* by increasing the set of existing test cases of S_2 and S_4 using an automated tool. We did not use *SOP* data in the test strategy planning and execution, considering that the *SOP* was unknown for the automatic generation of test cases, which implied generating test cases for all parts of S_2 and S_4, demanding time and processing because they depend on the applied criteria and parameters as well as on the size of the software for which the test cases were generated.

In response to question *RQ*$_5$, we observed that, although we generated test cases for all parts of S_2 and S_4 and incorporated these cases into the set of existing test cases for the software, the test strategy reduced the misalignment, but the misalignment between *SOP* and the test profile of S_2 and S_4 was unavoidable. In addition, the automated test generator generates only the test data and assumes the produced output is correct. As such, even if we have improved the coverage of *SOP*, we still need to verify whether the resultant output corresponds to the expected output according to the software specification. Thus, the data obtained from the *SOP* is relevant and can be used in existing testing strategies or in the definition of new strategies to contribute to their effectiveness and efficiency.

6 Lessons Learned

First of all, we would like to make it clear that the results obtained so far are not conclusive and they are part of an ongoing work Cavamura Júnior (2017), and more experimental studies are coming. However, based on the data presented in Section 5, we can provide some directions (albeit not exhaustive) on how to use the knowledge about *SOP* in favor of software quality.

- We verified during the experimental studies that the identification of *SOP* through instrumentation may affect software performance and produce a huge volume of data depending on the level of fragmentation adopted. Nevertheless, the information obtained about the *SOP* can contribute to software test activities.
- High levels of coverage do not necessarily indicate a test set is effective in detecting faults and it is unlikely that the use of a fixed value of coverage as a quality target will produce an effective test set (Inozemtseva and Holmes, 2014). Our data indicates that a good test set is one with good coverage of the software parts related to the *SOP*. In the occurrence of misalignment between the *SOP* and the tested software parts, the *SOP* can also be used as a criterion for generating test cases to improve the test suite in order to minimize the misalignment.
- Another possible use of the *SOP* is related to what de Andrade Freitas et al. (2016) called as "Market Vulnerability", wherein each fault in software affects users differently. We should avoid bothering most of our users with constant failures as much as possible when using features most important from their point of view. The *SOP* reflects these software areas. It is possible to use *SOP* to assess the impact caused by each fault in software operability. Thus, a rank of known faults can be built based on their impact to the majority of users, providing information able to assist in precifying these faults with respect to the software market.

- Since the *SOP* represents the most used parts of the software, information about the *SOP* can be used as a criterion to prioritize any other activities inherent to the software development process.

7 Threats to Validity

Regarding the *EXS* activities, we considered the participants' level of knowledge in *EXS* a threat to validity. We selected undergraduate and postgraduate students, who had equivalent experience and knowledge required to perform the activity, to operate S_1, S_2 and S_4 software in order to minimize the risks. We conducted training on S_1, S_2 and S_4, as well as a review of the theoretical concepts inherent in S_1, S_2 and S_4. As S_3 was developed on demand, participants already knew the processes automated by it.

On *EXS-AT*$_2$ the execution of some test cases belonging to the test sets of S_1, S_2 and S_4 run with errors. For S_1 0.69% of the automatically generated test cases finished the execution with errors. For S_2 1.36% of the automatically generated test cases finished the execution with errors. For S_4 17.10% of the automatically generated test cases finished the execution with errors.

With the configuration and execution environment in conformity, we chose not to modify the implementation of the existing test cases in order to eliminate the execution errors.

Table 9. Comparison of the results obtained by the test profiles.

TC_{comm}	S2			S4		
–	*SOP* vs $\text{S}_2\text{TP}_{ini}$	*SOP* vs $\text{S}_2\text{TP}_{ext}$	(%)	*SOP* vs $\text{S}_4\text{TP}_{ini}$	*SOP* vs $\text{S}_4\text{TP}_{ext}$	(%)
$OP \cap TP$	995	1185	19.09 (+)	4167	4721	13.29 (+)
$OP \not\subset TP$	333	143	57.05 (-)	4743	4189	11.68 (-)
$TP \not\subset OP$	1340	2524	88.35 (+)	1319	2977	125.7 (+)

We considered these a threat to validity because some methods may have been executed as a result of these errors, thus not being part of the test profile.

On *EXS–AT*$_3$ activity, we assumed failures reported by users were revealed by the software parts composing *SOP*, i.e., these failures did not occur in operations sporadically processed by users. We are performing a more comprehensive *EXS* using data obtained from free software repositories.

On *EXS–AT*$_4$ the execution of some test cases automatically generated for S_2 (S_2TC_{tool}) and S_4 (S_4TC_{tool}) rendered errors. For S_2, 4.2% of the automatically generated test cases generated errors during their execution. For S_4 0.53% of the automatically generated test cases generated errors. Although these errors have low representativeness, they are considered a threat to validity since some methods may have been executed as a result of these errors, thus not being part of the extended test profiles of S_2TP_{ext} and S_4TP_{ext}, respectively.

In further experiments we intend to investigate the cause of such errors and compute their impact on the test profile.

8 Conclusions

This paper investigates the possible mismatch between *SOP* and the tested software parts by introducing the term "test profile". The results provided answers to the defined research questions, stating: a) the originality of this study; b) that there are significant variations in the way software is used by users; c) there may exist a misalignment between the *SOP* and the test profile; d) the existing misalignment is relevant due to the evidence that failures occur in the untested *SOP* parts; e) Although the adopted test strategy reduced the misalignment between the SOP and test profile, it was not enough to avoid the misalignment.

The answers to the research questions provide the expected contributions to this work. These contributions may motivate new research or contribute to existing research in Software Engineering, more specifically in the field of Software Quality. The contributions also show that information about software operating profiles can contribute to the software quality activities applied in the industry since the quality of software also depends on its operational use (Cukic and Bastani, 1996).

Thus, the contributions provide evidence that *SOP* is relevant not only to activities that determine software reliability but also to the planning and execution of the test activity regardless of the adopted test strategy. For future research we intend to improve software quality from the users' point of view considering the *SOP* (Cavamura Júnior, 2019).

We expect that the proposed strategy allows: (i) to dynamically adapt an existing test suite to the *SOP*, and; (ii) use *SOP* as a prioritization criterion which, given a set of faults, allows to identify the ones that cause the most significant impact on users' experience when operating the software, and thus consider such impact on pricing the faults for correction, alongside other criteria. We are investigating and approaching the use of machine learning and genetic algorithms to enable the proposed strategy. Lastly, we are working on the implementation of a tool to automate the proposed strategy and to provide support for technology transfer and experimentation.

References

Ali-Shahid, M. M. and Sulaiman, S. (2015). Improving reliability using software operational profile and testing profile. In 2015 International Conference on Computer, Communications, and Control Technology (I4CT), pages 384–388. IEEE Press.

Amrita and Yadav, D. K. (2015). A novel method for allocating software test cases. In 3rd International Conference on Recent Trends in Computing 2015 (ICRTC-2015), volume 57, pages 131 – 138, Delhi, India. Elsevier.

Assesc, F. (2012). Propriedade intelectual e software - cursos de mídia eletrônica e sistema de informação.

Bach, T., Andrzejak, A., Pannemans, R., and Lo, D. (2017). The impact of coverage on bug density in a large industrial software project. In 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 307–313.

Basili, V. R., Caldiera, G., and Rombach, D. H. (2002). Encyclopedia of Software Engineering, volume 1, chapter The Goal Question Metric Approach, pages 528–532. John Wiley Sons.

Begel, A. and Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. ICSE 2014, pages 12–23.

Bertolino, A., Miranda, B., Pietrantuono, R., and Russo, S. (2017). Adaptive coverage and operational profile-based testing for reliability improvement. In Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pages 541–551, Piscataway, NJ, USA. IEEE Press.

Bittanti, S., Bolzern, P., and Scattolini, R. (1988). An introduction to software reliability modelling, chapter 12, pages 43–67. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cavamura Júnior, L. (2017). Impact of The Use of Operational Profile on Software Engineering Activities. Phd thesis, Computing Department – Federal University of São Carlos, São Carlos, SP, Brazil. On going PhD Project (in Portuguese).

Cavamura Júnior, L. (2019). Operational profile and software testing: Aligning user interest and test strategy. In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pages 492–494.

Cavamura Júnior, L., Fabbri, S. C. P. F., and Vincenzi, A. M. R. (2020). Software operational profile: investigating specific applicabilities. In Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE'2020, Curitiba, PR, Brazil. Curran Associates. Accepted for publication. (in Portuguese).

Chen, M. ., Lyu, M. R., and Wong, W. E. (2001). Effect of code coverage on software reliability measurement. IEEE Transactions on Reliability, 50(2):165–170.

Cukic, B. and Bastani, F. B. (1996). On reducing the sensitivity of software reliability to variations in the operational profile. In Proceedings of the International Symposium on Software Reliability Engineering, ISSRE, pages 45–54, White Plains, NY, USA. IEEE, Los Alamitos, CA, United States.

de Andrade Freitas, E. N., Camilo-Junior, C. G., and Vincenzi, A. M. R. (2016). SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing. In XXVII IEEE International Symposium on Software Reliability Engineering – ISSRE'2016, pages 36–46. IEEE Press. bibtex*[organization=IEEE Computer Society] event-place: Ottawa, Canadá.

Falbo, R. A. (2005). Engenharia de software.

Ferrari, F. C., Rashid, A., and Maldonado, J. C. (2013). Towards the practical mutation testing of aspectj programs. Science of Computer Programming, 78(9):1639 – 1662.

Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 416–419, New York, NY, USA. ACM.

Fukutake, H., Xu, L., Takagi, T., Watanabe, R., and Yaegashi, R. (2015). The method to create test suite based on operational profiles for combination test of status. In 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2015 - Proceedings, pages 1–4, White Plains, NY, USA. Institute of Electrical and Electronics Engineers Inc.

Gittens, M., Lutfiyya, H., and Bauer, M. (2004). An extended operational profile model. In Proceedings - International Symposium on Software Reliability Engineering, ISSRE, pages 314 – 325, Saint-Malo, France.

Inozemtseva, L. and Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, page 435–445, New York, NY, USA. Association for Computing Machinery.

Ivanković, M., Petrović, G., Just, R., and Fraser, G. (2019). Code coverage at google. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, page 955–963, New York, NY, USA. Association for Computing Machinery.

Kashyap, A. (2013). A Markov Chain and Likelihood-Based Model Approach for Automated Test Case Generation, Validation and Prioritization: Theory and Application. Proquest dissertations and theses, The George Washington University.

Laddad, R. (2009). AspectJ in Action: Enterprise AOP with Spring Applications. Manning Publications Co., Greenwich, CT, USA, 2nd edition.

Leung, Y.-W. (1997). Software reliability allocation under an uncertain operational profile. Journal of the Operational Research Society, 48(4):401 – 411.

Linger, R. C., Mills, H. D., and Witt, B. I. (1979). Structured programming - theory and practice. In The systems programming series.

Mafra, S. N., Barcelos, R. F., and Travassos, G. (2006). Applying an evidence based methodology to define new software technologies. In XX Brazilian Symposium on Software Engineering - SBES'2006, pages 239–254, Florianópolis, SC, Brazil. Available at: `http://www.ic.uff.br/~esteban/files/sbes-prova.pdf`. Access on: 05/04/2020. (in Portuguese).

Musa, J. (1993). Operational profiles in software-reliability engineering. IEEE Software, 10(2):14–32. cited By 396.

Musa, J. and Ehrlich, W. (1996). Advances in software reliability engineering. Advances in Computers, 42(C):77–117. cited By 1.

Musa, J. D. (1979). Software reliability measures applied to systems engineering. In Managing Requirements Knowledge, International Workshop on(AFIPS), volume 00, page 941, S.I. IEEE.

Musa, J. D. (1994). Adjusting measured field failure intensity for operational profile variation. In Proceedings of the International Symposium on Software Reliability Engineering, ISSRE, pages 330–333, Monterey, CA, USA. IEEE, Los Alamitos, CA, United States.

Nakagawa, E. Y., Scannavino, K. R. F., Fabbri, S. C. P. F., and Ferrari, F. C. (2017). Revisão Sistemática da Literatura em Engenharia de Software: Teoria e Prática. Elsevier Brasil.

Namba, Y., Akimoto, S., and Takagi, T. (2015). Overview of graphical operational profiles for generating test cases of gui software. In K., S., editor, 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2015 - Proceedings, pages 1–3, White Plains, NY, USA. Institute of Electrical and Electronics Engineers Inc.

Poore, J., Walton, G., and Whittaker, J. (2000). A constraint-based approach to the representation of software usage models. Information and Software Technology, 42(12):825 – 833.

Pressman, R. S. (2010). Software Engineering A Practitioner's Approach. McGraw-Hill, New York, NY, 7rd edition.

Rincon, A. M. (2011). Qualidade de conjuntos de teste de software de código aberto, uma análise baseada em critérios estruturais.

Rocha, A. D. (2005). Uma ferramenta baseada em aspectos para apoio ao teste funcional de programas java.

Shukla, R. (2009). Deriving parameter characteristics. In Proceedings of the 2nd India Software Engineering Conference, ISEC 2009, pages 57–63, New York, NY, USA. ACM.

Shull, F., Carver, J., and Travassos, G. H. (2001). An empirical methodology for introducing software processes. In Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9, page 288–296, New York, NY, USA. Association for Computing Machinery.

Sommerville, I. (1995). Software Engineering. Addison-Wesley, Wokingham, England, fifth edition edition.

Sommerville, I. (2011). Software Engineering. Addison-Wesley, Harlow, England, 9 edition.

Takagi, T., Furukawa, Z., and Yamasaki, T. (2007). An overview and case study of a statistical regression testing method for software maintenance. Electronics and Communications in Japan Part II Electronics, 90(12):23–34.

Travassos, G. H., dos Santos, P. S. M., Mian, P. G., Neto, P. G. M., and Biolchini, J. (2008). An environment to support large scale experimentation in software engineering. In 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008), pages 193–202.