Universidade Federal de São Carlos– UFSCar
Centro de Ciências Exatas e de Tecnologia– CCET
Departamento de Computação– DC
Programa de Pós-Graduação em em Ciência da Computação– PPGCC

Luiz Cavamura Júnior

Combinando perfil operacional do software e perfil de teste para uma estratégia de teste aderente às necessidades dos usuários

São Carlos
2022

Luiz Cavamura Júnior

Combinando perfil operacional do software e perfil de teste para uma estratégia de teste aderente às necessidades dos usuários

Tese apresentada ao Programa de Pós-Graduação em em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Metodologias e Técnicas de Computação

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi
Coorientador: Profa. Dra. Sandra Camargo Pinto Ferraz Fabbri

São Carlos
2022

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Luiz Cavamura Júnior, realizada em 28/01/2022.

Comissão Julgadora:

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Prof. Dr. Eduardo Noronha de Andrade Freitas (IFG)

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Daniel Lucrédio (UFSCar)

Prof. Dr. Jacson Rodrigues Barbosa (UFG)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.
O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Este trabalho é dedicado à minha família. À minha esposa e filho pela compreensão, auxílio e incentivo. Aos meus pais.

Agradecimentos

Agradeço...

...a Deus, sempre, por tudo.

...a toda minha família, em especial a minha esposa e filho, pela compreensão, auxílio e incentivo durante o doutoramento.

...ao Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP) pelo incentivo.

...a todos que, de alguma maneira, em algum momento, participaram dessa caminhada comigo ao longo do doutoramento.

...e, claro, especialmente...

...ao Prof. Dr. Auri Marcelo Rizzo Vincenzi e a Profa. Dra. Sandra Camargo Pinto Ferraz Fabbri pela oportunidade, confiança, paciência e orientação. Sem vocês, este trabalho não existiria. Obrigado.

Enfim, Professor Doutor...

"Somente se aproxima da perfeição quem a procura com constância, sabedoria e, sobretudo humildade."
(Jigoro Kano)
"Um professor sempre afeta a eternidade. Ele nunca saberá onde sua influência termina.
(Henry Brooks Adams)

Resumo

O Perfil Operacional do Software (*POS*) é uma representação da maneira como os usuários usam o software na prática, possibilitando identificar as partes do software mais operadas pelos usuários. Sendo a confiabilidade de um software dependente do contexto no qual ele é usado, o *POS* é empregado na engenharia de confiabilidade de software. Contudo, há indícios de um possível descompasso entre as partes testadas do software e o *POS*. Este trabalho investiga o possível descompasso entre as partes testadas do software e o *POS* e, também, como dados extraídos do *POS* podem prover contribuições a outras estratégias de teste não relacionadas ao teste de confiabilidade. Estudos experimentais foram realizados para obter dados que permitissem atingir e materializar os objetivos por meio de publicações ao longo do período de doutoramento. As principais contribuições desta pesquisa, considerando a sequência de publicações que as corroboram são: *i)*Evidências que comprovam que há variações significativas na maneira como os usuários operam o software mesmo quando realizam uma mesma operação, que é possível a existência de um descompasso entre as partes testadas do software e o *POS* e que falhas podem ocorrer nas partes do *POS* não testadas; *ii)* Evidências de que os geradores automáticos de teste considerados estado da arte para essa tarefa contribuem para a redução do descompasso mas não o elimina; *iii)* Apresentação do conceito de "Perfil de Teste"; *iv)* Projeto e implementação da Ferramenta *OPDdTe* cujo propósito é contribuir para diminuir o descompasso entre o *POS* e o *perfil de teste* por meio da geração automática de casos de teste executáveis com dados de teste obtidos dinamicamente do *POS*; *v)* Realização de um mapeamento sistemático e uma revisão sistemática da literatura que investigam o uso do *POS*. Os resultados obtidos pela pesquisa evidenciaram a relevância do *POS* ao teste de software, possibilitando alinhar as estratégias de teste ao uso operacional do software e, assim, estar em consonância com as necessidades dos usuários.

Palavras-chave: Perfil Operacional, Perfil de Teste, *OPDdTe* Teste de Software.

Abstract

The Software's Operational Profile (*SOP*) is a representation of how users use the software in practice, thus identifying the parts of the software most operated by the users. Since the reliability of a software depends on the context in which it is used, the *SOP* is employed in software reliability engineering. However, there are signs of a possible mismatch between the software's tested parts and the *SOP*. The objective of this work is to investigate the possible mismatch between the tested parts of the software and the *SOP* and how data extracted from the *SOP* can provide contributions to other test strategies not related to reliability testing. We performed experimental studies to obtain data that would allow achieving and materializing the objectives through publications throughout the doctoral period, providing the contributions of this research. The main contributions of this research are: *i)*Evidence that proves that there are significant variations in the way users operate the software even when performing the same operation that it is possible that there is a mismatch between the tested parts of the software and the *SOP*, and that failures can occur in parts of the *SOP* not tested; *ii)*Evidence that a test strategy based on merging an existing test suite with a test suite generated by an automated test data tool can decrease the mismatch but does not prevent this mismatch; *iii)*Presentation of the "Test Profile" concept; *iv)*Design and implementation of *OPDdTe* tool whose purpose is to contribute to reducing the mismatch between the *SOP* and the test profile through the automatic generation of executable test cases with test data obtained dynamically from the *SOP*; *v)*Conducting a systematic mapping and a systematic review of the literature that investigates the use of the *SOP*. The results obtained by the research show the relevance of *SOP* to software testing, making it possible to align testing strategies with the operational use of the software and, thus, be in line with the users' needs.

Keywords: Operational Profile, Test Profile, *OPDdTe*, Software Test.

Lista de publicações

A seguir é apresentada a lista de publicações originadas durante o doutoramento até o presente momento.

❏ Artigo completo em conferência:

- CAVAMURA, L. J. Operational profile and software testing: Aligning user interest and test strategy. In: **Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST) – Doctoral Symposium**. [S.l.: s.n.], 2019. p. 492–494.
- CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. Perfil operacional do software: investigando aplicabilidades específicas. In: AYALA, C. P. et al. (Ed.). **Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020, Curitiba, Paraná, Brazil, November 9-13, 2020**. [S.l.]: Curran Associates, 2020. p. 292–305.

❏ Artigo completo em periódico:

- ______. Software operational profile vs. test profile: Towards a better software testing strategy. **Journal of Software Engineering Research and Development**, v. 8, p. 5:1 – 5:17, ago. 2020.

❏ Artigo em fase de submissão:

- ______. OPDaTe: A unit test case generator based on real data from software operating profile. ——, —, x, p. xxx–xxx(x), jan. 2022. ISSN -. Artigo submetido para avaliação.

Lista de ilustrações

Figura 1 – Publicações originadas durante o período de doutoramento 26
Figura 2 – Conceitos envolvidos na definição do perfil operacional 32
Figura 3 – Atribuição explícita e implícita 33
Figura 4 – Relacionamento entre o custo e melhorias relacionadas a confiabilidade. Obtido de (SOMMERVILLE, 2015) 38
Figura 5 – Gráfico de confiabilidade. Adaptado de (MUSA; EHRLICH, 1996) . . . 39

Lista de tabelas

Tabela 1 – Publicações durante o período de doutoramento 27
Tabela 2 – Objetivos e questões de pesquisa das publicações que compõem o corpo da tese 29
Tabela 3 – Problemas relacionados ao uso do *POS* 40

Lista de siglas

POS Perfil Operacional do Software

SOP Software's Operational Profile

Sumário

1 INTRODUÇÃO 23
1.1 Contexto 23
1.2 Motivação 24
1.3 Hipótese 25
1.4 Objetivos 25
1.5 Organização do texto 26
1.5.1 Relação entre o problema de pesquisa e as publicações 27
1.5.2 Considerações finais 28

2 PERFIL OPERACIONAL DO SOFTWARE E A QUALIDADE DE SOFTWARE 31
2.1 Perfil Operacional do Software 31
2.2 O Perfil Operacional do Software e a Qualidade de Software . . 34
2.2.1 Requisitos 37
2.2.2 Arquitetura 37
2.2.3 Teste 38
2.2.4 Operação 39
2.3 Dificuldades e Problemas relacionados ao uso do *POS* 40
2.4 Considerações Finais 40

3 PERFIL OPERACIONAL E TESTE DE SOFTWARE: ALINHANDO O INTERESSE DO USUÁRIO E A ESTRATÉGIA DE TESTE 41

4 PERFIL OPERACIONAL DO SOFTWARE: INVESTIGANDO APLICABILIDADES ESPECÍFICAS 45

5 PERFIL OPERACIONAL DO SOFTWARE VS. PERFIL DE TESTE: EM DIREÇÃO A UMA MELHOR ESTRATÉGIA DE TESTE DE SOFTWARE 61

6 OPDATE: UM GERADOR DE CASOS DE TESTE DE UNIDADE BASEADO EM DADOS REAIS OBTIDOS DO PERFIL OPERACIONAL DO SOFTWARE 81

7 CONCLUSÃO, TRABALHOS FUTUROS E LIÇÕES APRENDIDAS 103

7.1 Conclusão 103

7.2 Trabalhos Futuros 104

7.3 Lições aprendidas 105

REFERÊNCIAS 107

Capítulo 1

Introdução

Este capítulo apresenta a contextualização do tema abordado, as motivações para a realização da pesquisa, a hipótese de pesquisa e os objetivos deste trabalho. As publicações decorrentes da pesquisa, ocorridas ao longo do doutoramento, compõem o corpo desta teste. Assim, ao final deste capítulo, o corpo da tese é explicado.

1.1 Contexto

A concepção do software provém da necessidade de soluções eficientes para problemas da sociedade contemporânea, sendo empregado em processos das mais variadas áreas da sociedade. Como a tecnologia é evolutiva, e a melhoria contínua é uma das características da sociedade contemporânea, tais processos estão em constante atualização. O software, como elemento estratégico nesses processos, precisa ter a capacidade de assimilar novas maneiras de uso em decorrência das alterações nos processos para o qual foi criado. Essa característica faz com que as funcionalidades do software, mesmo que criadas para atender regras de negócio comuns a diversas organizações, possam também ser parametrizáveis para atender as necessidades específicas e particulares.

As diferentes maneiras de utilização do software podem, também, ser decorrentes do carácter criativo do intelecto humano (ASSESC, 2012). Esse mesmo carácter criativo proporciona, aos usuários do software, a capacidade de adaptar, com base em suas experiências profissionais, o comportamento do software (SOMMERVILLE, 2015), encontrando, assim, diferentes maneiras de utilização para o software. As diferentes maneiras de utilização do software faz com que partes do software tenham uma frequência de utilização e relevância distintas entre as organizações e seus usuários, atribuindo, ao software, diferentes perfis operacionais, denominado Perfil Operacional do Software (*POS*).

O *POS* corresponde a uma representação quantitativa do software baseada na maneira como o software é operado pelos usuários, a qual é composta pelas operações do software e uma distribuição probabilística associada a essas operações (MUSA, 1993; GITTENS; LUTFIYYA; BAUER, 2004; SOMMERVILLE, 2015).

Em decorrência dos diferentes perfis operacionais que podem ser atribuídos a um software, a ocorrência de uma falha durante a operação do software por um usuário pode não ser reproduzida por outros usuários, dado que usuários experientes conseguem adaptar a maneira com a qual o software é operado (SOMMERVILLE, 2015), tornando a qualidade do software dependente do seu uso operacional (HE et al., 2021; ZHAKIPBAYEV; BEKEY, 2021). Assim, como a confiabilidade de um software é determinada pela operação do software sem a ocorrência de falhas em um determinado tempo e em um ambiente específico, o *POS* é usado no teste de confiabilidade para de determinar a confiabilidade do software. A confiabilidade de um software corresponde à probabilidade de operação do software, por um determinado tempo, em um ambiente específico, sem a ocorrência de falhas (MUSA, 1979).

Como a qualidade do software está condicionada também a requisitos não-funcionais explícitos e implícitos, tais como usabilidade, segurança, confiabilidade, dentre outros (KOSCIANSKI, 2006; PRESSMAN, 2019; SOMMERVILLE, 2015), informações sobre a maneira como o software é operado pelos usuários, ou seja, sobre o *POS*, tornam-se relevantes para assegurar a qualidade do software.

1.2 Motivação

Uma pesquisa evidenciou que informações sobre o *POS* foram consideradas essenciais ou relevantes a questões relacionadas às atividades inerentes ao processo de desenvolvimento do software (BEGEL; ZIMMERMANN, 2014). São exemplos dessas questões: *"Quais partes do software são mais utilizadas?"*, *"Como os usuários usam a aplicação?"*, *"Quais são os padrões de utilização do software?"* e *"Como a cobertura dos testes correspondem ao código realmente executado pelos usuários?"*. Um outro estudo analisou dinamicamente 10 softwares de código gratuito/livre/aberto (*free/libre/open source – FLOSS*) e constatou que somente 1 software atingiu um percentual de cobertura, provida pelo conjunto de testes, entre 70 e 80 (RINCON, 2011). Mesmo que este intervalo seja considerado aceitável, há um percentual de código, não coberto pelo conjunto de testes, que pode estar relacionado às funcionalidades mais utilizadas pelos usuários, evidenciando um possível descompasso entre as partes testadas e as partes que os usuários efetivamente utilizam (RINCON, 2011). Assim, há indícios de um possível descompasso entre o *POS* e as partes testadas do software e, também, de que informações sobre *POS* são relevantes para assegurar a qualidade do software.

Esse descompasso pode tornar frequente a ocorrência de falhas aos usuários. A ocorrência de falhas durante a execução do software causa prejuízos às partes interessadas nesse software. Para os desenvolvedores do software, a ocorrência de falhas ocasiona custos decorrentes de investigação, retrabalho e insatisfação do cliente. Para a organização que investiu no software, a ocorrência de falhas ocasiona custos decorrentes da não utilização do software, que pode gerar ociosidade aos usuários e parada nos processos de negócio da organização, além da falta de confiabilidade no software. A frequência com que um defeito se torna aparente é mais significante que os defeitos ainda remanescentes no software, o que torna a qualidade do software dependente do seu uso operacional (HE et al., 2021; ZHAKIPBAYEV; BEKEY, 2021). A realização do teste de software sob uma estratégia exaustiva para cobrir todas as partes do software torna-se difícil e pode se tornar impraticável dada a complexidade do software (MYERS; SANDLER; BADGETT, 2012). O uso do *POS* não garante que todos os defeitos sejam detectados, mas garante que as partes mais usadas do software sejam testadas suficientemente (ALI-SHAHID; SULAIMAN, 2015).

1.3 Hipótese

Este trabalho defende a tese de que existe um descompasso entre as partes testadas do software e o *POS* e que informações sobre o *POS* provê contribuições a outras estratégias de teste não relacionadas ao teste de confiabilidade, de maneira a diminuir esse desalinhamento, almejando melhorar a qualidade do software do ponto de vista dos usuários.

1.4 Objetivos

Considerando que há indícios de um possível descompasso entre o *POS* e as partes testadas do software, o objetivo geral desta tese é evidenciar que existe um descompasso entre o *POS* e as partes testadas do software e que o *POS* pode prover contribuições a outras estratégias de teste não relacionadas ao teste de confiabilidade, almejando melhorar a qualidade do software do ponto de vista dos usuários.

Para atingir o objetivo geral, são objetivos específicos deste trabalho:

1. Constatar a originalidade da pesquisa e apresentá-la à comunidade científica para obter um *feedback* dos pesquisadores da área.

2. Evidenciar que há variações significativas na maneira como o software é operado pelos usuários.

3. Evidenciar o possível descompasso entre o *POS* e as partes testadas do software e que falhas podem ocorrer nas partes do *POS* não testadas.

4. Evidenciar que estratégias de teste baseadas no uso de ferramentas de geração de dados de teste automatizadas não eliminam o possível descompasso entre o *POS* e as partes testadas do software.

5. Definir e apresentar o termo "Perfil de Teste".

6. Visando diminuir o possível descompasso entre o *POS* e as partes testadas do software, projetar e implementar uma ferramenta computacional para geração automática de casos de teste de unidade executáveis cujos dados de teste são obtidos dinamicamente do *POS*.

1.5 Organização do texto

O Capítulo 2 apresenta a fundamentação teórica referente ao *POS*, a qual é abordada nas publicações que compõem o corpo desta tese. Os capítulos 3 a 6 correspondem às publicações que compõem o corpo desta tese. Essas publicações foram originadas durante o período de doutoramento e abordam o problema de pesquisa. O Capítulo 7 apresenta as conclusões, trabalhos futuros e as lições aprendidas.

A Figura 1 mostra uma *timeline* com as publicações originadas durante o período de doutoramento e apresentadas na lista de publicações deste trabalho.

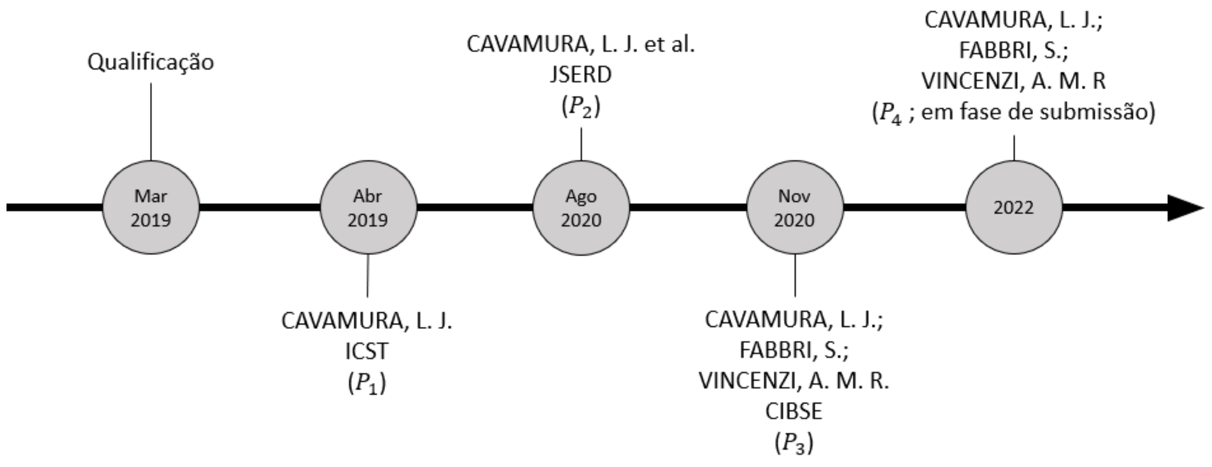

Qualificação
Mar 2019
Abr 2019
CAVAMURA, L. J. ICST (P_1)
CAVAMURA, L. J. et al. JSERD (P_2)
Ago 2020
Nov 2020
CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. CIBSE (P_3)
2022
CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R (P_4 ; em fase de submissão)

Figura 1 – Publicações originadas durante o período de doutoramento

A Figura 1 mostra as publicações em ordem cronológica, para as quais uma identificação foi atribuída (P_1, P_2, P_3 e P_4).

A Tabela 1 associa as publicações, mostradas na Figura 1, aos respectivos capítulos deste trabalho. A coluna "Id.", na Tabela 1, corresponde a identificação dada às publicações contidas neste trabalho. Adicionalmente, para cada publicação apresentada na Tabela 1, são informados, também, o tipo da publicação e a disposição da publicação no corpo desta tese.

Tabela 1 – Publicações durante o período de doutoramento

Id.	Tipo	Disponibilidade	Publicação
Publicação 1 (P_1)	Síntese da pesquisa	Capítulo 3	CAVAMURA, L. J. Operational profile and software testing: Aligning user interest and test strategy. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST) - Doctoral Symposium, 2019. p. 492–494.
Publicação 2 (P_2)	Artigo completo em periódico	Capítulo 5	CAVAMURA, L. J. et al. Software operational profile vs. test profile: Towards a better software testing strategy. Journal of Software Engineering Research and Development, v. 8, p.5:1 – 5:17, Aug. 2020.
Publicação 3 (P_3)	Artigo completo em conferência	Capítulo 4	CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. Perfil operacional do software: investigando aplicabilidades específicas. In: AYALA, C. P. et al. (Ed.). Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020, Curitiba, Paraná, Brazil, November 9-13, 2020. [S.l.]: Curran Associates, 2020. p.292–305.
Publicação 4 (P_4)	Artigo em fase de submissão	Capítulo 6	CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. Opdate: A unit test case generator based on real data from software operating profile. — , —, p. xxx-xxx(x), January 2022. ISSN x.

1.5.1 Relação entre o problema de pesquisa e as publicações

As contribuições providas pelas publicações que compõem o corpo desta tese fornecem as evidências que atendem ao objetivo geral e aos objetivos específicos descritos neste capítulo. A Tabela 2 apresenta os objetivos de cada publicação apresentada na Tabela 1 e as respectivas questões de pesquisa respondidas pelas mesmas.

A publicação P_1 é uma síntese das atividades de pesquisa desenvolvidas até a sua submissão no simpósio doutoral do *12th IEEE International Conference on Software Testing, Verification and Validation* (*12th ICST*). A pesquisa foi apresentada à comunidade científica por meio do simpósio doutoral do *12th ICST*. A relevância do tema abordado pela pesquisa e o interesse da comunidade científica nesse tema foram constatados por meio dos comentários fornecidos pela banca examinadora da pesquisa apresentada e, também, durante a apresentação dos trabalhos apresentados na trilha principal do *12th ICST*, dos quais alguns trabalhos exploraram o *POS* em suas abordagens. A participação no *12th ICST* permitiu firmar uma parceria com a profa. Dra. Ana Paiva da Universidade do Porto, Portugal, a qual proveu contribuições para a publicação P_2. A publicação P_1 é apresentada no Capítulo 3.

As publicações P_2, P_3 e P_4 abordam o problema de pesquisa investigado. As publicações P_2 e P_3 fornecem contribuições que evidenciam o problema investigado, ou seja, o possível descompasso entre o *POS* e as partes testadas do software. A publicação P_4 provê contribuições que permitem diminuir o possível descompasso entre o *POS* e as partes testadas do software.

A publicação P_2 é uma extensão de CAVAMURA et al. (2019). CAVAMURA et al. (2019) foi publicado no *XVIII Simpósio Brasileiro de Qualidade de Software* (SBQS 2019), sendo premiado como o melhor artigo (1^o colocado) da trilha principal. A publicação P_2 é apresentada no Capítulo 5. As questões de pesquisas respondidas pela publicação P_2 são mostradas na Tabela 2.

A publicação P_3 apresenta a revisão sistemática realizada para constatar a originalidade da pesquisa. Uma síntese dos resultados apresentados na publicação P_3 é apresentada na publicação P_2, provendo resposta a uma das questões de pesquisa da publicação P_2. Por investigar a originalidade da pesquisa abordada nesta tese, a publicação P_3 (Capítulo 4) é apresentada antes da publicação P_2 (Capítulo 5). As questões de pesquisas respondidas pela publicação P_3 são mostradas na Tabela 2.

A publicação P_4 aborda o problema de pesquisa investigado e apresenta uma ferramenta computacional, denominada *OPDaTe*. A ferramenta *OPDaTe* gera automaticamente, em nível de métodos, casos de teste de unidade executáveis usando dados reais, capturados dinamicamente, como dados de teste. Os recursos disponíveis na ferramenta *OPDaTe* permitem, à ferramenta, contribuir para a diminuição do possível descompasso entre o POS e as partes testadas do software. A publicação P_4, em fase de submissão, é apresentada no Capítulo 6. As questões de pesquisas respondidas pela publicação P_4 são mostradas na Tabela 2.

Conforme as informações apresentadas nesta seção, os Capítulos 3, 4, 5, 6 correspondem, respectivamente, às publicações P_1, P_3, P_2 e P_4, as quais compõem o corpo desta tese.

1.5.2 Considerações finais

Este capítulo forneceu o contexto, as motivações, a hipótese e os os objetivos desta tese. Este capítulo forneceu, também, a organização do texto, apresentando a estrutura de capítulos que compõem esta tese.

Tabela 2 – Objetivos e questões de pesquisa das publicações que compõem o corpo da tese

Publicação	Objetivo	Questões de Pesquisa Respondidas
P_1 (Capítulo 3)	Apresentar a pesquisa à comunidade científica de teste, verificação e validação de software; obter um *feedbak* dos pesquisadores sobre o tema abordado, os objetivos propostos e as atividades iniciais desenvolvidas	—
P_2 (Capítulo 5)	-Evidenciar: a) a originalidade da pesquisa; b) que há variações significativas na maneira como os usuários operam o software; c) o possível desalinhamento entre o *POS* e as partes testadas do software; d) que podem ocorrer falhas nas partes do *POS* não testadas; Introduzir o termo "perfil de teste" -Verificar se uma estratégia de teste baseada no uso de geradores de dados de teste automáticos pode contribuir para reduzir o possível desalinhamento entre o *POS* e as partes testadas do software	-Existem outros estudos com o mesmo objetivo ou objetivos semelhantes, cujos resultados fornecem as contribuições propostas nesta pesquisa? -Existem variações relevantes na forma como os usuários operam o software? -Pode ocorrer um desalinhamento entre o *POS* e as partes testadas do software? -Dado o possível desalinhamento entre o *POS* e as partes testadas do software, podem ocorrer falhas nas partes de *POS* não testadas? -Dado o desalinhamento entre o *POS* e as partes de software testadas, uma estratégia de teste incluindo gerador de dados de teste automatizado pode contribuir para reduzir o desalinhamento?
P_3 (Capítulo 4)	Investigar o uso do *POS* em contextos relacionados à pesquisa de doutoramento para formar uma base de conhecimento sobre o uso *POS* nas situações de uso investigadas. Situações de uso investigadas: a) em atividades associadas ao teste de regressão; b)como um critério de priorização aplicável a artefatos e atividades inerentes ao processo de desenvolvimento de software; c)como um critério de avaliação para casos de teste existentes.	-Como o *POS* foi usado nas técnicas identificadas nos estudos selecionados? -Como o *POS* foi obtido pelas técnicas identificadas? -Como o *POS* foi caracterizado pelas técnicas identificadas? -Relacionado ao uso do *POS*, quais foram as dificuldades reportadas pelos estudos selecionados?
P_4 (Capítulo 6)	Propor uma ferramenta computacional que possa ser usada para diminuir o possível descompasso entre o *POS* e as partes testadas do software, a qual gera casos de teste executáveis para as partes do software especificadas na ferramenta. Os casos de teste gerados usam dados de teste obtidos do *POS*.	-A *OPDdTe* foi capaz de gerar automaticamente casos de teste executáveis usando dados de teste extraídos do *POS*? -Os casos de teste executáveis gerados automaticamente forneceram uma cobertura satisfatória para as respectivas unidades testadas? -Uma estratégia de teste baseada em suítes de teste obtidas de diferentes perfis operacionais pode contribuir para expandir a cobertura das unidades testadas?

Capítulo 2

Perfil Operacional do Software e a Qualidade de Software

Este capítulo fornece a fundamentação teórica sobre o Perfil Operacional do Software e a Qualidade de Software.

2.1 Perfil Operacional do Software

O *POS* foi desenvolvido para obter uma especificação da maneira como o software é operado por seus usuários (MUSA; EHRLICH, 1996; SOMMERVILLE, 2015). Várias abordagens são propostas para descrever e construir o *POS*. Dentre essas abordagens, a abordagem definida por MUSA (1993), em decorrência de suas características, é uma das mais relevantes (LI; LUO; WANG, 2013). MUSA (1993) define o *POS* como uma caracterização quantitativa baseada na maneira com a qual o software é operado. Essa caracterização quantitativa corresponde às operações executadas pelo software, para as quais, uma probabilidade de ocorrência é atribuída. Uma operação corresponde a uma tarefa realizada pelo software, a qual é delimitada por fatores externos à implementação do software.

Uma operação realizada pelo software, em decorrência dos possíveis caminhos de execução contidos na implementação, pode, conforme os dados de entrada fornecidos, necessários à execução da operação, apresentar comportamentos diferentes e, consequentemente, fornecer resultados diferentes. Assim, uma operação pode ser executada de maneiras diferentes conforme os dados das variáveis de entrada por ela processados. Essas diferentes maneiras de execução são denominadas tipos de execução. A Figura 2 ilustra, como exemplo, uma representação gráfica das operações de um software e os respectivos tipos de execução de cada operação.

Os dados de entrada, que caracterizam um tipo de execução, formam um conjunto de dados denominado estado de entrada. Os estados de entrada (*EE*), associados aos tipos de execução, formam o espaço de entrada do software.

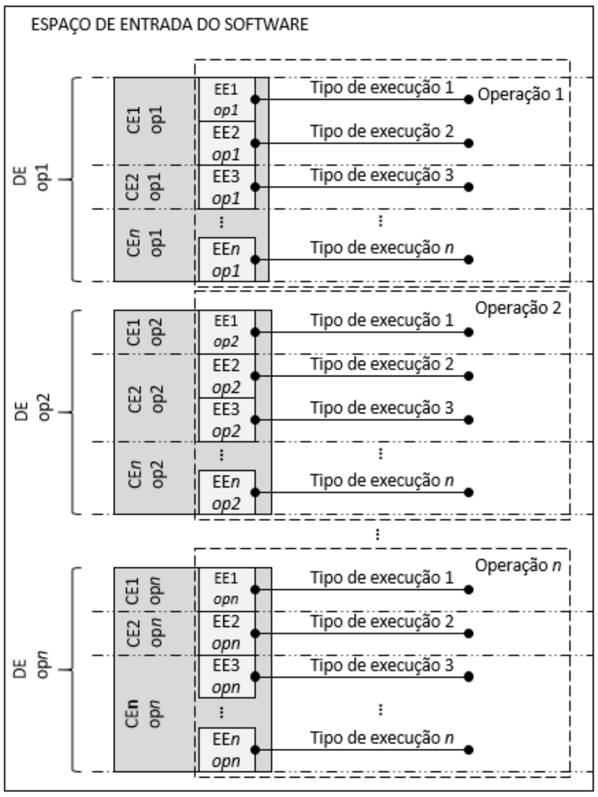


Figura 2 – Conceitos envolvidos na definição do perfil operacional

Como os estados de entrada caracterizam os tipos de execução de uma operação, o espaço de entrada do software pode ser fracionado por operações, associando, a cada operação, um conjunto de estados de entrada denominado domínio da operação. Assim, a cada operação disponibilizada pelo software, um domínio de entrada está associado, cujos elementos, estados de entrada da operação, determinam como a operação será executada, ou seja, definem o tipo de execução da operação. A Figura 2 ilustra uma representação gráfica dos estados de entrada, identificados por "*EE1, EE2, EE3,..,EEn*", o espaço de entrada do software e os domínios de entrada de cada operação, identificados por "*DE op1, DE op2,..,DE opn*".

O conjunto de operações disponíveis em um software é finito, porém, os tipos de execução de uma operação, por serem definidos pelo domínio de entrada da operação, correspondem a um conjunto cujo número de elementos pode ser infinito. Assim, para que seja possível atribuir uma probabilidade de ocorrência aos tipos de execução das operações, o domínio de entrada da operação é particionado em subdomínios. Cada subdomínio gerado corresponde a uma categoria de execução da operação. As categorias de execução de uma operação agrupam tipos de 15 execução cujos diferentes estados de entrada ocasionam um mesmo comportamento para a operação.

A Figura 2 ilustra uma representação gráfica das categorias de execução, identificadas por "*CE1, CE2,...,CEn*", que particionam o domínio de entrada de cada operação e agrupam os tipo de execução que possuem um mesmo comportamento para a operação. Por meio da Figura 2 percebe-se a relação existente entre os conceitos de operação, tipos de execução, estados de entrada, espaço de entrada, domínios de entrada e categorias de execução.

A Figura 3 ilustra, como exemplo, por meio de um modelo baseado em *árvore*, duas representações gráficas, itens *a)* e *b)*, de uma mesma operação. Cada possível caminho, da *raiz* até as *folhas*, corresponde a uma categoria de execução que representa um conjunto de tipos de execução que denotam um mesmo comportamento para a operação. Esse conjunto pode ser infinito porque cada tipo de execução é caracterizado pela combinação dos possíveis valores que *E1* e *E2* podem assumir. O domínio de entrada da operação corresponde ao conjunto formado por todos os possíveis valores que as variáveis *E1* e *E2* podem assumir para a operação.

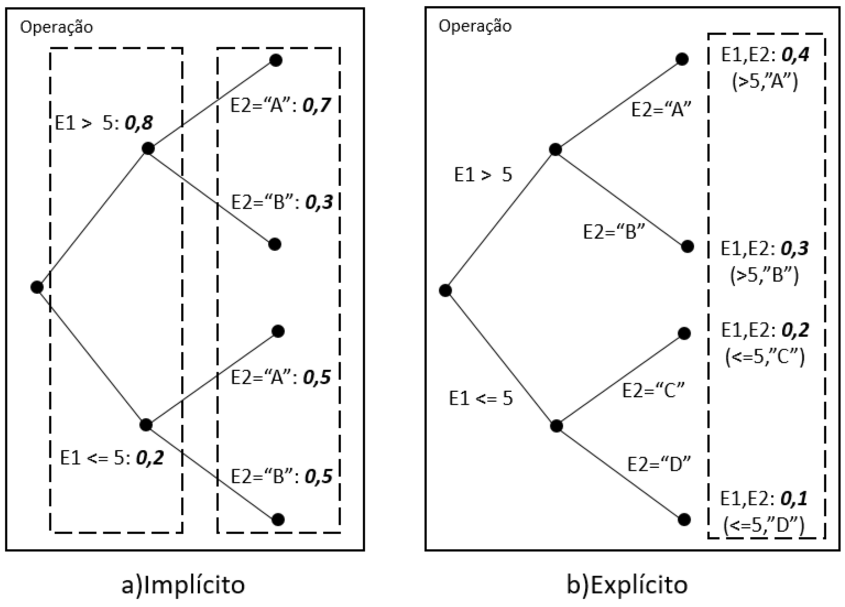


Figura 3 – Atribuição explícita e implícita

As categorias de execução de uma operação uma probabilidade de ocorrência é atribuída, obtendo-se assim uma caracterização quantitativa do software correspondente ao perfil operacional (MUSA, 1994; MUSA, 1993).

A construção do perfil operacional do software é realizada, idealmente, no início do processo de desenvolvimento do software. Modelos baseados em *árvores* e cadeia de *Markov* podem ser usados para construir o perfil operacional do software (SMIDTS et al., 2014). Os dados, que permitem determinar as probabilidades de ocorrência das operações, podem ser obtidos por meio de arquivos de *log* gerados pelo uso de versões anteriores do software cujo perfil operacional do software está sendo especificado ou, também, gerados por softwares similares (MUSA, 1993; TAKAGI; FURUKAWA; YAMASAKI, 2007). A expectativa dos desenvolvedores também pode determinar essas probabilidades (TAKAGI; FURUKAWA; YAMASAKI, 2007).

As probabilidades de ocorrência podem ser atribuídas de maneira explicita ou implícita. A atribuição explícita corresponde à probabilidade de ocorrência definida e atribuída para um conjunto de valores, pertencentes ao domínio da operação, que definem uma categoria de execução e para os quais existe uma relação de dependência entre si. A atribuição implícita corresponde à probabilidade de ocorrência definida e atribuída diretamente aos valores das variáveis de entrada que são independentes de outras variáveis. A Figura 3 ilustra um exemplo que representa as atribuições implícita e explícita, itens *a)* e *b)* respectivamente.

As probabilidades de ocorrência, ilustradas na Figura 3 itens *a)* e *b)*, correspondem aos valores descritos em negrito e itálico. Verifica-se, pela Figura 3 item *a)*, que os possíveis valores para a variável de entrada *E2* não dependem dos possíveis valores para a variável *E1*, sendo assim, a probabilidade de ocorrência é atribuída diretamente às variáveis. A Figura 3 item *b)* mostra que os possíveis valores para a variável de entrada *E2* dependem dos possíveis valores para a variável *E1*, sendo assim, a probabilidade de ocorrência é atribuída para o conjunto de valores que caracteriza a categoria de execução.

Por representar a maneira como o software será utilizado por seus usuários e sendo a confiabilidade de um software dependente do contexto em que o software é usado, o *POS* é usado em atividades relacionadas a engenharia de confiabilidade de software. Nessas atividades, o *POS* tem, como propósito, gerar dados de teste que, quando aplicados à execução do software em um ambiente de testes, reproduzem a maneira com a qual o software é executado no ambiente de produção, garantindo que os indicadores de confiabilidade do software sejam válidos para a operação do software em um ambiente de produção (MUSA; EHRLICH, 1996). Assim, o *POS* não garante que todos os defeitos sejam detectados, mas garante que as operações mais utilizadas do software sejam devidamente testadas conforme a maneira com a qual são executadas (ALI-SHAHID; SULAIMAN, 2015).

2.2 O Perfil Operacional do Software e a Qualidade de Software

O software é decorrente da manifestação do caráter criativo do intelecto humano (ASSESC, 2012) e, assim, defeitos podem ser inseridos no software em decorrência de enganos ocorridos durante o processo de desenvolvimento. O custo para se corrigir um defeito no software aumenta conforme o tempo entre o momento em que o defeito é encontrado e o momento em que o mesmo é inserido (BOEHM; BASILI, 2001). Por ser conveniente ao cotidiano da sociedade contemporânea, o software, quando falha, interrompe as mais variadas atividades da sociedade. Em 2017, 3,6 bilhões de pessoas foram afetadas e 1,7 trilhão de receita foi perdida em decorrência de falhas de software (TRICENTIS, 2018). Estima-se que 65% do tempo total consumido pelo processo de desenvolvimento é decorrente do processo de manutenção no software, processo que abrange, dentre outras atribuições, a correção dos defeitos que originaram falhas (TAKAGI; FURUKAWA; YAMASAKI, 2007). Assim, atividades de garantia de qualidade são essenciais pois visam diminuir os riscos inerentes ao processo de desenvolvimento de software. As atividades de garantia da qualidade são executadas durante todo o processo de desenvolvimento do software com o objetivo de fornecer um software com qualidade. Nesse contexto, baseado nos fundamentos de gestão de qualidade, entende-se por qualidade, segundo **Crosby**, citado em (SOMMERVILLE, 2015), a conformidade do produto com a sua especificação detalhada e a noção de tolerâncias relacionadas à especificação. Tendo como contexto a Engenharia de Software, PRESSMAN (2019) define qualidade de software como um processo eficaz para a criação de um produto de valor à quem o produz e a quem irá utilizá-lo. A qualidade, quando analisada, pode ser passível de subjetividade, dado que a interpretação das características analisadas pode variar conforme o ponto de vista do analista. Assim, a qualidade do software pode ser analisada conforme a perspectiva com a qual é abordada. Na visão do usuário, por exemplo, um software com qualidade é o que atende suas necessidades, é fácil de usar e é confiável (FALBO, 2005).

A confiabilidade do software, assim como a manutibilidade, eficiência, entre outros, é um dos atributos associados à qualidade de software e a representa a partir do ponto de vista do usuário. (MUSA, 1979; BITTANTI; BOLZERN; SCATTOLINI, 1988). Um defeito que afeta a confiabilidade para um usuário pode nunca ser revelado no modo de trabalho de outro usuário (SOMMERVILLE, 2015). Para o usuário do software, a frequência com que uma falha se torna aparente, durante a operação do software, é mais significante que falhas remanescentes (TAKAGI; FURUKAWA; YAMASAKI, 2007).

A confiabilidade do software corresponde a probabilidade de operação do software, por um determinado tempo, em um ambiente específico, sem a ocorrência de falhas (MUSA, 1979; CUKIC; BASTANI, 1996). Essa definição é exemplificada por PRESSMAN (2019): Se um programa de computador, cuja confiabilidade é de 0,999 (99%) para oito horas de processamento, for executado 1000 vezes, demandando 8 horas de tempo de execução, é estimado um funcionamento correto, livre de falhas, em 999 vezes.

Sendo a confiabilidade determinada pelo tempo da operação do software sem a ocorrência de falhas, e que uma falha revelada a um usuário pode nunca ser revelada no modo de trabalho de outro usuário, a atividade de teste está implícita à confiabilidade do software, a qual depende da maneira como o software será operado por seus usuários (MUSA, 1994; MUSA, 1993; SOMMERVILLE, 2015). Assim, para se mensurar a confiabilidade, o software deveria ser testado como base no POS, simulando, durante a execução do teste, a maneira como o software é operado pelos usuários finais do produto de software. Um modelo de uso do software, que representa o POS, é usado para projetar os casos de teste a serem executados. Conforme descrito na seção anterior, o *POS* não garante que todos os defeitos sejam detectados, mas garante que as operações mais utilizadas sejam testadas suficientemente (ALI-SHAHID; SULAIMAN, 2015).

Na literatura, alguns estudos que abordaram técnicas de teste que usaram o modelo de uso do software classificaram, essas técnicas, como teste estatístico ou teste de uso estatístico, teste de confiabilidade, teste baseado em modelos, teste baseado em uso ou teste baseado no *POS* (POORE; WALTON; WHITTAKER, 2000; KASHYAP, 2013; SOMMERVILLE, 2015; PRESSMAN, 2019; MUSA; EHRLICH, 1996). Neste trabalho, o teste de software, a partir de um modelo de uso, será referenciado como "teste baseado no *POS*".

Dados coletados enquanto o software é operado pelos usuários são usados para construir o modelo de uso do software. Esses dados fornecem os elementos que compõem o modelo de uso e, também, uma distribuição probabilística entre esses elementos. A distribuição probabilística refere-se à probabilidade da ocorrência ou a frequência de execução desses elementos durante a operação do software por seus usuários. POORE; WALTON; WHITTAKER (2000) atribuem o termo "perfil de uso" à distribuição probabilística associada ao modelo de uso. Os elementos que compõem o modelo de uso correspondem à entidades relacionadas ao software, cuja frequência de execução ou probabilidade de ocorrência permitem identificar as partes do software mais, ou menos, usadas pelos usuários.

O tipo das entidades adotado para construir o modelo de uso do software é definido de acordo com a granularidade com a qual o software será abordado para a definição do perfil operacional, ou seja, em qual nível de acesso os dados serão extraídos durante a operação do software pelos usuários. Assim, essas entidades, que compõem o modelo de uso, podem representar operações realizadas pelo software, estados que o software pode assumir, dados de entrada fornecidos ao software, entre outros.

Outras atividades, além das necessárias para se obter o modelo de uso e realizar o teste baseado no *POS*, são necessárias para obter e avaliar a confiabilidade do software. Essas atividades estão inseridas no processo que constitui a Engenharia de Confiabilidade de Software (*ECS*).

A *ECS* compreende a tecnologia e a prática para (MUSA; EHRLICH, 1996):

1. Definir objetivos quantitativos de confiabilidade que, se atendidos, maximizam a satisfação do cliente;

2. Projetar e desenvolver o software para atender aos objetivos definidos;

3. Focar o desenvolvimento e o teste nas operações mais utilizadas e críticas;

4. Testar o software para obter os resultados referentes aos objetivos definidos.

As fases que compõem o processo da *ECS* são: *requisitos*, *arquitetura*, *teste* e *operação*. Essas fases estão correlacionadas às fases do processo de desenvolvimento do software e são descritas a seguir.

2.2.1 Requisitos

Na fase de requisitos o *POS* é definido com base nos requisitos e em informações obtidas pelos envolvidos na elicitação dos requisitos.

TAKAGI; FURUKAWA; YAMASAKI (2007) mencionam que arquivos de *log* gerados por softwares semelhantes ou *releases* anteriores do software, cujo *POS* está sendo definido, podem prover informações para a definição do *POS*, podendo também, na ausência de fontes de informação, ser usada uma distribuição probabilística uniforme para as operações do software. Ainda na fase de requisitos, uma análise de riscos, relacionada a falhas que podem ocorrer na operação do software e as consequências dessas falhas, é feita para identificar classes de falhas e classificá-las conforme a gravidade das falhas que as classes representam. São definidas, também, as taxas de falha por unidade de tempo, denominadas *objetivos*. Entende-se por taxa de falha a frequência aceitável com que as falhas poderão ocorrer. Vários *objetivos* podem ser definidos para o software. A definição desses *objetivos* é feita com base nas classes de falhas que foram identificadas e classificadas. Os objetivos, ou seja, as frequências com que as falhas poderão ocorrer, serão menores para as classes de falhas com maior severidade.

2.2.2 Arquitetura

A arquitetura de software refere-se ao software como uma estrutura organizada em que as partes que o compõem, componentes de software, interagem entre si e manipulam estruturas de dados (PRESSMAN, 2019). Nesse contexto, esses componentes podem ser abstraídos em módulos obtidos por uma divisão natural do software, os quais podem ser endereçados e gerenciados separadamente e se integram para atender aos requisitos do software (PRESSMAN, 2019; MUSA; EHRLICH, 1996). Nessa fase, os objetivos definidos na fase anterior são atribuídos aos componentes do software e as estratégias de confiabilidade, a serem adotadas, são definidas. As estratégias de confiabilidade referem-se aos níveis de tolerância a falhas que serão implementados. Assim, se os objetivos definidos forem baixos, ou seja, é definida uma baixa taxa de falhas, haverá um impacto no tempo de desenvolvimento, e consequentemente no custo, decorrente das estratégias de tolerância a falhas que deverão ser implementadas. Se o prazo para o desenvolvimento é escasso e os objetivos são moderados, as estratégias de tolerância a falhas podem ser minimizadas considerando que a atividade de teste, a ser realizada, seja eficaz. A Figura 4 mostra o relacionamento entre o custo e melhorias relacionadas a confiabilidade.

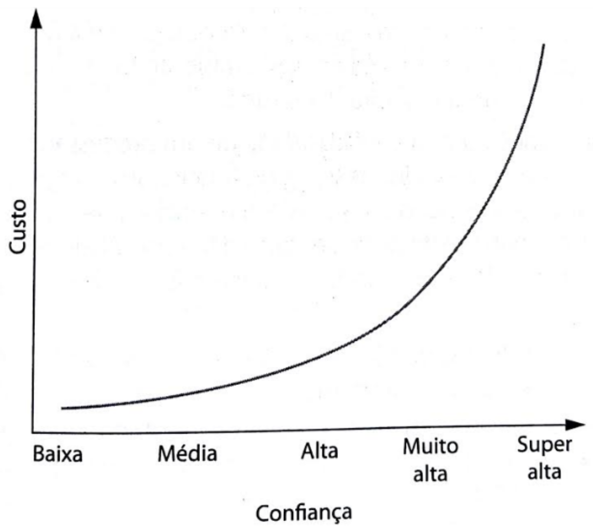


Figura 4 – Relacionamento entre o custo e melhorias relacionadas a confiabilidade. Obtido de (SOMMERVILLE, 2015)

.

Nessa fase, o *POS* e a criticidade das operações podem, também, ser usados para organizar, planejar e alocar recursos às atividades de desenvolvimento visando entregar, inicialmente, as funcionalidades altamente usadas e críticas.

2.2.3 Teste

Nessa fase deve-se assegurar que os objetivos atribuídos aos componentes do software sejam alcançados e, consequentemente, reduzir e garantir que o objetivo atribuído ao software como um todo seja alcançado.

Os componentes do software devem ser testados com o *POS* e, para isso, os componentes podem ser integrados. Casos de teste são selecionados usando o *POS*. As falhas detectadas devem registradas e associadas ao componente testado que às evidenciaram, registrando, também, a frequência com a qual ocorrem. Essas informações devem ser monitoradas e gerenciadas para verificar se os objetivos atribuídos são atendidos. A Figura 5 mostra, como exemplo, um gráfico gerado com as informações que são registradas, o qual permite determinar, como base no objetivo atribuído, se o componente testado é aceito, rejeitado ou se deve continuar a ser testado.

Como exemplo, com base na Figura 5, um componente que apresentou três falhas em um intervalo de tempo de 100 horas de processamento, é aceito com base no objetivo atribuído à ele. Um componente que apresentou mais que cinco falhas em um intervalo de tempo inferior a 20 horas de processamento, deve ser rejeitado e, assim, retrabalhado para garantir o objetivo atribuído à ele.

Com base nos resultados obtidos pelo teste, a confiabilidade do software é estimada. Um programa de estimativa pode ser usado para estimar a confiabilidade processando os dados de falha.

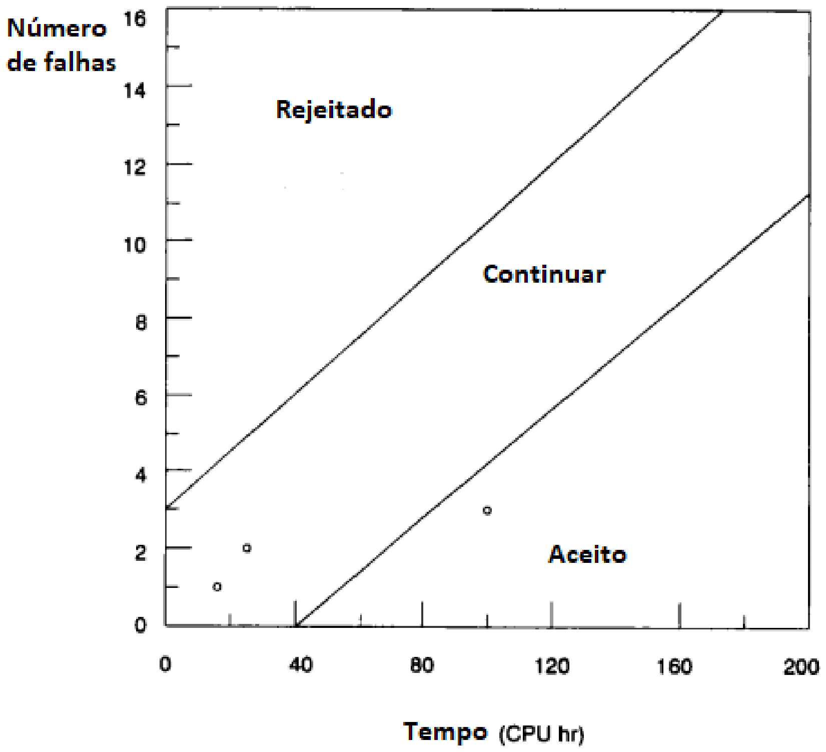


Figura 5 – Gráfico de confiabilidade. Adaptado de (MUSA; EHRLICH, 1996)

2.2.4 Operação

A engenharia de confiabilidade de software pode ser aplicada em campo para avaliar a introdução de novos recursos e orientar a melhoria do produto e do processo de desenvolvimento. Quando novas funcionalidades são agregadas ao software que está sendo operado em ambiente de produção, naturalmente, há um aumento na taxa de falhas, a qual diminui a medida que os defeitos que as originaram são corrigidos, estabelecendo-se, assim um padrão quando novas funcionalidades são agregadas ao software. Quando novas funcionalidades são inseridas e identificam-se variações significativas em relação ao padrão estabelecido, como por exemplo, uma taxa de falhas superior ao previsto, comprometendo a confiabilidade, as causas dessas variações são investigadas e a inserção de novas funcionalidades são interrompidas, possivelmente restaurando o software a uma versão anterior com taxas de falhas consideradas aceitáveis.

2.3 Dificuldades e Problemas relacionados ao uso do *POS*

Embora o *POS* seja um atributo que representa a qualidade de software do ponto de vista do usuário, sendo essencial para se determinar a confiabilidade do software, dificuldades e problemas quanto ao uso do *POS* são identificados na literatura. A Tabela 3 apresenta algumas dessas dificuldades e problemas identificados.

Tabela 3 – Problemas relacionados ao uso do *POS*

Referência	Ano de Publicação	Problema apontado
(NAMBA; AKIMOTO; TAKAGI, 2015)	2015	A definição do *POS* demanda alto esforço, tornando-se difícil conforme a complexidade do software
(SHUKLA, 2009)	2009	Estudos relacionados ao *POS* focam em explorar as operações. Os parâmetros das operações são pouco explorados.
(LEUNG, 1997)	1997	Erros de estimativa são inevitáveis e o *POS* muda quando o software está em operação em um ambiente de produção
(FUKUTAKE et al., 2015)	2015	A probabilidade de uso torna-se pequena quando o modelo de uso do software possui vários estados.
(CUKIC; BASTANI, 1996)	1996	Identificar o *POS* é difícil pois é necessário prever o uso do software.
(BERTOLINO et al., 2017)	2017	Teste baseado em perfil operacional, focando nas falhas com maior probabilidade de ocorrência, pode sofrer saturação e perder eficácia
(SOMMERVILLE, 2015)	2015	A confiabilidade do software depende do contexto em que será usado e usuários experientes adaptam o seu comportamento

Conforme é mostrado na Tabela 3, existem problemas relacionados à identificação do *POS* relatados na literatura. Embora existam dificuldades em identificar e manter o *POS*, a qualidade do software depende do uso operacional do software (HE et al., 2021; ZHAKIPBAYEV; BEKEY, 2021). Assim, o *POS* pode contribuir com a qualidade do software fornecendo diretrizes às atividades de garantia da qualidade, permitindo que tais atividades possam considerar, também, o foco nas partes mais usadas do software.

2.4 Considerações Finais

Este capítulo forneceu a fundamentação teórica referente à pesquisa apresentada no Capítulo 1. A definição e os conceitos relacionados ao *POS*, assim como a relação entre o POS e a qualidade de software, foram apresentados. Os Capítulos seguintes, de 3 a 6, correspondem às publicações originadas desse trabalho até o momento e que compõem o corpo desta tese. Capítulo 7 apresenta as conclusões, trabalhos futuros e as lições aprendidas provenientes da pesquisa realizada.

Capítulo 3

Perfil operacional e teste de software: Alinhando o interesse do usuário e a estratégia de teste

Publicação referente a apresentação da pesquisa no "12th IEEE Conference on Software Testing, Validation and Verification (ICST)"

CAVAMURA, L. J. Operational profile and software testing: Aligning user interest andtest strategy. In: Proceedings of the 12th IEEE Conference on Software Testing, Validationand Verification (ICST) - Doctoral Symposium, 2019. p. 492–494.

Operational Profile and Software Testing: Aligning User Interest and Test Strategy

1st Luiz Cavamura Jnior
Computer Department
UFSCar
Sao Carlos, Brazil
luiz_cavamura@ufscar.br

***Abstract*—*Context*: The software's operational profile is a representation of how users use the software, in which the most commonly-used parts of the software are identified. There is evidence of a possible mismatch between the tested parts of the software and the operational profile of the software. *Objective*: This paper aims to present a doctoral research that investigates the use of software operational profile as a resource to improve software quality from the point of view of users. As one of the expected contributions, the research proposes a strategy in which the operational profile of the software is used to: a) evaluate and adapt a test suite based on the operational profile of the software; b) as a prioritization criterion that allows, given a set of defects, to identify the defects that have the greatest impact on the operation of the software by the users and, thus, to consider this impact in the pricing of the defects. *Method*: To present the research, this paper describes: a) the problem addressed and the purpose of the research; b) the results obtained by the activities carried out to evaluate the feasibility of the research (systematic mapping of the literature, systematic review of the literature and exploratory study); c) the hypotheses to be investigated; d) expected contributions; e) the current state of the research. *Results*: The results obtained by the activities carried out were favorable to the accomplishment and continuity of the research, evidencing the originality of the research and the proposed strategy. *Conclusion*: Once the feasibility of the research and the progress that has been obtained in its execution are verified, the results to be obtained are promising and, consequently, the expected contributions must be obtained.**

***Index Terms*—operational profile, software testing, software engineering**

I. Introduction

Software, as a strategic element in several processes of contemporary society, needs to be able to assimilate new ways of using it as a result of the evolution of the processes for which it was created. Thus, software features, even if designed to meet business rules common to many organizations, can be parameterized to meet specific needs, providing different ways of using them. The different ways of using the software can also be derived from the creative character of the human intellect, which allows users, based on professional experiences, to adapt software behavior [1]. The Software's Operational Profile (*SOP*) corresponds to a quantitative representation of the software based on the way the software is operated by the users, which is comprised of the software operations and a probabilistic distribution associated with those operations [2]. Thus information about the *SOP* becomes relevant to ensure the quality of the software. This paper describes a doctoral research that investigates the use of *SOP* as a resource to ensure and improve the quality of the software and is organized as follows: Section II exposes the problem addressed and presents the research with its respective objective. Section III presents the results obtained by the activities that evaluated the feasibility of the research. Section IV, V and VI present, respectively, the hypotheses to be investigated, the expected contributions and the current state of the research. The conclusions are described in Section VII.

II. About the Research and Problem addressed

One study [3] evidenced that information about the "SOP" was considered essential or relevant to issues related to the activities of the software development process. Examples of these questions are: "*Which parts of the software are most commonly used?*", "*How do users use the application?*" and "*How does the coverage of the tests correspond to the code actually executed by the users?*". Another study dynamically analyzed 10 free/libre/open source softwares found that only one reached a percentage of coverage, provided by the test suite, between 70 and 80 [4]. Even if the interval is considered acceptable, there is a percentage of code, not covered, that may be related to the features most used by users, evidencing a possible mismatch between the parts tested and the parts that users actually use. This mismatch can become frequent to the occurrence of failures to the users, providing a critical situation because, in users' perception, the frequency with which a defect becomes apparent is more significant than the defects still remaining in the software, which makes the quality of the software depending on its operational use [5]. Thus, there are evidences of a possible mismatch between the *SOP* and the tested parts of the software, and also that informations about *SOP* are relevant to ensure the quality of the software. The research, presented in this paper, investigates this possible mismatch and the contributions that *SOP* can provide to software quality, specifically to the test activities. The use of *SOP* does not guarantee that all defects are detected, but it ensures that the most used parts of the software are sufficiently tested [6]. The research proposes a strategy to dynamically adapt an existing test suite to the *SPO* and also to use *SOP* as a prioritization criterion that allows, given a set of defects, to identify the defects that it has greater impact on the operation of the software by the user and, thus, consider this impact, along with other criteria, such as the criticality of the operation, in the pricing of defects.

III. Feasibility Analysis

The feasibility of the research and the uniqueness of the proposed strategy were evaluated by the following activities: a) Exploratory study (*EE*) to show that there are relevant variations in the way the software is operated by the users and that there may be a mismatch between the tested parts of the software and the *SOP*; b) Systematic Literature Mapping (*SLM*) and Systematic Literature Review (*SLR*) to provide the theoretical basis and evidence the originality of the research and the proposed strategy. The *EE* identified and evaluated the operational profile of three software, *s1*, *s2* and *s3*. The variation in the way the software is operated by the users was evaluated with the data obtained from the operation of *s1*, which originated 30 representations for the *SOP*. These representations differed by 51% between themselves, evidencing that there are relevant variations in the way the software is operated. The EE also showed, in all three softwares evaluated, a mismatch between the *SOP* and the tested parts of the software. Checked that 46.79%, 25.08% and 46.45% of the methods processed by at least 1 participant, in the operation of *s1*, *s2* and *s3* respectively, were not covered by the test suite of their software. By analyzing data extracted from the platform that provides and manages the source code of *s2*, it was verified that in 13.86% of the methods processed in the operations performed by the users, not covered by the test suite, there were records of modifications related to reported problems, providing indications of the possible occurrence of failures in the most used parts of the software and not covered by the test suite.

The *SLM* identified 4726 studies, of which 182 studies were selected for data extraction. The analysis of the extracted data evidenced that the quality of software is the area most approached by the selected studies, however, there are few selected studies related to this area, which characterizes it, together with the other identified areas, as an area that can be explored.

The *SLR* identified 874 studies, of which 11 studies were selected for data extraction. The analysis of the extracted data evidenced that the *SOP* is used in techniques for the generation, selection and prioritization of test cases. The use of *SOP* as a criterion for assessing and adapting an existing set of test cases has not been identified. It is also emphasized that the *SOP* was not approached as a prioritization criterion related to the impact of the defects in the operation of the software by the users.

IV. Hypotheses to be investigated

Once the feasibility of the research and the proposed strategy have been verified, the following hypotheses to be investigated are identified:
H_1: There is a mismatch between the *SOP* and the coverage provided by the software test suite.
H_2: The use of *SOP* is relevant and contributes to the improvement of software quality from the point of view of the user.

V. Expected Contributions

The research is expected to provide software engineering with the following contributions: a)

1) Definition of a strategy that allows: to dynamically evaluate and adapt an existing test suite based on *SOP* to cover the most used parts of the software; to expand the use of *SOP* as a prioritization criterion to identify the defects that have the greatest impact on the operation of the software by the user, and thus to consider this impact, along with other criteria, in the pricing of defects.
2) Implementation of a computational tool to automate the proposed strategy.

VI. Current Search Status

The research presented is in the initial phase. The activities already carried out had as objective to evidence the viability of the research and the proposed strategy, to define the proposed strategy structure and to start the implementation of the computational tool that automates the proposed strategy.

The activities to be carried out to continue the research include: a) conducting new experiments to ascertain the mismatch or consonance between the *SOP* and the software's test suite; b) finalize the definition of the proposed strategy; c) conducting experiments to evaluate the defined strategy; d) refactoring the strategy defined according to the results obtained; e) realization of new experiments to evaluate the refactored strategy; f) development of a computational tool; g) write and publish papers to disseminate the results obtained; h) write and defend the doctoral thesis.

VII. Conclusions

This paper presented a doctoral research that investigates the use of *SOP* as a resource to improve software quality from the point of view of users. The research proposes a strategy in which the *SOP* is used to evaluate and adapt an existing test suite and as a prioritization criterion to rank the impact of the defects in the operation of the software by the users, contributing to the pricing of the defects. The research is in the initial phase and its execution is in line with the deadlines that were planned. Given the results obtained and progress of the activities being carried out, it is estimated that the data to be obtained by the research is promising and thus the expected results are obtained.

References

[1] I. Sommerville, *Software Engineering*, 9th ed. Harlow, England: Addison-Wesley, 2010.
[2] J. Musa, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, no. 2, pp. 14–32, 1993, cited By 396.

[3] A. Begel and T. Zimmermann, “Analyze this! 145 questions for data scientists in software engineering,” *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pp. 12–23, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568225.2568233

[4] A. M. Rincon, “Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais,” 2011.

[5] B. Cukic and F. B. Bastani, “On reducing the sensitivity of software reliability to variations in the operational profile,” Anon, Ed. White Plains, NY, USA: IEEE, Los Alamitos, CA, United States, 1996, pp. 45–54, cited By 6; Conference of Proceedings of the 1996 7th International Symposium on Software Reliability Engineering, ISSRE’96 ; Conference Date: 30 October 1996 Through 2 November 1996; Conference Code:45835.

[6] M. b. Ali-Shahid and S. Sulaiman, “Improving reliability using software operational profile and testing profile.” Institute of Electrical and Electronics Engineers Inc., 2015, pp. 384–388, cited By 1; Conference of 2nd International Conference on Computer, Communications, and Control Technology, I4CT 2015 ; Conference Date: 21 April 2015 Through 23 April 2015; Conference Code:114412.

Capítulo 4

Perfil operacional do software: investigando aplicabilidades específicas

Artigo publicado no "Iberoamerican Conference on Software Engineering" (CIbSE)

CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. Perfil operacional do software:investigando aplicabilidades específicas. In: AYALA, C. P. et al. (Ed.).Proceedings ofthe XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020,Curitiba, Paraná, Brazil, November 9-13, 2020. [S.l.]: Curran Associates, 2020. p.292–305.

Perfil Operacional do Software: investigando aplicabilidades específicas

Luiz Cavamura Júnior[1], Sandra Fabbri[1], and Auri M. R. Vincenzir[1]

Departamento de Computação, Universidade Federal de São Carlos (UFSCar)
São Carlos, Brasil
{luiz_cavamura,sfabbri,auri}@ufscar.br

Resumo O Perfil Operacional do Software (*POS*) é uma representação da maneira como os usuários usam o software, na qual as operações, executada pelo software, são associadas a uma probabilidade de ocorrência. Sendo a confiabilidade de um software dependente do contexto no qual ele é usado, o *POS* é empregado na engenharia de confiabilidade de software. Os resultados obtidos pela realização de um Mapeamento Sistemático da Literatura (*MSL*) evidenciaram que a Qualidade de Software é a área mais favorecida pelas contribuições providas pelo uso do *POS* no processo de desenvolvimento de software. Contudo, poucos foram os estudos selecionados associados à essa área, caracterizando-a, junto às demais áreas, como uma área que pode ser explorada. Esses resultados motivaram a realização de uma Revisão Sistemática da Literatura (RSL) que investigou o uso do *POS* como um critério de avaliação para casos de teste existentes, ou seja, sob a perspectiva de uso do software, e como um critério de priorização aplicável a artefatos e atividades inerentes ao processo de desenvolvimento de software. Como o *POS* representa a maneira como o software é operado pelos usuários e o teste de regressão é realizado quando há modificações no software, podendo ocorrer enquanto o software está em operação, o uso do *POS* em atividades de teste de regressão também foi investigado. O propósito deste artigo é prover, como contribuições, os resultados obtidos pela *RSL* realizada. Por meio dos resultados obtidos, constatou-se que, sendo um recurso adotado para determinar a confiabilidade do software, o *POS* é usado em técnicas de teste estatístico e de regressão para gerar, selecionar e priorizar os casos de teste. O uso do *POS* em outros contextos de teste e com outros propósitos não têm sido explorado. Constatou-se, também, que por ser um recurso usado em tais técnicas, informações detalhadas sobre o *POS* são escassas. Os resultados obtidos permitiram, também, caracterizar o uso do *POS* nas situações investigadas.

Keywords: Perfil Operacional · Revisão Sistemática · Qualidade de Software · Teste de Software · Perfil de Teste.

1 Introdução

A necessidade de processos eficientes faz com que as organizações estejam empenhadas na melhoria desses processos. O software é um elemento estratégico nesses processos e, assim, pode ter recursos que permitam ser parametrizado para se adaptar às diversificadas maneiras de operação, atendendo às necessidades específicas de cada organização. Nesse contexto, visando a praticidade e eficiência em suas atividades, usuários experientes conseguem adaptar a maneira como o software é operado [24]. Como a qualidade do software é dependente do seu uso operacional [8], a maneira como o software é operado torna-se relevante ao processo de desenvolvimento de software. O Perfil Operacional do Software (*POS*) corresponde a uma especificação do software que representa a maneira como o software é operado por seus usuários [8,24]. Musa [16] define o *POS* como uma caracterização quantitativa do software. Essa caracterização é baseada nas operações do software, para as quais uma distribuição probabilística é atribuída para indicar as partes do software mais usadas pelos usuários.

Dada a relevância do *POS* ao processo de desenvolvimento de software, este trabalho relata a realização de uma Revisão Sistemática da Literatura (*RSL*) que investigou o uso do *POS* sob três perspectivas (situações de uso): em atividades associadas ao teste de regressão (SU_1); como um critério de priorização aplicável a artefatos e atividades inerentes ao processo de desenvolvimento de software (SU_2); e como um critério de avaliação para casos de teste existentes (SU_3). As situações de uso investigas pela *RSL* foram definidas e estão associadas a uma pesquisa de doutoramento que investiga o *POS* em um escopo mais amplo [5]. A investigação do uso do *POS* em técnicas de teste de regressão foi motivada pela natureza do teste de regressão, o qual é executado quando modificações são feitas em um software existente [28]. No momento em que o teste de regressão é realizado, o software pode fornecer dados sobre a maneira como é operado por seus usuários. O *POS*, por representar a maneira como o software é operado pelos usuários, corresponde a um critério de priorização baseado no uso. A investigação do uso do *POS* como critério de priorização busca identificar outras atividades e recursos que podem adotar o *POS* como um critério de priorização. A investigação do uso do *POS* como um critério de avaliação dos testes existentes, gerados por outros critérios, foi definida em decorrência do possível descompasso entre as partes testadas do software e o *POS* [6]. O uso do *POS* como critério de priorização e avaliação são exemplos de uso que podem contribuir com a qualidade do software do ponto de vista dos usuários e, também, ser usado para priorizar atividades do processo de desenvolvimento do software de maior interesse do usuário, tais como manutenção, melhorias e customizações.

Este trabalho provê, como contribuições, a partir dos resultados obtidos, uma base de conhecimento sobre o uso do *POS* nos contextos investigados, fornecendo uma caracterização e entendimento quanto ao uso do *POS* nesses contextos. Este trabalho está organizado da seguinte forma: na Seção 2, os conceitos que envolvem o *POS* e a qualidade de software são abordados. A Seção 3 refere-se aos trabalhos relacionados. A Seção 4 descreve a metodologia adotada para a realização da pesquisa. As Seções 5, 6, 7, 8 e 9 descrevem a *RSL* realizada e os respectivos resultados. As ameaças à validade deste trabalho e as conclusões são apresentadas nas Seções 10 e 11 respectivamente.

2 *POS* e a Qualidade de Software

Por representar a maneira como o software será utilizado por seus usuários e sendo a confiabilidade de um software dependente do contexto em que o software é usado, o *POS* é usado em atividades relacionadas a engenharia de confiabilidade de software. Nessas atividades, o *POS* é usado para gerar dados de teste que reproduzem a maneira com a qual o software é operado pelos usuários, garantindo que os indicadores de confiabilidade do software sejam válidos para a operação do software em um ambiente de produção [17]. A confiabilidade do software é um dos atributos associados à qualidade de software e a representa a partir do ponto de vista do usuário [4,15]. Para o usuário do software, a frequência com que uma falha se torna aparente, durante a operação do software, é mais significante que as falhas remanescentes [26]. A confiabilidade do software corresponde a probabilidade de operação do software, por um determinado tempo, em um ambiente específico, sem a ocorrência de falhas [8,15]. Sendo a confiabilidade determinada pelo tempo da operação do software sem a ocorrência de falhas, e que uma falha revelada a um usuário pode não ser revelada no modo de trabalho de outro usuário, a atividade de teste está implícita à confiabilidade do software. O *POS* não garante que todos os defeitos sejam detectados, mas garante que as operações mais utilizadas do software sejam testadas.

Um Mapeamento Sistemático da Literatura (*MSL*) investigou como as contribuições providas pelos estudos que abordaram o uso do *POS* estão distribuídas nas áreas da engenharia de software correlatas ao processo de desenvolvimento de software, evidenciando que a qualidade de software é a área mais abordada pelos estudos que usaram o *POS* como um recurso nas estratégias abordadas nesses estudos. A maioria dessas estratégias estão associadas à confiabilidade de software. Embora a qualidade de software seja a área mais abordada, são poucos os estudos relacionados a essa área, situação que à caracteriza, junto às demais áreas identificadas, como uma área que pode ser explorada. A descrição detalhada do *MSL* realizado, e dos respectivos resultados, está disponível em: lcvm.com.br/artigos/anexos/cibse2020/cap-3-rs-ms.pdf.

3 Trabalhos Relacionados

Dentre os trabalhos selecionados pelo *MSL*, Smidts et al. [23] forneceram uma caracterização do modelo baseado no *POS* usado na geração de casos de teste, assemelhando-se a este trabalho dada a abordagem investigativa focada diretamente no *POS*. Os demais trabalhos usaram o *POS* como um recurso às estratégias propostas, não fornecendo, detalhadamente, informações sobre o *POS*. A caracterização, provida em [23], foi obtida por meio de mapeamentos sistemáticos e análises qualitativas. Os dados extraídos foram estruturados e classificados. A classificação desses dados forneceu uma taxonomia que diferencia e analisa os modelos baseados no *POS* sob várias perspectivas: características comuns entre os modelos, dependências necessárias à elaboração dos modelos, características do software sob teste usadas pelos modelos, dentre outros.

Constatou-se, por meio dos resultados obtidos em [23], que uma variedade de modelos baseados no *POS* podem ser construídos, sendo a definição e complexidade do modelo dependente do contexto de aplicação do software sob teste. Embora as classificações e a taxonomia desenvolvidas proveem uma compreensão e perspectivas sobre o uso do *POS* [23], por abordar especificamente os modelos de representação aplicados para a geração de casos de teste, o estudo não provê informações detalhadas sobre a obtenção do *POS*. O uso do *POS* associado a outras finalidades também não foi explorado em [23]. A *RSL* descrita neste trabalho investiga o uso do *POS* em situações de uso específicas, fornecendo, por meio da abordagem investigativa adotada, informações não providas pelo estudo realizado por Smidts et al. [23].

4 Metodologia

Foram definidas 4 questões de pesquisa cuja consecução das respostas proveem a investigação sobre o uso do *POS* nas situações descritas na seção anterior. A Tabela 1 apresenta as questões de pesquisa que foram definidas, aplicáveis às situações de uso do *POS* investigadas.

Tabela 1. Questões de pesquisa

Q_i	Descrição
Q_1	Como o *POS* foi usado nas técnicas identificadas nos estudos selecionados?
Q_2	Como o *POS* foi obtido pelas técnicas identificadas?
Q_3	Como o *POS* foi caracterizado pelas técnicas identificadas?
Q_4	Relacionado ao uso do *POS*, quais foram as dificuldades reportadas pelos estudos selecionados?

Para obter respostas às questões de pesquisa, a investigação foi feita por meio de uma *RSL* sucedida pela técnica *snowballing*(*backward* e *forward*). Para a realização dessas atividades, a metodologia de pesquisa proposta em [25] foi instanciada. A Figura 1 ilustra essa metodologia.

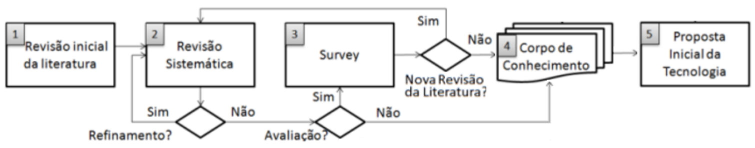


Figura 1. Representação da metodologia adotada. Adaptado de [25]

A etapa inicial (1), denominada "Revisão inicial da literatura", consiste em coletar informações básicas a respeito do tema a ser pesquisado, necessárias para a elaboração do protocolo da *RSL*. A etapa seguinte (2) compreende o planejamento e execução da *RSL* a qual, dependendo da análise dos resultados obtidos, pode ser refinada. Após a realização da *RSL*, opcionalmente, em decorrência de possíveis questionamentos que podem surgir sobre o conhecimento obtido, um *survey* (*Etapa 3*) pode ser realizado, o qual consiste em submeter o conhecimento adquirido à uma avaliação de especialistas. As duas etapas finais da metodologia (4 e 5) compreendem, respectivamente, à estruturação e consolidação do conhecimento adquirido, tornando-se a base para a idealização de novas tecnologias.

Para este trabalho, a *Etapa 1* usou os resultados do *MSL* citado na Seção 2. A *Etapa 2* refere-se à execução da *RSL* seguida pela execução da técnica *snowballing*. Referente à *Etapa 3*, o conhecimento adquirido na *Etapa 2*, realizada pelo primeiro autor, foi submetido a uma avaliação de outros pesquisadores, demais autores deste trabalho. Na *Etapa 4*, os dados obtidos pela *RSL* foram estruturados, permitindo obter respostas às questões de pesquisa e, assim, prover as contribuições esperadas para este trabalho. Essas contribuições servem como dados de entrada para a *Etapa 5*, a qual não foi abordada neste trabalho.

A *RSL* foi conduzida por um processo composto por três fases [18]: Planejamento, Condução e Publicação. O paradigma *GQM* (*Goal, Question, Metric*) foi adotado como referência para o planejamento da *RSL*. A ferramenta computacional *Start*, cuja finalidade, é prover apoio, ao pesquisador, na realização de revisões sistemáticas da literatura [9], foi usada para dar suporte à realização da *RSL*.

5 Planejamento

A *RSL* foi guiada por um protocolo no qual as diretrizes da pesquisa foram definidas. As subseções seguintes apresentam uma síntese desse protocolo.

5.1 Questões de pesquisa

O paradigma *GQM* foi adaptado para, a partir dos propósitos da pesquisa, estruturar as questões de pesquisa, identificando os dados a serem extraídos dos estudos selecionados para prover as repostas às respectivas questões. A Figura 2 ilustra, na estrutura do paradigma *GQM*, o objetivo, as questões de pesquisa e os dados a serem obtidos.

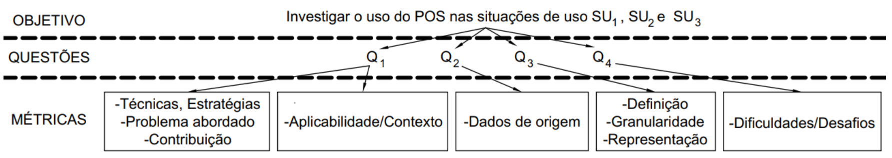

OBJETIVO
Investigar o uso do POS nas situações de uso SU1 , SU2 e SU3
QUESTÕES
Q1
Q2
Q3
Q4
MÉTRICAS
-Técnicas, Estratégias -Problema abordado -Contribuição
-Aplicabilidade/Contexto
-Dados de origem
-Definição -Granularidade -Representação
-Dificuldades/Desafios

Figura 2. GQM referente ao planejamento da *RSL*

Conforme ilustrado pela Figura 2, o objetivo da *RSL* corresponde ao *Goal*, as questões de pesquisa correspondem ao *Question* e os dados a serem extraídos, dos estudos selecionados, correspondem ao *Metric*.

5.2 Grupo de controle e critérios de seleção de estudos

Foram considerados, como grupo de controle, os estudos [2] e [16]. O estudo [2] evidencia, por meio dos resultados apresentados, que informações sobre como o software é operado por seus usuários são relevantes ao processo de desenvolvimento de software. O estudo [16] refere-se a abordagem de Musa sobre o *POS*, a qual é uma das mais relevantes abordagens para a construção do *POS*. A ocorrência desses estudos, como grupo de controle, foi verificada no conjunto de estudos identificados pela *RSL* e, também, nas respectivas referências bibliográficas desses estudos. A Tabela 2 apresenta os critérios de inclusão (I) e exclusão (E) usados para a seleção dos estudos.

Tabela 2. Critérios de inclusão e exclusão

	Critério
I	O estudo usa o *POS* em teste de regressão
I	O estudo usa o *POS* como critério de priorização
I	O estudo foca no *POS* e suas características
I	O estudo usa o *POS* em técnica/estratégia para avaliar casos de teste
E	O estudo não usa o *POS* em teste de regressão
E	O estudo não aborda o teste de regressão
E	O estudo não aborda o *POS* e suas características
E	O estudo não usa o *POS* como critério de priorização
E	O estudo aborda o *POS* porém o *POS* não é usado em técnica/estratégia para avaliar casos de teste
E	O estudo aborda a avaliação de casos de teste porém não usa o *POS*

5.3 *String* de busca

As *keywords* iniciais, relacionadas ao *POS*, foram obtidas pelo *MSL* citado na Seção 2, definindo uma *string* de busca inicial. A *string* de busca inicial foi usada em um estudo piloto para identificar novas *keywords*. Os estudos, referentes ao resultado retornado pelo estudo piloto, foram registrados na ferramenta computacional *Start*. Dentre suas funcionalidades, a *Start* fornece uma lista das *keywords* detectadas nos estudos registrados, classificando-as como "usadas"ou "não usadas"na *string* de busca, e ordenando-as pela frequência com que cada uma é empregada nos estudos registrados [9]. As *keywords* consideradas relevantes foram identificadas e incorporadas à *string* de busca inicial, dando origem a uma nova versão da *string* de busca, iniciando-se, assim, uma iteratividade na execução do estudo piloto. Essa iteratividade foi processada enquanto novas *keywords* eram identificadas.

Após a definição das *keywords* relacionadas ao *POS* (S_{pos}), novas *keywords* para tratar a especificidade de cada situação de uso do *POS* investigada foram adicionadas. As *strings* de busca, relacionadas às respectivas situações de uso do *POS* a serem investigadas ($SU_{1,2,3}$), foram construídas a partir das *keywords* e do uso dos operadores lógicos "*and*" ($\wedge$) e "*or*" ($\vee$). As *strings* de busca são mostradas na Tabela 3.

Tabela 3. *Strings* de busca

String de Busca	SU
(”operational profile” $\vee$ ”usage model” $\vee$ ”usage profile” $\vee$ ”user profile” $\vee$ ”users profile” $\vee$ ”profile of use” $\vee$ ”usage testing” $\vee$ ”input distribution” $\vee$ ”usage distribution” $\vee$ ”operational distribution” $\vee$ ”usage chain”)	S_{pos}
(”regression”) $\wedge$ (“software”) $\wedge$ (”testing” $\vee$ ”test”) $\wedge$ S_{pos}	SU_1
(”prioritization” $\vee$ ”prioritize”) $\wedge$ S_{pos}	SU_2
(”effectiveness” $\vee$ ”efficiency” $\vee$ ”evaluation”) $\wedge$ S_{pos}	SU_3

6 Condução

As fontes de informação selecionadas foram *Scopus*, *ACM Digital Library*, *IEEE Xplore*, *SpringerLink*, *ScienceDirect*, *Engineering Village* e *Web of Science*. A identificação dos estudos foi feita por meio de busca automática nas fontes de informação usadas, identificando, para as situações de uso investigadas SU_1, SU_2 e SU_3, respectivamente, 130, 86 e 214 estudos. A identificação e seleção de estudos foram realizadas separadamente, não havendo dependência entre as situações de uso investigadas. Na fase de seleção, dos estudos identificados, foram selecionados os estudos em que no título, resumo ou palavras-chaves, foram identificados ao menos um dos critérios de inclusão definidos. Na fase de extração, foram mantidos os estudos que, após a leitura completa e análise desses estudos, mantiverem ao menos um dos critérios de inclusão atribuídos. Os critérios de inclusão sobrepõe os critérios de exclusão. Do total de estudos identificados, 9 estudos foram selecionados para a fase de extração de dados. Esses estudos foram submetidos a técnica *snowballing*, identificando 214 novos estudos para a situação de uso SU_1. Para as situações de uso SU_2 e SU_3, a técnica *snowballing* identificou, respectivamente 46 e 184 novos estudos. Dos 444 novos estudos identificados pela técnica *snowballing*, 2 estudos foram adicionados à fase de extração de dados, totalizando, para a extração de dados, 11 estudos. A Figura 3 mostra a quantidade de estudos processados em cada etapa da *RSL*.

7 Qualidade dos Estudos Selecionados

A cada estudo selecionado na *RSL*, um indicador, cujo valor varia entre 0 e 1, foi atribuído visando quantificar a relevância do estudo aos propósitos deste trabalho. O indicador corresponde ao valor da média aritmética dos valores atribuídos às questões de avaliação para cada estudo selecionado. Quanto maior o valor do indicador, maior é a relevância do estudo com base na avaliação da qualidade do estudo. Dada a abordagem investigativa e exploratória deste trabalho, não foi definido um valor mínimo a ser obtido pelos estudos selecionados. A Tabela 4 apresenta as questões avaliativas, a motivação que deu origem a questão e os valores que podem ser atribuídos à questão.

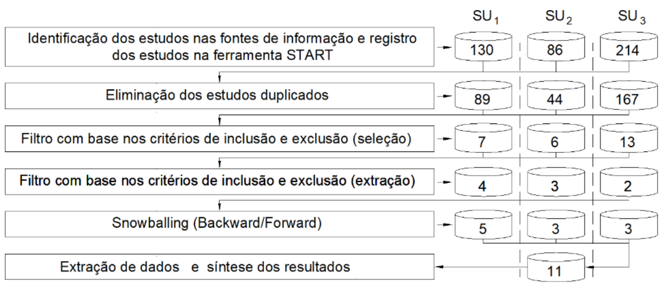

SU1
SU2
SU3
Identificação dos estudos nas fontes de informação e registro dos estudos na ferramenta START
130
86
214
Eliminação dos estudos duplicados
89
44
167
Filtro com base nos critérios de inclusão e exclusão (seleção)
7
6
13
Filtro com base nos critérios de inclusão e exclusão (extração)
4
3
2
Snowballing (Backward/Forward)
5
3
3
Extração de dados e síntese dos resultados
11

Figura 3. Quantitativo de estudos processados

Tabela 4. Questões avaliativas da qualidade dos estudos

Questão	**Motivação**	**Valor**
1-Os estudos selecionados citam estudos publicados pelos autores dos estudos de controle?	Citações de estudos publicados pelos autores dos estudos de controle são indícios de que o estudo selecionado tenha maior relevância em relação aos demais estudos	Entre 0 e 1 (conforme a quantidade de autores, dos estudos de controle, citados)
2-Qual o número de citações, em outros estudos, do estudo selecionado?	O número de citações do estudo selecionado evidencia a abrangência do estudo em contextos relacionados ao abordado neste trabalho	Entre 0 e 1 (conforme o número de citações do estudo selecionado)
3-O estudo selecionado tem o POS como foco principal da pesquisa ou relata o uso do perfil operacional?	Estudos cuja abordagem relata o uso do *POS* estão diretamente relacionados aos propósitos deste trabalho	0 ou 1 conforme a abordagem dada ao *POS* (Relata o uso do *POS*: 1; Foco nos conceitos do *POS*: 0,5)

8 Resultados Obtidos

A *RSL*, sucedida pela técnica *snowballing*, resultou em 11 estudos selecionados para a fase de extração de dados. A Tabela 5 apresenta esses estudos associados às situações de uso investigadas ($SU_{1,2,3}$).

Tabela 5. Estudos selecionados para a extração de dados

SU	Extração de dados
SU_1	Whittaker et al. [27], Takagi et al. [26], Kashyap [11], Sarwar et al. [22], Rogstad and Briand [21]
SU_2	Marijan et al. [13], Miranskyy et al. [14], Nakornburi and Suwannasart [19]
SU_3	Amrita and Yadav [1], Bertolino et al. [3], Chen et al. [7], Takagi et al. [26]

Observa-se que o estudo [26] foi identificado e selecionado nas investigações das situações de uso SU_1 e SU_3. As subseções seguintes apresentam, para cada situação de uso investigada, uma síntese desses estudos e, também, uma caracterização do uso do *POS* obtida por meio dos dados extraídos.

8.1 Uso do *POS* em técnicas de teste de regressão (SU_1)

Os estudos [22] e [21] abordam a seleção e priorização de casos de teste. Para Sarwar et al. [22], a execução de todos os planos de teste (manuais e automatizados) é uma atividade cansativa que consome tempo, custo e pode não ser efetiva na descoberta, antecipadamente, de defeitos críticos. Assim, é necessário selecionar poucos, porém relevantes, casos de teste a partir de uma massa de teste. Para isso, é proposta uma técnica para priorizar casos de teste aplicada a processos de desenvolvimento de software gerenciados pelo método organizacional *Scrum*. A técnica representa o *POS* por meio da frequência com a qual os casos de teste são executados durante o teste de regressão. Os casos de teste são priorizados por meio de um indicador quantitativo atribuído a cada caso de teste. O indicador é calculado considerando-se a frequência de execução do caso de teste, o número de falhas detectadas pelo caso de teste e a criticidade atribuída ao caso de teste.

Rogstad and Briand [21] realizaram uma investigação empírica sobre estratégias para a seleção e geração de dados de teste para técnicas de teste combinatório em nível de teste de regressão. A investigação foi realizada em um ambiente de desenvolvimento de uma aplicação persistente real e representativa. Essa investigação foi motivada pela demanda de tempo e esforço dos testadores decorrentes do uso de dados obtidos em um ambiente de produção do software (dados reais) nas atividades de teste. O uso de dados obtidos em ambiente de produção é uma estratégia comum em atividades de teste, porém, esses dados nem sempre satisfazem os requisitos do plano de teste, forçando os testadores a manipularem ou estenderem os dados de produção para torna-los adequados. O *POS* do software é usado adicionalmente em uma das técnicas de teste aplicadas na investigação, gerando um conjunto de testes alinhado com o uso do software.

Em decorrência dos problemas inerentes às tecnologias existentes, tais como a demanda de esforço e uso de técnicas *ad-hoc*, os quais geram uma alta quantidade de caminhos e cenários de teste que não refletem o uso real do software, Kashyap [11] apresenta uma abordagem para criação de modelo de comportamento que permite a geração e priorização automática dos casos de teste com base em dados de uso do software. Nessa abordagem, é criado um modelo de comportamento que permite a geração e priorização automática dos casos de teste com base em dados de uso do software.

Para Takagi et al. [26], métodos existentes baseados em teste de regressão não são apropriados para avaliar a confiabilidade do software. Takagi et al. [26] apresentam uma técnica que avalia a confiabilidade do software como parte do processo de manutenção, estimando e prevendo falhas que influenciarão na confiabilidade das próximas versões do software. A técnica gera um modelo de teste a partir do *POS*. Os resultados obtidos pela execução dos testes são mapeados no modelo de teste criado. Quando uma falha é detectada, um estado, que representa a falha, é inserido no modelo de teste e associada aos estados que representam as operações nas quais a falha foi detectada. O modelo de teste é usado para determinar o critério de parada dos testes, podendo ser determinado pela cobertura dos estados ou transições que compõe o modelo de teste, pelo número de transições incidentes nos estados de falha ou pelo número de falhas detectadas e mapeadas no modelo de teste.

Segundo Whittaker et al. [27], as estimativas de confiabilidade são obtidas por meio de modelos de probabilidade construídos a partir de dados coletados durante a realização dos testes e, assim, situações de falha são tratadas assumindo-se que obedecem uma regra de probabilidade, permitindo, a partir disso, estimar e prever futuras ocorrências de falhas. Essa abordagem fornece uma análise precisa quando a distribuição probabilística tem suporte empírico e os testes cobriram grande parte do software, porém, quando não há suporte empírico e os recursos de teste são escassos, outras abordagens são necessárias [27]. Whittaker et al. [27] propõem uma estratégia para a detecção de defeitos na manutenção do software. A estratégia proposta combina dados e resultados de testes passados com dados atuais em um único modelo de uso. Com base no modelo de uso, padrões de ocorrência de falhas são identificados para prever os resultados dos testes na próxima compilação do software.

8.2 Uso do *POS* como um critério de priorização (SU_2)

Segundo Marijan et al. [13], à medida que a complexidade de um software aumenta, a qualidade do sistema é submetida a um potencial decréscimo. Para tratar essa situação, é proposta uma metodologia para verificação automática de sistemas multimídia compreendendo a modelagem do comportamento do sistema, a geração de casos de teste baseados no modelo criado e a execução automática dos casos de teste gerados [13]. A modelagem do *POS* é baseada na especificação do sistema e em dados coletados durante a operação do sistema pelos usuários. As sequencias de transições do modelo uso dão origem aos casos de teste a serem executados. Os resultados dos testes são analisados para avaliar a confiabilidade do software.

Os dados de uso coletados dos clientes fornecem informações quantitativas valiosas sobre os padrões de uso do software. Para Miranskyy et al. [14], reunir esses perfis a partir de um grande conjunto de clientes demanda tempo e recursos. Assim, Miranskyy et al. [14] propõem uma técnica para a seleção e priorização de um conjunto mínimo de clientes a serem considerados na elaboração do *POS*, tendo, como objetivo, prover, na execução dos testes, uma cobertura para determinados conjuntos de defeitos. A técnica proposta, inicialmente, classifica e prioriza os clientes com base no histórico e tipos defeitos reportados por eles. Após a classificação e priorização dos clientes por meio de uma análise qualitativa, a técnica seleciona, para a criação do *POS*, um conjunto mínimo de clientes que tenham descoberto o maior número de defeitos. A seleção manual desse conjunto pode ser inviável. Assim, para se obter o conjunto mínimo de clientes, a priorização dos clientes, feita de maneira qualitativa, é transformada em quantitativa adotando-se o algoritmo de programação inteira binária.

Nakornburi and Suwannasart [19] relatam o desenvolvimento de uma ferramenta computacional para gerar casos de teste usando técnicas de teste combinatório e o *POS*. O *POS* é usado para priorizar e selecionar um conjunto ideal de parâmetros de entrada e os respectivos valores para os casos de teste gerados pela ferramenta proposta.

8.3 Uso do *POS* como um critério de avaliação (SU_3)

Segundo Amrita and Yadav [1], muitos pesquisadores abordam a seleção de casos de teste com base no *POS*, porém, pouco é discutido sobre as partes infrequentes. Amrita and Yadav [1] propõem um modelo que fornece a flexibilidade de alocar casos de teste conforme a prioridade definida pelo *POS* e, também, pela experiência do testador. Assim, o modelo proposto considera tanto a probabilidade de ocorrência quanto a severidade da operação. Com base nessas informações, usando lógica *fuzzy*, o modelo propõe a seleção de casos de teste. O modelo usa a criticidade atribuída às operações do software e o *POS* como variáveis de entrada para um sistema de inferências *fuzzy*. Assim, as operações do software são classificadas em: operação frequente e crítica; operação frequente e não crítica; operação infrequente e crítica; operação infrequente e não crítica. O valor da variável de saída, a ser obtido pelo sistema de inferências *fuzzy*, corresponde ao número de casos de teste a serem alocados para a operação. Fornecidas as variáveis de entrada e definidas as regras *fuzzy*, é obtido um conjunto *fuzzy* resultante que, após a geração, com base nas variações adotadas, é *desfuziado* para se obter os valores equivalentes e, consequentemente, a quantidade sugerida de casos de teste a serem alocados em cada operação.

Segundo Bertolino et al. [3], o teste baseado no *POS*, por focar nas falhas com maior probabilidade de ocorrência, pode sofrer saturação e, assim, perder eficácia na detecção de falhas. Nesse contexto, uma técnica de teste de software adaptativa e iterativa, baseada no *POS*, denominada *covrel*, é proposta por Bertolino et al. [3]. A técnica proposta é executada iterativamente. Na primeira iteração, a técnica usa o *POS* para selecionar e executar os casos de teste randomicamente, com base na probabilidade de ocorrência das partições do domínio de entrada do programa. Nas iterações seguintes, a cada iteração, a técnica calcula, para cada partição: (*i*) a quantidade de casos de teste ideal a serem selecionados; (*ii*) seleciona, prioriza e executa a quantidade de casos de teste calculada. Para determinar a quantidade ideal de casos de teste, uma probabilidade é obtida para representar o quanto o teste da partição contribui com a confiabilidade do programa. A seleção e priorização dos casos de teste é feita com base na frequência com que as partes do programa são exercitadas durante a execução dos testes nas iterações anteriores. Como o foco da abordagem é selecionar casos de teste que cobrem partes do programa que são pouco exercitadas, casos de teste associados às partes do programa pouco exercitadas têm maior prioridade.

Quando um conjunto de testes não detecta novas falhas, o uso desses testes, ou similares, pode superestimar a confiabilidade do software. Assim, quanto mais redundantes forem os casos de teste em relação ao código coberto, mais superestimada será a confiabilidade do software. Nesse contexto, Chen et al. [7] propõem uma técnica que determina uma taxa de compressão que ajusta o intervalo de tempo entre as falhas quando são detectados casos de teste redundantes durante a execução do conjunto de teste. Para obter a taxa de compressão são considerados o tempo de execução do conjunto de teste, a cobertura de código provida e, também, a detecção de casos de testes redundantes no conjunto de teste.

8.4 Caracterização do uso do *POS* nos estudos selecionados

A Tabela 6 mostra, a partir dos dados extraídos dos estudos selecionados (Tabela 5), uma caracterização do uso do *POS* nesses estudos.

Tabela 6. Caracterização do POS nas situações de uso investigadas

	Granularidade adotada para o POS				Origem do POS						Representação do POS			Aplicabilidade						Atividade fim que usou o POS				Artefato	
Referência	Dados de Entrada	Operações	Casos de teste	Clientes	Arquivos de *log*	Projetos anteriores	Histórico de uso do software	Conjunto de teste	Observação dos usuários	Não especificado	Registro de frequência de uso da granularidade adotada	Modelo de uso contendo distribuição probabilística	Não se aplica	Teste combinatório	Teste de regressão	Teste randômico	Teste baseado em modelo	Teste de confiabilidade	Não especificado	Geração	Seleção	Priorização	Estimativa	Casos de teste	Dados do cliente
[19]	X				X						X			X						X				X	
[21]	X				X							X		X	X					X				X	
[26]		X			X							X			X	X				X				X	
[11]		X								X		X							X	X		X		X	
[22]			X					X			X				X							X		X	
[13]		X				X			X			X					X			X				X	
[27]		X					X					X			X								X	X	
[14]				X						X			X						X		X				X
[3]	X									X		X					X				X	X		X	
[1]		X								X		X					X					X		X	
[7]	X									X		X		X				X			X	X		X	

Por meio da Tabela 6, é possível visualizar a granularidade do *POS*, as fontes de dados usadas para obter o *POS*, a maneira como o *POS* foi representado nos estudos selecionados, a atividade na qual o *POS* foi usado e os artefatos abordados pelas atividades nas quais o *POS* foi usado. A granularidade, apresentada na Tabela 6, no contexto deste trabalho, corresponde ao nível de fragmentação, conceitual ou estrutural, com a qual o software é abordado para que uma distribuição probabilística seja atribuída aos fragmentos gerados.

9 Análise dos Resultados

A análise dos resultados obtidos permitiu responder às questões de pesquisa que foram definidas na fase de planejamento da *RSL*. As questões de pesquisa com as respectivas respostas são apresentadas seguir.

(a) **Como o *POS* foi usado nas técnicas identificadas nos estudos selecionados em cada situação de uso do *POS* investigada?**

– *Uso do POS em técnicas de teste de regressão (SU$_1$)*: Dado que modelos de uso do software são a base do teste estatístico, as abordagens baseadas em teste estatístico são geralmente referenciadas como teste baseado em uso [11]. Assim, verificou-se, nos estudos aceitos, que o uso do *POS* é associado, geralmente, à técnicas de teste combinatório e estatístico, as quais são empregadas nas estratégias de teste de confiabilidade e de regressão. Esse cenário é entendível dado que o teste estatístico tem, como um de seus objetivos, avaliar a confiabilidade do software e o teste de regressão tem, em sua essência, validar modificações ocorridas no software, as quais podem afetar a confiabilidade do software. Em ambas as situações, dados sobre a maneira com a qual o software é operado por seus usuários são relevantes e podem ser coletados durante a operação do software. Dos 344 estudos identificados e processados para a situação de uso do *POS* SU$_1$, 5 estudos foram selecionados para a fase de extração: [21,26,22,27,11]. Esses estudos abordaram o uso do *POS* em sistemáticas para gerar e priorizar casos de teste.

– *Uso POS como critério de priorização (SU$_2$)*: Dos 132 estudos identificados e processados para a referida categoria de uso, 3 estudos foram selecionados para a fase de extração: [19,13,14]. Desses estudos, 2 estudos usaram o *POS* em sistemáticas para geração e priorização de casos de teste. Um estudo usou o *POS* em uma sistemática para selecionar e priorizar de um conjunto mínimo de clientes para a obtenção de perfis operacionais específicos para cobrir determinados conjuntos de defeitos.

– *Uso do POS como critério de avaliação de casos de teste (SU$_3$)*: Dos 398 estudos identificados e processados para a referida categoria de uso, 3 estudos foram selecionados para a fase de extração: [3,1,7]. Esses estudos usaram o *POS* em técnicas para avaliar os casos de teste tendo, como propósito, selecionar e priorizar casos de teste relevantes às operações a serem testadas. Desses estudos, os estudos [3,7] avaliaram os casos de teste por meio da cobertura provida por eles, sendo que, em um desses estudos, a cobertura era avaliada em relação ao *POS* e, no outro estudo, o cobertura dos casos de teste era usada como um critério de comparação para identificar casos de teste redundantes. O estudo [1] avaliou os casos de teste por meio de critérios definidos na técnica apresentada, tais como criticidade da operação testada, frequência de execução dos casos de teste e número de falhas descobertas.

Constatou-se, assim, que as técnicas identificadas trataram a obtenção e o uso do *POS* como "atividades-meio". Entende-se por "atividade-meio" uma atividade cuja execução é necessária para que a técnica proposta pelo estudo possa ser realizada ("atividade-fim"). Por ser uma "atividade-meio", poucos detalhes foram fornecidos quanto a obtenção e representação do *POS*. Verifica-se, também, pela Tabela 6, que o uso *POS* tem maior evidência nos estudos cujas técnicas apresentadas tratam a seleção e priorização de casos de teste.

(b) **Como o *POS* foi obtido pelas técnicas identificadas?**
Dados obtidos a partir de arquivos de *log*, gerados durante a operação do software, foi a estratégia com maior evidência dentre os estudos selecionados. O uso dessa estratégia foi constato nos estudos [19,21,26,13]. Informações providas pela expectativa dos desenvolvedores, por softwares similares, pela experiência em projetos anteriores e pela observação do comportamento dos usuários podem, também, contribuir para se obter o *POS* [26,13]. [26] considera, também, na inexistência de outras fontes de informação, uma distribuição probabilística uniforme para todas as transições contidas no modelo de uso que representa o *POS*. O uso da especificação dos software para a estruturação do modelo de uso que representa o *POS* é considerado em [13].

(c) **Como o *POS* é caracterizado pelas técnicas identificadas?**
Os dados extraídos dos estudos selecionados, listados na Tabela 5, permitiram caracterizar o uso do *POS* quanto à granularidade adotada para a concepção do *POS*, origem dos dados que permitiram a concepção do *POS*, a maneira como o *POS* é representado, a estratégia e técnica de teste favorecidas pelo uso do *POS* e a relação entre o uso do *POS* e a abordagem provida pelos estudo aos casos de teste (Tabela 6). Constatou-se que a granularidade adotada para se obter o *POS* não se restringe à dados de entrada, operações e os estados que o software pode assumir. Em [26,13], além das operações do software, os estados de falha assumidos pelo software também foram considerados como um nível de granularidade para a elaboração do *POS*. O *POS*, abordado em [27], foi obtido a partir da frequência de execução e quantidade de falhas descobertas pelos casos de teste do software.

(d) **Relacionado à obtenção e uso do *POS*, quais foram as dificuldades e problemas reportados pelos estudos selecionados?**
Como foi descrito, a identificação do *POS* nas técnicas identificadas é abordada como uma "atividade-meio", não fornecendo detalhes sobre a maneira como o *POS* foi obtido. Apenas Bertolino et al. [3], por se tratar do problema abordado pela técnica apresentada, relata problemas relacionados ao uso do *POS*, porém, ressalta-se que o problema apresentado em [3] não havia sido identificado na revisão informal da literatura, a qual reportou problemas relacionados a: (*i*) demanda de esforço para se obter o *POS*[20,24];(*ii*) a dificuldade em prever o uso dos software [8]; (*iii*) modificações constantes do *POS*[12]; (*iv*) probabilidade de uso, a qual torna-se pequena quando o modelo de uso possui muitos estados que representam as operações [10].

10 Ameaças à Validade

Visando diminuir a subjetividade que poderia ocorrer na avaliação dos estudos durante a realização do *RSL*, os resultados parciais, periodicamente, obtidos por um pesquisador (primeiro autor), eram submetidos a outros pesquisadores (demais autores) para a realização de uma análise informal dos resultados obtidos. A qualidade dos estudos selecionados nessas atividades também foi avaliada, porém, dada a abordagem investigativa deste trabalho, não foi definido um valor mínimo a ser obtido pelos estudos selecionados. Os resultados apresentados fazem parte de uma tese de doutoramento com escopo mais amplo [5]. Assim, a *RSL* abordou as situações de uso do *POS* relacionadas ao escopo da tese de doutoramento citada, não abrangendo o uso do *POS* em situações diferentes das investigadas.

11 Conclusões

Este artigo apresentou os resultados obtidos por *RSL* que investigou o uso do *POS* sob três perspectivas: (*i*) em atividades associadas ao teste de regressão, (*ii*) como um critério de priorização aplicável ao processo de desenvolvimento de software e (*iii*) como um critério de avaliação para casos de teste existentes. A *RSL* está associada a uma pesquisa de doutoramento que investiga um possível descompasso entre o *POS* e o Perfil de Teste [6] e as contribuições que o *POS* pode proporcionar ao teste de software, almejando melhorar a qualidade do software do ponto de vista dos usuários. Essa pesquisa propõe, como contribuição à Engenharia de Software, especificamente à área de testes, uma estratégia que que permita (*i*) adequar, dinamicamente, um conjunto de testes existente ao *POS* e, também, (*ii*) usar o *POS* como um critério de priorização que permita, dado um conjunto de defeitos, identificar os defeitos que tem maior impacto na operação do software pelos usuários e, assim, considerar esse impacto, juntamente com outros critérios, na precificação dos defeitos.

Os resultados obtidos pela *RSL* proporcionaram respostas às questões de pesquisa que foram definidas, provendo uma base de conhecimento centralizada sobre o uso do *POS* nos contextos investigados e, assim, uma caracterização e o entendimento quanto ao *POS* nas situações de uso investigadas. Verifica-se que o uso do *POS* como um critério de priorização está associado à atividade de teste, sendo usado na priorização de casos de teste. O uso do *POS* como um critério de priorização associado à outros artefatos do processo de desenvolvimento de software não é explorado. Quanto ao uso do *POS* como um critério de avaliação, verificou-se que as técnicas identificadas abordam a geração, seleção e priorização dos casos de teste por meio da análise de cobertura. O uso do *POS* como um critério de avaliação para adequar um conjunto de teste existente, que não tenha sido criado inicialmente por meio do *POS*, também não foi explorado. Assume-se, assim, que as contribuições providas por este artigo são relevantes e fornecem informações que podem contribuir e favorecer os estudos associados à Engenharia de Software e, especificamente, à área de qualidade de software.

Referências

1. Amrita, Yadav, D.K.: A novel method for allocating software test cases. In: 3rd International Conference on Recent Trends in Computing 2015 (ICRTC-2015). vol. 57, pp. 131 – 138. Elsevier, Delhi, India (2015)
2. Begel, A., Zimmermann, T.: Analyze this! 145 questions for data scientists in software engineering. In: Proceedings of the 36th International Conference on Software Engineering. pp. 12–23. ICSE 2014, ACM, New York, NY, USA (2014)
3. Bertolino, A., Miranda, B., Pietrantuono, R., Russo, S.: Adaptive coverage and operational profile-based testing for reliability improvement. In: Proceedings of the 39th International Conference on Software Engineering. pp. 541–551. ICSE '17, IEEE, Piscataway, NJ, USA (2017)
4. Bittanti, S., Bolzern, P., Scattolini, R.: An introduction to software reliability modelling, chap. 12, pp. 43–67. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
5. Cavamura Jr, L.: Operational profile and software testing: Aligning user interest and test strategy. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 492–494 (April 2019)

6. Cavamura Jr, L., Morimotto, R., Vincenzi, A.R.M., Fabbri, S.: Operational profile vs. test profile (October 2019)
7. Chen, M.., Lyu, M.R., Wong, W.E.: Effect of code coverage on software reliability measurement. IEEE Transactions on Reliability **50**(2), 165–170 (June 2001)
8. Cukic, B., Bastani, F.B.: On reducing the sensitivity of software reliability to variations in the operational profile. pp. 45–54. IEEE, White Plains, NY, USA (1996)
9. Fabbri, S., Silva, C., Hernandes, E., Octaviano, F., Di Thommazo, A., Belgamo, A.: Improvements in the start tool to better support the systematic review process. In: Proc. of the 20th International Conference on Evaluation and Assessment in Software Engineering. EASE '16, ACM, New York, NY, USA (2016)
10. Fukutake, H., Xu, L., Takagi, T., Watanabe, R., Yaegashi, R.: The method to create test suite based on operational profiles for combination test of status. pp. 1–4. Institute of Electrical and Electronics Engineers Inc., White Plains, NY, USA (2015)
11. Kashyap, A.: A Markov Chain and Likelihood-Based Model Approach for Automated Test Case Generation, Validation and Prioritization: Theory and Application. Ph.D. thesis (2013)
12. Leung, Y.W.: Software reliability allocation under an uncertain operational profile. Journal of the Operational Research Society **48**(4), 401 – 411 (1997)
13. Marijan, D., Teslic, N., Tekcan, T., Pekovic, V.: Multimedia system verification through a usage model and a black test box. In: The 2010 International Conference on Computer Engineering Systems. pp. 178–182. IEEE, Pasadena, CA, United states (Nov 2010)
14. Miranskyy, A., Cialini, E., Godwin, D.: Selection of customers for operational and usage profiling. Providence, RI (2009)
15. Musa, J.D.: Software reliability measures applied to systems engineering. In: Managing Requirements Knowledge, International Workshop on(AFIPS). vol. 00, p. 941. IEEE, S.I. (12 1979)
16. Musa, J.: Operational profiles in software-reliability engineering. IEEE **10**(2), 14–32 (1993)
17. Musa, J.D., Ehrlich, W.: Advances in software reliability engineering. Advances in Computers, vol. 42, pp. 77 – 117. Elsevier (1996)
18. Nakagawa, E.Y., Scannavino, K.R.F., Fabbri, S., Ferrari, F.C.: Revisão Sistemática da Literatura em Engenharia de Software: Teoria e Prática. Elsevier, Rio de Janeiro, 1rd edn. (2017)
19. Nakornburi, S., Suwannasart, T.: A tool for constrained pairwise test case generation using statistical user profile based prioritization. pp. 1–6. Institute of Electrical and Electronics Engineers Inc., White Plains, NY, USA (2016)
20. Namba, Y., Akimoto, S., Takagi, T.: Overview of graphical operational profiles for generating test cases of gui software. pp. 1–3. Institute of Electrical and Electronics Engineers Inc., White Plains, NY, USA (2015)
21. Rogstad, E., Briand, L.: Cost-effective strategies for the regression testing of database applications: Case study and lessons learned. Journal of Systems and Software **113**, 257 – 274 (2016)
22. Sarwar, S., Mahmood, Y., Qayyum, Z.U., Shafi, I.: Test case prioritization for nunit based test plans in agile environment. In: Artificial Intelligence: Methodology, Systems, and Applications. pp. 246–253. AIMSA 2014, Cham (2014)
23. Smidts, C., Mutha, C., Rodríguez, M., Gerber, M.J.: Software testing with an operational profile: Op definition. ACM Comput. Surv. **46**(3), 39:1–39:39 (Feb 2014)
24. Sommerville, I.: Software Engineering. Addison-Wesley, Harlow, England, 9 edn. (2011)
25. Spínola, R.O., Dias-neto, A.C., Travassos, G.H.: Abordagem para desenvolver tecnologia de software com apoio de estudos secundários e primários. In: 5th Experimental Software Engineering Latin American Workshop. ICMC/USP, São Carlos, SP, Brasil (2008)
26. Takagi, T., Furukawa, Z., Yamasaki, T.: 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 (2007)

27. Whittaker, J., Rekab, K., Thomason, M.: A markov chain model for predicting the reliability of multi-build software. Information and Software Technology **42**(12), 889 – 894 (2000)
28. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (Mar 2012)

Capítulo 5

Perfil operacional de software versus perfil de teste: em direção a uma melhor estratégia de teste de software

*Artigo publicado no "Journal of Software Engineering Reserach and Development" (JSERD). Esse artigo é uma extensão de CAVAMURA et al. (2019). CAVAMURA et al. (2019) foi publicado no XVIII Simpósio Brasileiro de Qualidade de Software (SBQS 2019), sendo premiado como o melhor artigo (*1º *colocado) da trilha principal.*

CAVAMURA, L. J. et al. Software operational profile vs. test profile. In: **Proceedings of the XVIII Brazilian Symposium on Software Quality**. New York, NY, USA: Association for Computing Machinery, 2019. (SBQS'19), p. 139–148. ISBN 9781450372824.

CAVAMURA, L. J. et al. **Software operational profile vs. test profile: Towards a better software testing strategy**. Journal of Software Engineering Research and Development, v. 8, p.5:1 – 5:17, Aug. 2020.

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

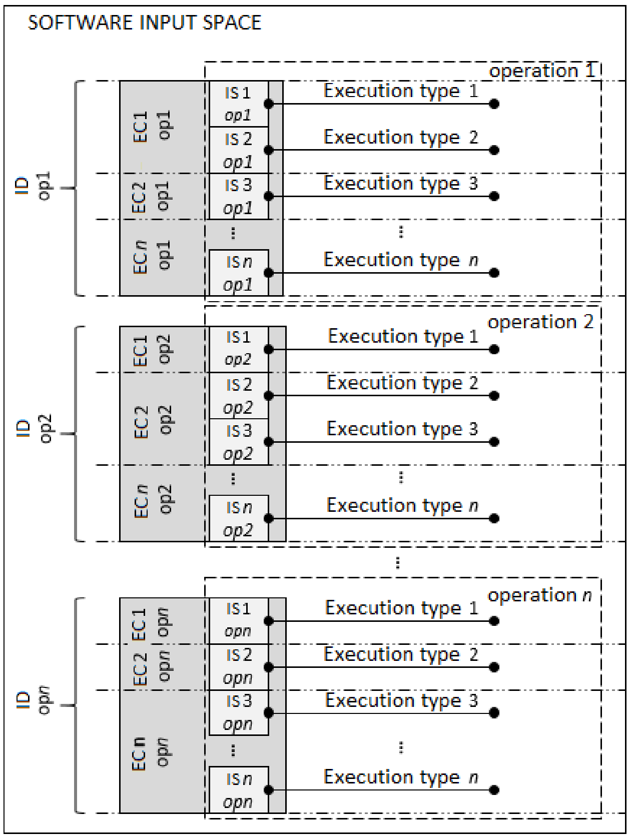


Figure 1. Concepts involved in the definition of the operational profile.

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.

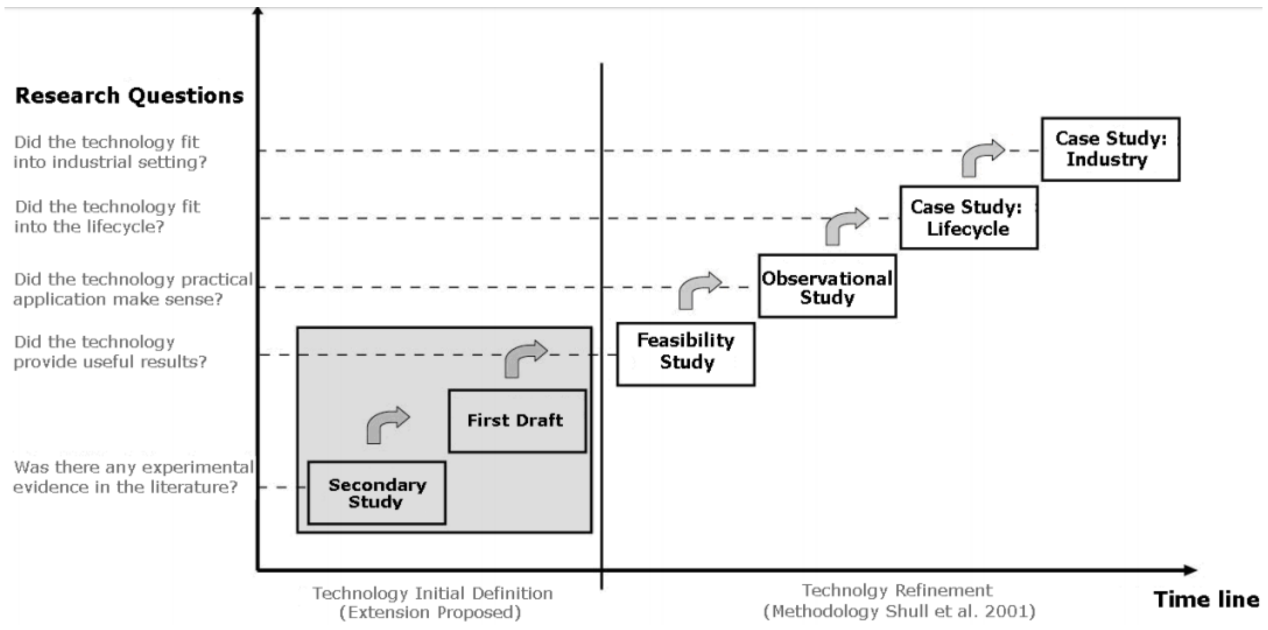

Research Questions
Did the technology fit into industrial setting?
Did the technology fit into the lifecycle?
Did the technology practical application make sense?
Did the technology provide useful results?
Was there any experimental evidence in the literature?
Case Study: Industry
Case Study: Lifecycle
Observational Study
Feasibility Study
First Draft
Secondary Study
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

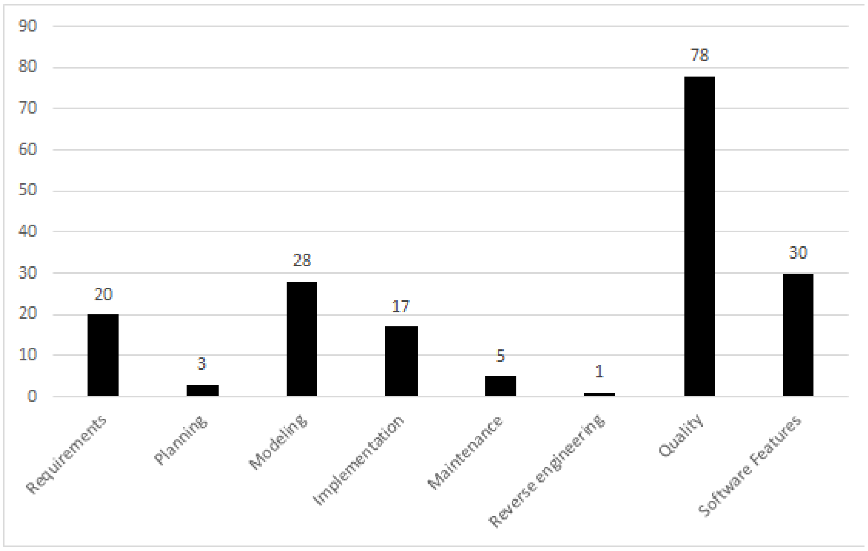


Figure 3. Distribution of the studies in software engineering areas.

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

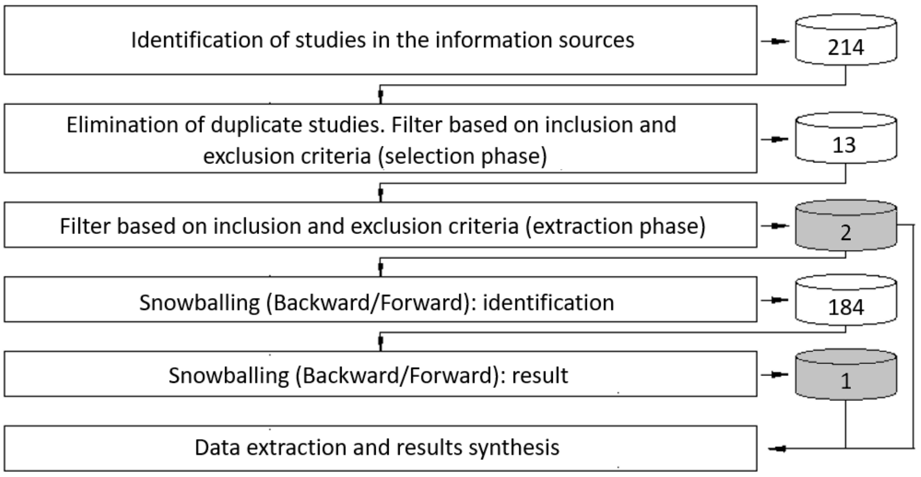

Identification of studies in the information sources
214
Elimination of duplicate studies. Filter based on inclusion and exclusion criteria (selection phase)
13
Filter based on inclusion and exclusion criteria (extraction phase)
2
Snowballing (Backward/Forward): identification
184
Snowballing (Backward/Forward):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

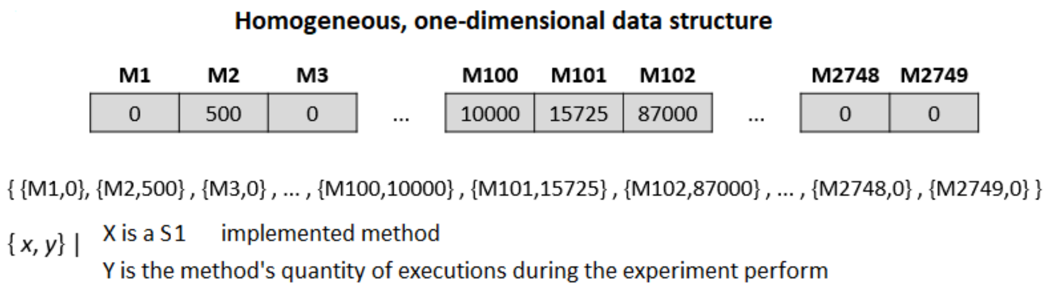
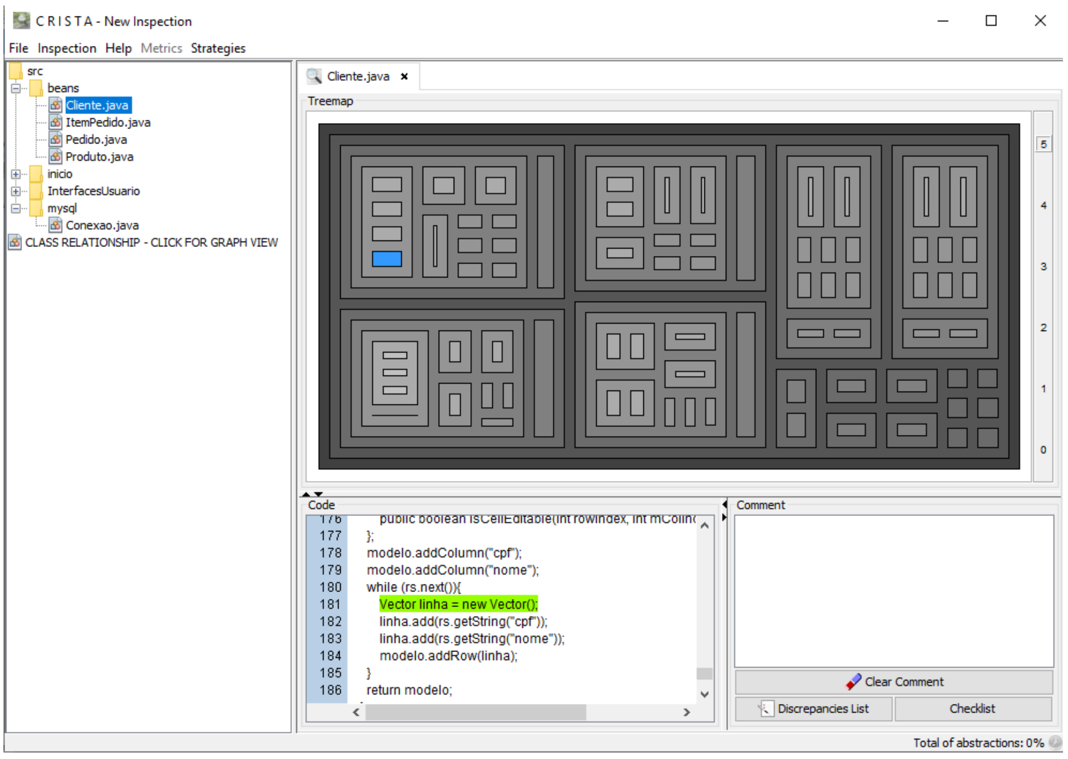

Homogeneous, one-dimensional data structure
M1 M2 M3 ... M100 M101 M102 ... M2748 M2749
0 500 0 ... 10000 15725 87000 ... 0 0
{ {M1,0}, {M2,500} , {M3,0} , ... , {M100,10000} , {M101,15725} , {M102,87000} , ... , {M2748,0} , {M2749,0} }
{ x, y} | X is a S1 implemented method
Y is the method's quantity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

Figure 5. Graphical representation of the data structure.



```
    public boolean isCellEditable(int rowIndex, int mColInd
};
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-

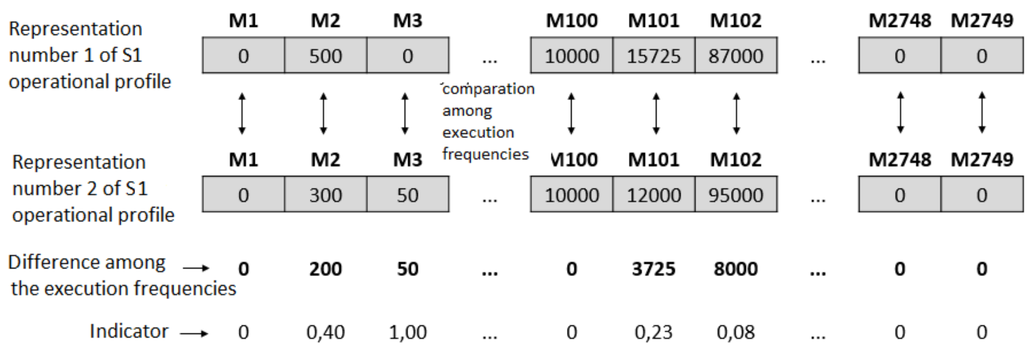

Representation number 1 of S1 operational profile
M1 M2 M3 M100 M101 M102 M2748 M2749
0 500 0 ... 10000 15725 87000 ... 0 0
comparation among execution frequencies
Representation number 2 of S1 operational profile
M1 M2 M3 M100 M101 M102 M2748 M2749
0 300 50 ... 10000 12000 95000 ... 0 0
Difference among 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.

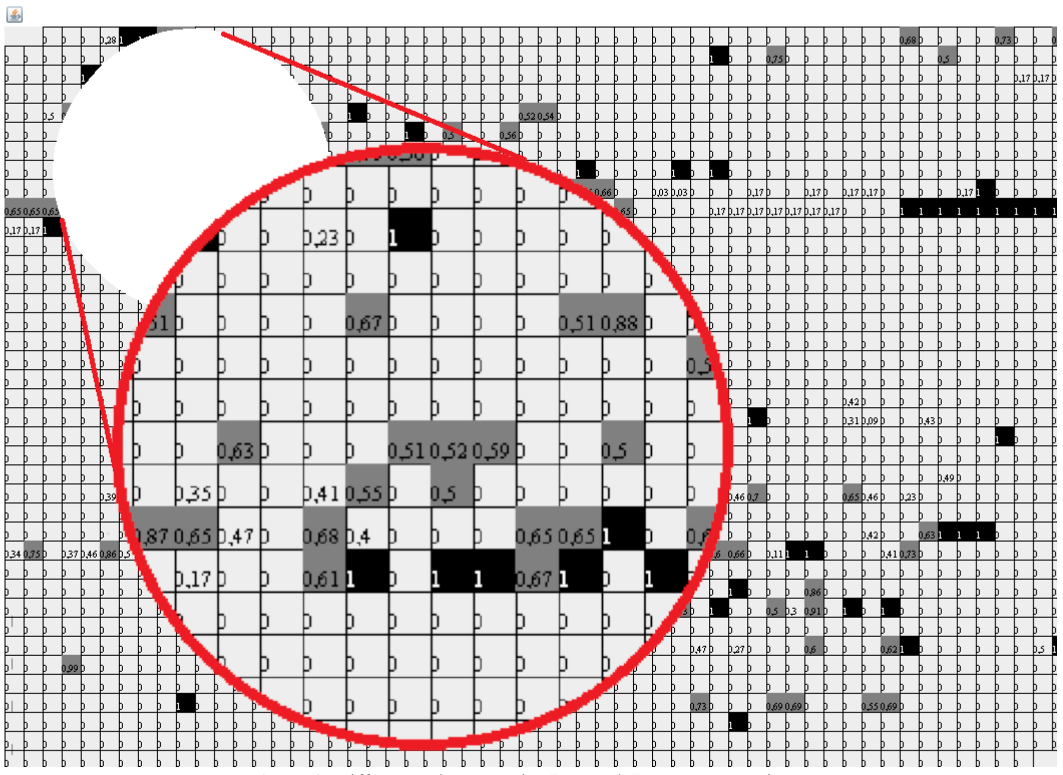


Figure 8. Differences between the $P12$ and $P13$ representations.

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 $EXS\text{–}AT_2$ 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.

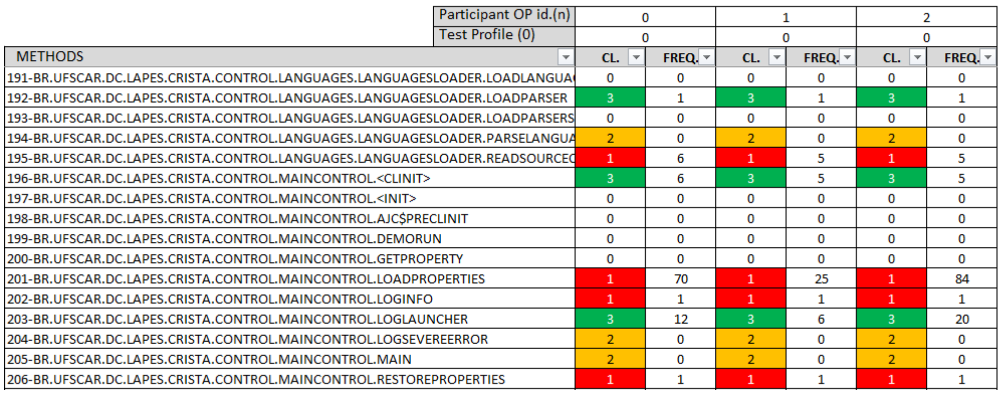
5.4 *EXS–AT*$_3$: Failures in untested *SOP* parts

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.

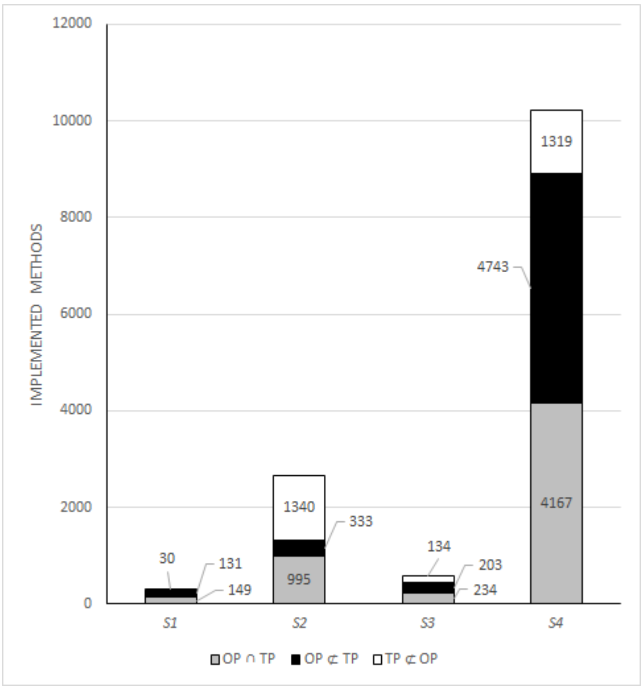


Figure 10. Results.

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 $EXS\text{–}AT_4$ 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 $EXS\text{–}AT_4$ activity, we selected S_2 and S_4 software. The reasons why we selected these software are because we used them in $EXS\text{–}AT_2$ and $EXS\text{–}AT_3$ 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 $EXS\text{–}AT_4$ 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}.

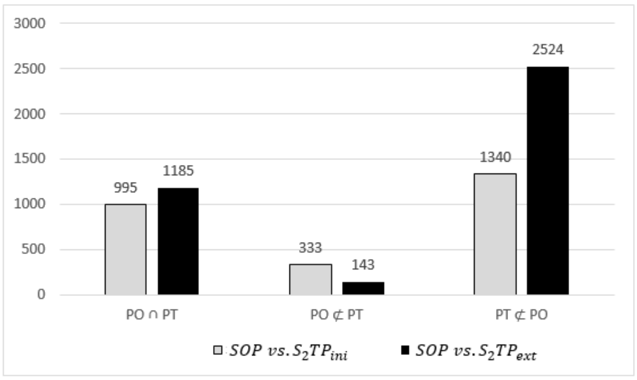


Figure 11. S_2TP_{ini} and S_2TP_{ext} results.

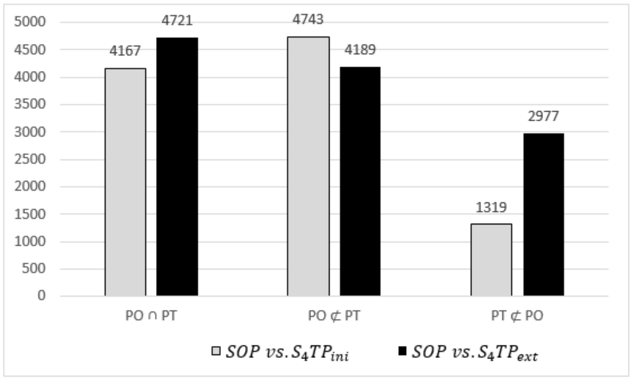


Figure 12. S_4TP_{ini} and S_4TP_{ext} results.

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 S_2TP_{ini}	*SOP* vs S_2TP_{ext}	(%)	*SOP* vs S_4TP_{ini}	*SOP* vs S_4TP_{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.

Capítulo 6

OPDaTe: Um gerador de casos de teste de unidade baseado em dados reais obtidos do perfil operacional do software

Artigo em fase de submissão.

CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. Opdate: A unit test case generator based on real data from software operating profile. xxx-xxxxxx, x, p. xxx-xxx(x), January 2022. ISSN X.

OPDaTe: A unit test case generator based on real data from Software Operating Profile*

Luiz Cavamura Júnior[1], Sandra C. P. F. Fabbri[1], and Auri M. R. Vincenzi[1]

[1]Computing Department, Federal University of São Carlos, Rod. Washington Luís, km 235 - Jardim Guanabara - PO Box 676 - Zip Code: 13565-905, São Carlos-SP, Brazil

[1]*{luiz_cavamura, sfabbri, auri}@ufscar.br*

February 11, 2022

Abstract

How users operate the software can affect the behavior of the software and thus lead to unexpected behavior. The Software Operational Profile (*SOP*) is a software specification representing how users operate the software, thus making it relevant to software testing. This paper introduces the *OPDaTe* tool that generates method-level executable unit test cases using data obtained dynamically from the *SOP*. In addition, we performed a feasibility study to promote an initial assessment of the effectiveness of *OPDaTe* about the following aspects: a) its ability to generate executable unit test cases using *SOP* data as test data; b) the code coverage provided by the executable test cases generated for the tested units; c) extend the software coverage under test and the units tested from test suites obtained from different software operational profiles. The feasibility study provided evidence that denotes a satisfactory result for the aspects evaluated. The feasibility study and its results are described in this paper.

Keywords— Software Quality, Software Testing, Operational Profile, Test Profile, Automated Test, Tool, OPDaTe

1 Introduction

SOP is a software specification based on the way users operate the software. Musa [1993] defines *SOP* as quantitative characterization of the software that allows identifying the most used parts of the software. The use of *SOP* in the software testing does not guarantee the detection of all faults, but it ensures that the most used software operations are tested [Ali-Shahid and Sulaiman, 2015]

A previous work Cavamura Jr. et al. [2020b] evidenced relevant information about *SOP* and the software testing:

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 misalignment between *SOP* and software testing;

3. Evidence that there are faults concentrated on untested parts of the software;

*Paper submitted to IET Software. Currently under evaluation.

We thank the Ph.D. Professor Daniel Lucrédio for his valuable discussions and suggestions about the *OPDaTe* development.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001

4. 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 slight improvement.

Based on the results of previous work and aiming to reduce the possible misalignment between *SOP* and the software test and that studies related to the *SOP* focus on exploring the operations and not exploring the operations' parameters [Shukla, 2009], this work presents the *OPDaTe* Tool.

Based on *SOP*, *OPDaTe* automatically generates runnable unit test cases at method level for programs written in *Java* language. *OPDaTe* uses dynamically captured data from the software operation under test as test data for the generation of test cases. Additionally, this paper also present the results obtained from an initial evaluation of the effectiveness of *OPDaTe* about its objectives.

We organize the remaining of this paper as follows. In Section 2 we presents the theoretical foundation. In Section 3, related works are presented. The *OPDaTe* and its features are presented in Section 4. Section 5 describes the adopted methodology for this research. The Feasibility Study (*FES*) performed to provide an initial evaluation of the *OPDaTe* is described in Section 6. Sections 7, 8 and 9 discuss the threats to validity related to the *FES* carried out, the lessons learned and future works, respectively. Finally, Section 10 concludes the paper.

2 Software Operational Profile and the Software testing

Software testing is a process to ensure that the software conforms with its project, avoiding unintended and unexpected actions by software [Myers et al., 2011]. This process takes time, investment, and effort. The absence of these demands makes software testing vulnerable because it does not directly impact software delivery. Thus, automated testing is due to the need for an increasingly reliable, effective, and efficient process to save time and minimize errors during the process [Vincenzi et al., 2018]. However, automated testing involves investments in infrastructure and requires effort to specify and program test cases [Vincenzi et al., 2018].

A test case corresponds to a tuple formed by the test data and the predicted value returned by the coding unit under test. Test data corresponds to the input data required to perform the tested code unit and generate the expected returned value for those inputs. Test data can be obtained by using test data generation techniques. These techniques differ by adopting different ways of generating and securing test data, such as random generation, generation with dynamic execution, among others [Vincenzi et al., 2018]. The use of production data, that is, data obtained directly from the software operation by users [Rogstad and Briand, 2016], can be attributed to the test cases.

As experienced users can adapt the way the software is operated [Sommerville, 2011], the software operations can present different behaviors and results, thus allowing to identify additional ways to perform the same operation in the same software product. The information obtained directly from the software operation is relevant for the software testing since the way the software is operated may impact the behavior and results of the software operations. Thus the software quality is dependent on its operational use [He et al., 2021, Zhakipbayev and Bekey, 2021]. Information obtained directly from the operation of the software by users is related to the Operational Profile of the Software.

Software Operational Profile (*SOP*) is a software specification based on how users operate the software [Musa and Ehrlich, 1996, Sommerville, 2011]. Musa [1993] defines *SOP* as a software quantitative characterization based on the way software is operated. This characterization corresponds to probabilistic distribution assigned to performed software operations to identify users' most used software parts. In this context, an operation corresponds to a task performed by the software delimited by external factors not related to software implementation. Software operations have various possible execution paths, depending on the given input data, causing different behavior and, consequently, providing other results for the same software operation.

Since the *SOP* represents how the users operate the software, *SOP* supports activities related to software engineering reliability. Software reliability depends on how the users operate the software because software reliability corresponds to the probability of a software operation occurring without any occurrence of failure in a specified period and a specific environment [Musa, 1979]. The *SOP* provides test data that reproduces the way software is executed in its production environment, ensuring the validity of reliability indicators [Musa and Ehrlich, 1996]. The use of *SOP* guarantees that the most used parts of the software are sufficiently tested [Ali-Shahid and Sulaiman, 2015]. Thus, it is one of the software's attributes related to software quality representing the user's point of view on software quality [Musa, 1979, Bittanti et al., 1988]. For the software user, the frequency with which a failure becomes apparent, during software operation, is more significant than remaining failures [Takagi et al., 2007].

Previous work investigated the possible mismatch between *SOP* and the tested software parts by introducing the "test profile" term. The term test profile represents the most exercised software parts during the test suite execution [Cavamura Jr. et al., 2020b]. Cavamura Jr. et al. [2020b] adopted the method level as granularity. Thus, the term granularity corresponds to the software fragmentation level (conceptual or structural). The *SOP* in the Cavamura Jr. et al. [2020b] work context corresponds to the processed methods frequency during the software operation by users indicating the software parts most operated. The test profile corresponds to the set of processed methods during the test suite execution. The results evidenced that there are significant variations in the way software is used by users.

Cavamura Jr. et al. Cavamura Jr. et al. [2020b] identified the *SOP* of four software, S_1, S_2, S_3, and S_4, respectively. In Table 1 we provide a general data about S_1, S_2, S_3, and S_4 software. The study used the *SOP* of the S_1 software to investigate whether there are significant variations in how users use the software. The frequency of execution of each method processed during the S_1 operation by one participant was compared to the frequency of execution of the respective method processed by the other participants. On average, the variation in execution frequencies for each method processed by the participants was approximately 50%. The results also evidenced a misalignment between the *SOP* and the test profile, and failures may occur in the untested *SOP* parts. For S_1, S_3, and S_4, approximately 50% of the methods processed by *SOP* were not processed in the test profile. About 25% of the methods processed by *SOP* were not processed in the test profile for S_2. It is also possible to verify the occurrence of methods processed by the test profile and not processed by *SOP*. For example, approximately 30% of the methods processed by test profile were not processed in the *SOP* for S_2.

Cavamura Jr. et al. [2020b] checked for failures in parts of the operational profile not contained in the test profile of the S_2 and S_4 software. As a result, we obtained evidence that failures may occur in the parts of the operational profile untested. For S_2, 62.02% of the *SOP* parts identified the test profile did not cover faults. For S_4, the respective value was 33.33%

Cavamura Jr. et al. [2020b] also checked whether an approach based on the combination of existing test cases with test cases generated by automated tools could decrease a misalignment among *SOP* e test profiles. We submitted S_2 and S_4 software to an automated test case generator. We incorporated the test cases generated by the test case generator into the existing test case set of the software. For S_2, the strategy decreased by approximately 60% the number of methods processed by *SOP* and not processed by the test profile. For S_4, the respective value was around 10% Thus the strategy was able to decrease the misalignment however don't able to avoid the misalignment.

Given the theoretical foundations presented, the *SOP* is relevant to activities that determine software reliability and the planning and execution of the test activity regardless of the adopted test strategy. We design the *OPDaTe* tool as a test case generator based on *SOP* data collected dynamically, i.e., during the software operation by

Table 1: Characterization of the software used in previous work.

Software	Purpose	Source	Methods	Origin of test cases
S_1	Provide software inspection support (*Crista*)	Closed source	2749	Computational tool
S_2	Bibliographic reference management (*JabRef*)	Open source	7100	Community
S_3	Process Automation (developed on demand)	Closed source	869	Test team
S_4	CASE tool (*ArgoUml*)	Open source	18099	Community

users. Therefore, it contributes to helping reduce the misalignment between the *SOP* and the test profile since that *OPDdTe* identify the methods most processed by software during the software operation and generate test cases for the interest methods specified by the tester. Thus, the tester can use the *OPDdTe* tool under different strategies. *OPDdTe* can generate test cases from the beginning of the software product execution process, generating test cases fully adherent to the *SOP*. *OPDdTe* can complement an existing test suite that possibly does not perform all the functionality that makes up the *SOP*. In addition, even when the available tests perform the functionalities that make up the *SOP*, the *OPDdTe* tool built test cases with real data demanded by users, unlike the tests generated during product development, which generally use data test samples.

3 Related Works

Previous work [Cavamura Jr. et al., 2020a] investigated the use of *SOP* in specific applicability, including *SOP* in activities associated with regression testing and the use of *SOP* as an evaluation criterion for existing test cases. The previous work carried out a Systematic Literature Review to carry out the investigation. The Systematic Literature Review processed 430 studies resulting in 11 studies selected for the data extraction and results from the synthesis phase. Furthermore, the use of *SOP* related to test cases generation was checked in 5 of the 11 selected studies. A synthesis of these 5 studies is presented below.

According to Marijan et al. [2010], software quality decreases as software complexity increases. Marijan et al. [2010] propose a methodology that allows evaluating this relationship in multimedia systems. The proposed method comprises the behavioral modeling of the system. The behavioral model is based on the functional specification of the system and represents the basis for deriving a system usage profile. The usage profile corresponds to the behavioral model plus the probabilities of occurrence of transitions existing in the behavioral model. The sequences of transitions of the behavioral model give rise to test cases that are automatically executed. The results obtained by performing the test cases are analyzed to assess the quality of the software about its complexity.

Nakornburi and Suwannasart [2016] report the development of a computational tool that allows prioritizing and selecting an optimal set of input parameters and generating test cases using combinatorial testing techniques. In the Nakornburi and Suwannasart [2016] study context, *SOP* is composed of data related to users and the environment in which the software is operated, such as the user's country of origin and the technology used to run the software. Based on this data, the proposed tool prioritizes and selects an ideal set of input parameters in the generated test cases. Furthermore, the tool also allows you to insert rules and restrictions related to possible combinations between these data.

Kashyap [2013] presents an approach for creating a behavior model that allows the automatic generation and prioritization of test cases. The *SOP* is used to estimate the probability of transitions between the states that make up the behavior model and represent the states that the software can assume. The behavior model allows generating the test cases. The proposed approach determines the representativeness of the test cases about the use of the software, establishing a priority to the generated test cases. The proposed approach sets a lower priority to test cases that do not reflect actual software usage.

Rogstad and Briand [2016] performed an empirical investigation on strategies for selecting and generating test data for combinatorial testing techniques at the regression test level. Rogstad and Briand [2016] additionally used the *SOP* in one of the testing techniques applied in the investigation. The technique that used *SOP* creates a representation of each functionality to be tested. The *SOP* provides a probabilistic distribution of the occurrence of equivalence classes referring to the data processed by the functionality contained in the representation. The functionality representations to be tested generates a test suite specification.

Takagi et al. [2007] present a technique that assesses software reliability as part of the maintenance process, estimating and predicting failures that will influence the reliability of future software versions. The technique uses *SOP* to generate a usage model applied to the technique. The usage model allows the generation of test cases associated with the proposed technique. The usage model comprises a finite state machine where states represent specific operations performed by users. The technique labeled the transitions between model states with the probability of occurrence obtained from the *SOP*. The *SOP* data is obtained from log files or instrumenting earlier versions of the software under test. The technique generates the test cases from sequences of operations provided by the usage model.

Of the studies that used *SOP* in approaches associated with test case generation identified in systematic review, studies [Takagi et al., 2007, Nakornburi and Suwannasart, 2016, Rogstad and Briand, 2016], and [Kashyap, 2013] used *SOP* to obtain a usage model. A probabilistic distribution is related to the model to indicate which parts of the software are most used. In these studies, *SOP* was obtained from the software specification, log files, or collected from the software operation by the users. The approaches proposed in these studies use the model created to generate and run the test cases. The approach proposed by Marijan et al. [2010] used data external to the software to specify a software usage profile. Then, the method uses this profile to generate test cases and select test data for the generated test cases.

Also related to this work, we are looking for automated test tools based on test amplification and symbolic execution to try reduce the possible misalignment between the *SOP* and the tested parts of the software. The *Palus* [Zhang, 2011] and *Dspot* [Danglot et al., 2019b] tools are examples of these approaches for *Java* programs. Test amplification aims to leverage existing test cases written by hand on the premise that developers gain valuable knowledge about test data and expected results when writing test cases [Danglot et al., 2019a]. Symbolic generation assigns symbolic values to the test data and uses expressions that represent the conditions for the execution of the program under test [Vincenzi et al., 2018].

Palus' strategy consists of dynamic analysis, static analysis, and guided random test generation. First, *Palus* performed a dynamic analysis by running the software under test to create a sequence model of software method calls. Then, *Palus* performs static analysis to identify dependency relationships between the methods based on the developed call model. Finally, a random test generator is guided by the model of calls and identified dependencies. The *Palus*'s random test case generator provide *JUnit* test cases as result. The goal of *Dspot* is to synthesize modifications to existing test cases to increase test quality. *Dspot* receives as input a set of existing *JUnit* test cases manually written by the developers. Then, *Dspot* produces variants from provided test cases. Thus, *Dspot* tries to improve the overall test suite quality by putting together existing test cases and their variants.

We performed a previous feasibility study where subjected the S_1 and S_3 software, described and used in our previous study, to the *Palus* and

Dspot tools to minimize possible misalignment between the *SOP* and the tested parts of the software. We adopted as the criterion for software selection was based on the origin of the test cases. We selected the S_3 software because a test team provided its test cases. We also selected the S_3 software because the automated tool provided its test cases. We chose S_3 to verify the impact of the test case origin on the results provided by the tools. We consider the execution of the test set of software S_1 and S_3 as the symbolic execution of the software under test. Thus, the *Palus* tool performed the dynamic analysis based on the performance of test cases of the S_1 and S_3 software.

The *Dspot* tool received the original test sets from the S_1 and S_3 software as an input argument. Table 2 shows the results obtained from *Palus* and *Dspot* for S_1 and S_3 software.

We found that the new test cases generated by the *Palus* tool for S_1 and S_3 software did not increase the coverage of the respective software when merged with the existing test case sets of this software. New test cases generated by the *Dspot* tool for S_3 software increased the coverage of S_3 software by 1.1% when joined to the existing test case set. For S_1 software, the test case generated by the *Dspot* tool did not increase the coverage of the respective software when joined to the existing test case set. The *Dspot* tool generated a new test case for the S_1 software from the existing test cases. This result may be due to the origin of existing test cases of the S_1 software. In addition, the *Dspot* tool generated a variant from test cases generated by an automated test generator. The results obtained by our previous studies [Cavamura Jr. et al., 2020b,a] and the results obtained by the previous feasibility study with the *Palus* and *Dspot* tools motivated us to create *OPDaTe* tool. In the following sections, we presented *OPDaTe* and the results obtained from an initial evaluation based on code coverage of the tested unit.

4 The *OPDaTe* tool

Based on *SOP*, *OPDaTe* automatically generates Runnable Unit Test Cases (*RUTCs*) at method level for programs written in *Java* language. We developed the *OPDaTe* under the object-oriented programming paradigm and implemented it using *Java* language. The *OPDaTe* uses dynamically captured data from the software operation under test as test data to generate test cases. Initially, *OPDaTe* creates Abstract Test Cases (*ATCs*) from data dynamically caught while its users operate the Software Under Test (*SUT*). From generated *ATCs*, *OPDaTe* develops Runnable Test Cases (*RUTCs*) written in the *JUnit* framework. The *OPDaTe* is composed of two components, an *Agent* and an *RUTCs* Generator ($RUTC_{ger}$). Figure 1 represents the basic architecture of *OPDaTe*.

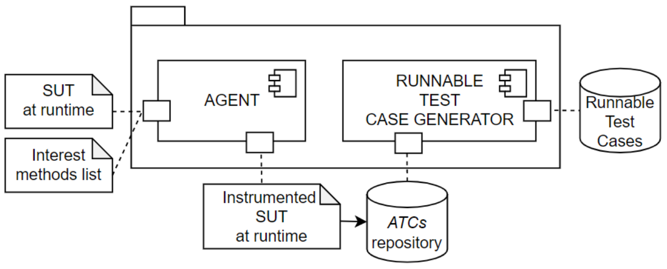


Figure 1: OPDaTe architecture

The *Agent* component is responsible for instrumenting the *SUT* to generate *ATCs* containing users' data collected during the *SUT* operation. The *Agent* receives as an input argument a list containing the name of the methods of interest. The methods of interest correspond to the methods for which the instrumented *SUT* will generate the respective *ATCs*. The list contains methods' names of interest with their classes and package or just the name of the classes and their packages. If the list contains classes' names, all the classes' methods will be considered methods of interest.

In the context of this work, an Abstract Test Case (*ATC*) corresponds to a test case specification for a method of interest. This specification is composed of test data and the expected return value for the specified test case. The *ATC*'s test data corresponds to the input argument types and values of the method of interest collected dynamically. The *Agent* collects both basic type data and data whose type is by reference. The expected return value of the *ATC* matches the type and value returned by the method of interest collected dynamically when it provides a return value. The instrumented *SUT* logs a warning on

Table 2: Previous feasibility study results

.	*Palus*		*Dspot*	
Software	new test cases	increased coverage (%)	new test cases	increased coverage (%)
S_1	37	0	1	0
S_3	27	0	23	1.1

the *ATC*'s test data when it cannot collect the value dynamically. The logs also happen when it fails to collect the return value of the method of interest.

The $RUTC_{ger}$ component generates the *RUTCs* from the generated *ATC*. We described each component in the following subsections.

4.1 *Agent*

The *Agent* is responsible for instrumenting the *SUT* to generate *ATCs* containing users' data collected during the software operation. Figure 2 represents the architecture of *Agent* component.

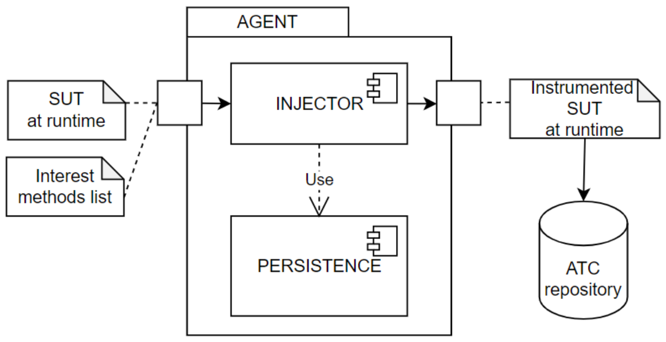


Figure 2: Agent Component

The *Agent* is composed of two other components: the *Injector* component and the *Persistence* component. The *Injector* is responsible for instrumenting the *SUT*. The *Persistence* component is used in the code injected by the *Injector* to persist *ATCs*.

The instrumented *SUT* monitors the execution of its methods, and every time a method of interest is executed, instrumented *SUT* generates the respective *ATC*. The instrumented *SUT* generates an *ATC* from the collected data for each performed method of interest or its overload. We can parameterize the *Agent* to collect a specific amount of *ATCs* for the provided methods of interest. Furthermore, the instrumented *SUT* persists each *ATC* in *JSON*[1] format in a file with the "`.atc`" extension.

4.2 Runnable Unit Test Case Generator Component

The $RUTC_{ger}$ generates the *RUTCs* from the generated *ATCs*. Figure 3 represents the architecture of $RUTC_{ger}$ component. It is composed of five components: *Core* component, *Persistence* Component, *DataProvider* (*DP*) component, *Oracle* component and *RunnableUnitTestCaseFactory* ($RUTC_{fac}$) component.

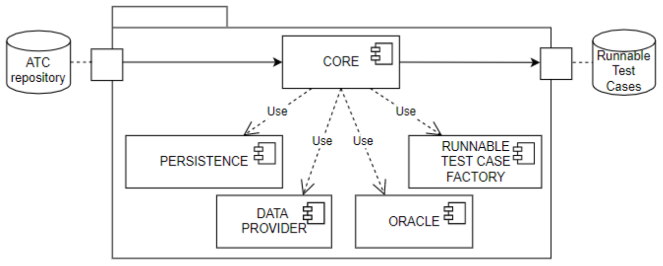


Figure 3: Runnable Unit Test Case Generator Component

Initially, the $RUTC_{ger}$ reads all "`atc`" files through the *Persistence* component, retrieving the *ATCs*. While abstract test cases are retrieved, the component also writes test data from *ATCs* to an internal repository. Then, the $RUTC_{ger}$ automatically tries to execute the respective method under test for each *ATC* through *Oracle* component. If the method under test is an overloaded method, the *Oracle* will also try to execute the other method instances.

The *DP* component provides the test data, that is, the input arguments, necessary for the automatic execution of the method under test

[1]*JSON* is a simple, readable data structure used to exchange data between systems.

and its overloads. For each input argument required for the method execution, the *DP* component first tries from the internal repository to retrieve the value obtained dynamically for the respective argument of the method under test. The *DP* component checks the status given by the *Agent* component for the collected argument value. *OPDaTe Agent* tries to dynamically collect type data by reference (objects) even if they are not serialized objects, that is, even if the class that originated them does not implement the "Serializable" interface. Thus, exceptions may occur when *OPDaTe Agent* tries to collect these data due to technology limitations. In this situation, the *OPDaTe Agent* indicates the failure status for the respective argument in the *ATC*. If the status indicates failed collection, the *DP* component tries to get another value of the same type or subtype in the internal repository. If the *DP* component doesn't find another value, it generates a random value for the argument.

The return value obtained by the *Oracle* from the automatic execution of the each method is stored and used as the expected value for the *RUTC* that $RUTC_{ger}$ will generate. This kind of test case is called Regression Test Case by other tools like EvoSuite[2] and Randoop[3], for instance. The value returned by the method under test during software operation is also dynamically collected by the *SUT* instrumentation and recorded in the *ATC*. However, the $RUTC_{ger}$ uses the value obtained by the *Oracle* to generate the *RUTCs*. We collect and record the value returned by the method during software operation to add value to the *ATC*, allowing the tester to have the test case specification in case he wants to write the *RUTC* manually. If the *ATC* does not have an expected return value, the *ATC* refers to a method with no return. In this case, the $RUTC_{fac}$ encapsulated the method in an exception handling in the runnable test case.

The $RUTC_{fac}$ component receives the data processed and obtained by the *Oracle* after the *Oracle* automatically executes the method under test and its overloads and obtains the values returned by the respective method. Finally, the $RUTC_{fac}$ component writes an *RUTC* in *JUnit* format for each instance of the method automatically executed, that is, the method under test and the other overloads of the method. The *JUnit* files generated are standardized to be inserted into projects under the *Maven* architecture.

The *Oracle* component may not automatically execute the method under test due to exceptions that may occur during the execution attempt, such as "IllegalAccessException"[4] and "InvocationTargetException"[5] exceptions. In case of the *Oracle* component cannot automatically run the interest method, the $RUTC_{fac}$ writes the respective runnable test case using the test data and returned value from the interest method' *ATC*. In this situation, the runnable test case may also not be able to execute the method of interest. To assist the tester, the $RUTC_{fac}$ notes in the runnable test case the possible causes by the not automatically execution of interest method by the *Oracle*. The respective *RUTC* is also marked to be ignored (@*Ignore* annotation[6]) to avoid execution errors during test running.

4.3 OPDaTe Operation Flow

The Figure 4 shows the *OPDaTe* tool operation flow. From Figure 4, we can see the two components of *OPDaTe*, the *OPDaTe Agent* and the *OPDaTe RUTCs* Generator ($RUTC_{ger}$).

Initially, the *SUT* is executed with the *OPDaTe Agent* to be operated by the users. The *OPDaTe Agent* input arguments are:

1. The first input argument is the file name that contains the list of methods of interest (Figure 4-1) for which the instrumented *SUT* must generate the *ATCs* files.

2. "*filtertype*": It determines whether the interest methods monitoring will be done by the

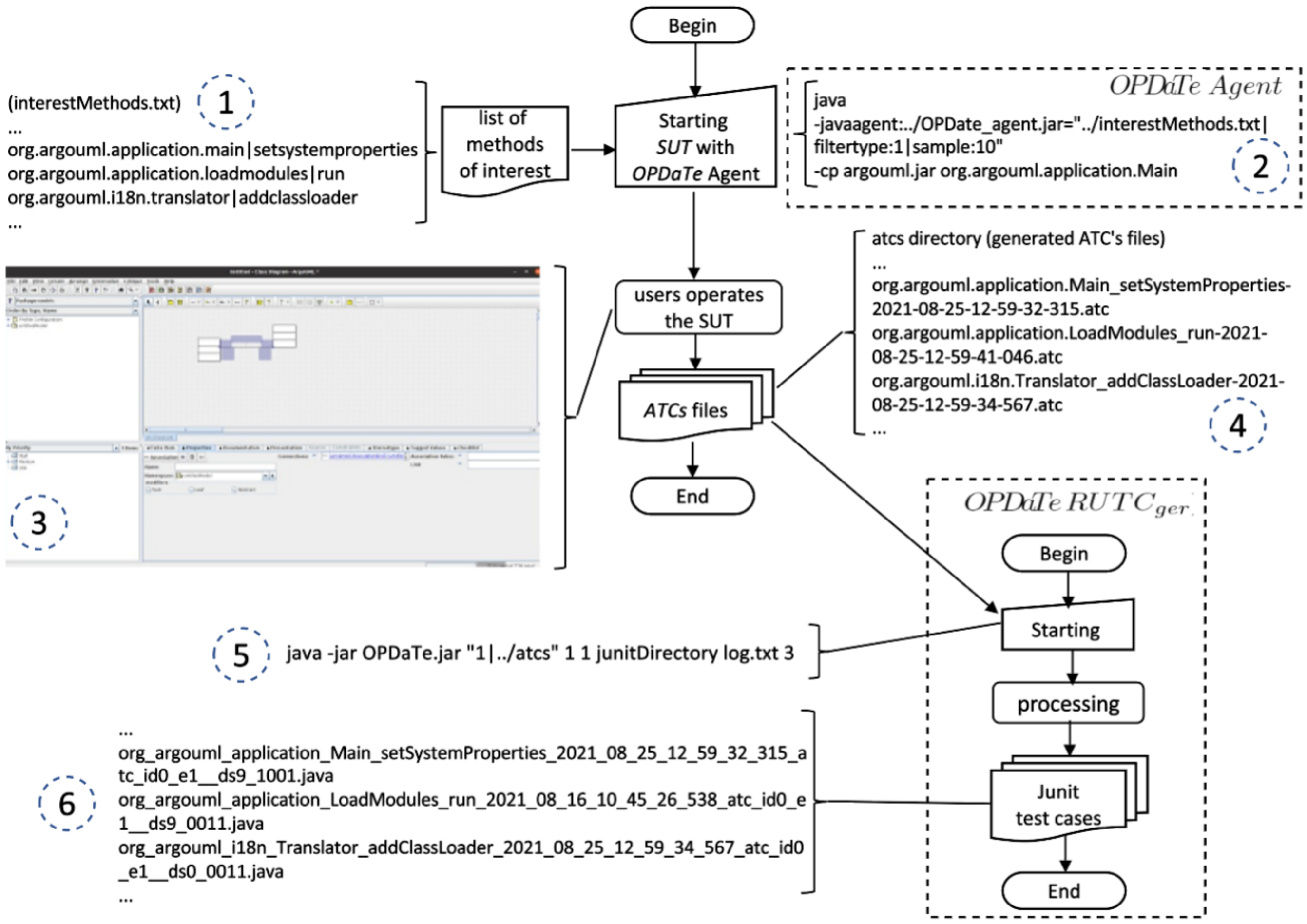
[2]`https://www.evosuite.org/evosuiter/`

[3]`https://randoop.github.io/randoop/manual/#regression_tests`

[4]`https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/IllegalAccessException.html`

[5]`https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/reflect/InvocationTargetException.html`

[6]Test cases annotated with the @Ignore tag are ignored when the test suite is run



```
(interestMethods.txt)
...
org.argouml.application.main|setsystemproperties
org.argouml.application.loadmodules|run
org.argouml.i18n.translator|addclassloader
...
```

```
java
-javaagent:../OPDate_agent.jar="../interestMethods.txt|
filtertype:1|sample:10"
-cp argouml.jar org.argouml.application.Main
```

```
atcs directory (generated ATC's files)
...
org.argouml.application.Main_setSystemProperties-
2021-08-25-12-59-32-315.atc
org.argouml.application.LoadModules_run-2021-
08-25-12-59-41-046.atc
org.argouml.i18n.Translator_addClassLoader-2021-
08-25-12-59-34-567.atc
...
```

```
java -jar OPDaTe.jar "1|../atcs" 1 1 junitDirectory log.txt 3
```

```
...
org_argouml_application_Main_setSystemProperties_2021_08_25_12_59_32_315_a
tc_id0_e1__ds9_1001.java
org_argouml_application_LoadModules_run_2021_08_16_10_45_26_538_atc_id0_e
1__ds9_0011.java
org_argouml_i18n_Translator_addClassLoader_2021_08_25_12_59_34_567_atc_id0
_e1__ds0_0011.java
...
```

Figure 4: OPDaTe tool operation flow

classes (value 0) or by the methods (value 1) specified in the file.

3. "*sample*": When its value is more than 0, it determines the specific amount of $ATCs$ that will be generated for each interest methods.

The instrumented SUT creates a directory named "atc" to store the generated $ATCs$ files. This directory is created below the current directory where we started the SUT execution. For example, Figure 4-2 shows the command line that runs the SUT with the *OPDaTe Agent*. The instrumented SUT generates $ATCs$ while users operate it (Figure 4-4). Figure 4-3 shows the SUT interface.

OPDaTe $RUTC_{ger}$ generates executable unit test cases (*JUnit* files) from $ATCs$ collected during execution of instrumented SUT by users. The SUT and its dependencies must be on the "classpath" to run the *OPDaTe* $RUTC_{ger}$. For example, Figure 4-5 shows the command line that executes the *OPDaTe* $RUTC_{ger}$. *OPDaTe* $RUTC_{ger}$ input arguments are:

1. "*first argument*": Correspond to the processing type and the input to be processed ("*type|entry*"). The processing type determines whether *OPDaTe* will process a specified ATC file (value 0) or ATC files contained in a directory (value 1). The entry corresponds to the ATC file's name to be processed or the directory containing the ATC files to be processed, according to the type of processing informed.

2. "*second argument*": It determines whether *OPDaTe* will generate test cases using only random test data (value 0) or use operational profile and random test data (value 1) when necessary.

3. "*third argument*": This argument determines whether the *OPDdTe* will use the dynamically obtained object that invokes the interest method (value 0) or if it will be created randomly (value 1). We have not yet implemented this parameterization. We will make this parameterization available in future *OPDdTe* releases. The value given to this argument does not affect *OPDdTe* processing. Currently, *OPDdTe Oracle* component randomly generates the objects that invoke the interest methods under test.

4. "*fourth argument*": Corresponds to the directory name will be created by *OPDdTe* to store the *Junit* files (*Maven* structure). *OPDdTe* creates this directory in the current directory where we started it.

5. "*fifth argument*": Corresponds to the log file's name will be created by *OPDdTe*.

6. "*sixth argument*": Output type: *OPDdTe* print the *JUnit* file content in the terminal (value 1); *OPDdTe* store the *JUnit* files into the gave directory; Both (value 3), *OPDdTe* print the *JUnit* file content in the terminal and store the *JUnit* file into the gave directory;

The *OPDdTe* $RUTC_{ger}$'s stores the *JUnit* files (Figure 4-6) into the directory specified in their list of input arguments. Then, *OPDdTe* $RUTC_{ger}$'s creates the *Maven* directory structure containing the *JUnit* files from the created directory.

We experimentally evaluated the ability of *OPDdTe* to create *RUTCs* using data extracted from the *SOP* as test data. We also assess the coverage provided by the test cases generated in the respective tested units. In addition, we also investigates if a test strategy based on test suites obtained from different *SOP* can contribute to expanding the coverage of the tested units. We described the methodology adopted and its activities in the next section.

5 Methodology

We defined three research questions to guide our initial evaluation about the *OPDdTe*:

- RQ_1: Is *OPDdTe* able to automatically generate executable test cases using real test data extracted from the *SOP*?
- RQ_2: Did the automatically generated runnable test cases provide satisfactory coverage for the respective tested units?
- RQ_3: Can a test strategy based on test suites obtained from different *SOP* contribute to expanding the coverage of the tested units?

We adopted and instantiated the methodological steps proposed by Shull et al. [2001] to investigate the effectiveness of *OPDdTe*. The methodological steps proposed by Shull et al. [2001] is shown in Figure 5.

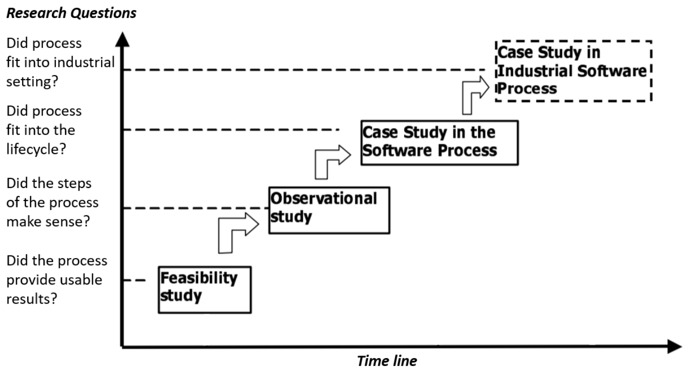

Research Questions
Did process fit into industrial setting?
Did process fit into the lifecycle?
Did the steps of the process make sense?
Did the process provide usable results?
Case Study in Industrial Software Process
Case Study in the Software Process
Observational study
Feasibility study
Time line

Figure 5: Adopted Research Methodology (adapted from Shull et al. [2001])

.

The focus of this paper is on the "Feasibility Study". The "Feasibility Study" phase comprised the accomplishment of feasibility study (*FES*) to provide answers to RQ_1, RQ_2 and RQ_3 and, thus, get results that provide initial evaluation about the *OPDdTe* effectiveness and the feasibility of future studies in the next adopted methodology phases. Given the "Feasibility Study" phase results presented in this paper, the "Observational Study" phase progresses. The "Case Study: Lifecycle" and "Case Study: Industry" phases will perform depending on the "Observational Study" phase results. As soon as we get their results, we will publish them. We instantiated the *GQM* [Basili et al., 2002] model to plan the *FES*. The instantiated model is shown in Table 3.

As shown in Table 3, we plan the execution of *FES* through 3 activities to provide answers for the research questions, called $FES-AT_1$, $FES-AT_2$ and $FES-AT_3$, respectively. In Section 6, we present the *FES*'s activities and their respective results are described.

6 Feasibility Study

We performed the Feasibility Study (*FES*) to answer research questions defined in this paper and thus get an initial evaluation of *OPDaTe*'s effectiveness. As mentioned in Section 5, the *FES* consisted of three activities, called $FES-AT_1$, $FES-AT_2$ and $FES-AT_3$.

Activity $FES-AT_1$ gets the *ATCs* during the *SUT* operation by users through the *Agent* component of *OPDaTe*. Activities $FES-AT_2$ and $FES-AT_3$ use the *RUTCs* generated by *OPDaTe* from *ATCs* provided by $FES-AT_1$. The results obtained from activities $FES-AT_2$ and $FES-AT_3$ answer research questions RQ_1, RQ_2, and RQ_3, respectively. In the following subsections, we describe the activities of the *FES*, and their results.

6.1 FES-AT1

We performed the $FES-AT_1$ activity to gets the *ATCs* during the software operation by users through the *Agent* component of *OPDaTe*. To perform the $FES-AT_1$, we chose a software used in our previous study [Cavamura Jr. et al., 2020b] to be submitted to *OPDaTe* as *SUT* in this study. The *SUT* adopted is an open-source *CASE* tool called *ArgoUML*[7] - version v0.30.1. In order to reduce the risks associated with the threats to validity of the activity, we chose 10 participants with equivalent experience and knowledge in Software Engineering. The chosen participants are Master's and Ph.D. students in Computer Science, with knowledge about the *SUT* operation. We assign each participant the task of drawing up the analysis class diagram from a provided requirements document. We provide the same requirements document for all participants. Participants used the *SUT* to build the requested class diagram according to their interpretation of the requirement document. We set a time limit for participants to complete the task. The tasks performed within the defined time period were considered successfully completed. All participants were able to complete the activity before the end of the stipulated time period.

The list of methods of interest that we provide as an input argument for the *OPDaTe* was composed of 2,961 of the 4,472 *SUT* methods classified by [Cavamura Jr. et al., 2020b] as contained in the *SOP* and not contained in the test profile. The *OPDaTe Agent* monitors the processing of *SUT* methods by collecting and persisting data when a method of interest is processed. However, monitoring affects the execution performance of the monitored software. This paper an initial evalauation of the *OPDaTe* tool based on the results obtained when using it in a software product.

This paper presents the *OPDaTe* tool and the initial assessment of its effectiveness in terms of its purpose in the feasibility study phase of the adopted methodology. Thus, the *OPDaTe* tool is a prototype in which we have not yet performed performance optimizations. Therefore, we do not include GUI classes and parallel processing methods in the list of methods of interest to obtain a better performance in executing the *SUT* during the activity. We implemented a program to analyze the methods that made up the list of methods of interest and thus identify the interface classes and the classes that implemented parallel processing methods.

Table 4 shows, for each participant, the total of *ATCs* ($ATCs_{all}$) generated during the performance of the activity. In addition, Table 4 also indicates the number of distinct *ATCs* ($ATCs_{dist}$) for the interest methods, disregarding the other instances of the *ATCs* generated for the same interest method. Each participant's $ATCs_{dist}$ group was composed of the last *ATC* generated for the methods of interest during the activity. The number of failed *ATCs* ($ATCs_{fail}$) contained in the

[7] `https://argouml-tigris-org.github.io/`

Table 3: Exploratory Study Planning.

Stage	Analyze	For the purpose of	Focus	Perspective	Context
$FES-AT_1$	The *SOP*	Generate *ATCs* during the software operation by users	Methods processed by users during software operation: collect of input parameters and return value (if any)	Software test researchers	Software applications users
$FES-AT_2$	runnable test cases generated by *OPDdTe*	Identify the source of test data	Runnable test cases test data	Software test researchers	Software applications users
$FES-AT_3$	**(a)**Coverage provided by runnable test cases generated by *OPDdTe*	**(a)**Check how much of the tested unit was covered	**(a)**Tested units by the runnable test case	Software test researchers	Software applications users
	(b) Coverage provided by test suites generated by *OPDdTe*	**(b)**Check if test suites obtained from different *SOP* contribute to expanding the coverage of the tested units	**(b)**Coverage provided by runnable test cases generated by *OPDdTe*		

$ATCs_{dist}$, i.e., not readable, is also shown in Table 4.

Table 4: ATCs generated by participants

Participant	Generated $ATCs$	$ATCs_{dist}$	$ATCs$ fails on $ATCs_{dist}$	Readable $ATCs_{dist}$
P_{01}	2857	409	3	406
P_{02}	3433	473	3	470
P_{03}	3433	473	3	470
P_{04}	3241	438	3	435
P_{05}	294193	479	0	439
P_{06}	2897	421	5	416
P_{07}	663123	410	3	407
P_{08}	3428	460	3	454
P_{09}	3080	414	3	411
P_{10}	2210	346	7	339

Through Table 4, we can verify that participants P_{05} and P_{07} stand out about the amount of $ATCs$ generated. This situation occurred because the execution environment used by participants P_{05} and P_{07} in which they performed the activity did not identify the parameterization of the *OPDdTe*'s *Agent* referring to the number of $ATCs$ instances to be collected. Coincidentally, participants P_{02} and P_{03} obtained the same values for the activity performed. To better analyze results, we performed activities $FES-AT_2$ and $FES-AT_3$ using the $ATCs_{dist}$ groups generated by each participant. For each $ATCs_{dist}$ group, we use the *OPDdTe* to generate the $RUTCs$ from the $ATCs$ contained in each group. So, for each $ATCs_{dist}$, we create a set of $RUTCs$. Table 5 shows the data obtained from each participant about the generation of $RUTCs$ from $ATCs$.

Table 5: Generated runnable unit test cases

Participant	$RUTCs$	Inactive $RUTCs$	Failed $RUTCs$	Effective $RUTCs$	Effective $RUTCs$ %
P_{01}	414	149	22	243	58.7
P_{02}	480	148	28	304	63.3
P_{03}	477	169	27	281	58.9
P_{04}	437	139	22	276	63.1
P_{05}	482	168	25	289	59.9
P_{06}	423	130	25	268	63.3
P_{07}	413	122	24	267	64.6
P_{08}	461	150	23	288	62.4
P_{09}	418	129	24	265	63.4
P_{10}	364	67	22	257	74.2

The value in the column "Participant" in Table 5 corresponds to the participant identification. The value in the column $RUTCs$ in Table 5 corresponds to the number of $RUTCs$ generated by *OPDdTe* from each $ATCs_{dist}$ (Table 4). The value in the column "Inactive $RUTCs$" in Table 5 corresponds to the number of runnable test cases classified as inactive by *OPDdTe*. Inactive test cases correspond to test cases annotated with the JUnit @Ignore tag by *OPDdTe*. Test cases annotated with the @Ignore tag are ignored when the test suite is run. The value in the column "Failed *RUTCs*" in Table 5 corresponds to the number of runnable test cases that failed after execution, showing characteristics of "flaky tests". "Flaky tests" are tests that have a non-deterministic behavior. They may provide different output on consecutive execution, even when they exercise an unaltered piece of code from the *SUT* [Eck et al., 2019]. When the *OPDdTe Agent* cannot retrieve the value of an argument whose data type is by reference, or when the *OPDdTe DP* cannot retrieve this value, the *DP* generates a value randomly so that *OPDdTe Oracle* can automatically execute the method under test. *OPDdTe* doesn't carry over this object to the materialized test case (*JUnit* file). Thus the materialized test case randomly generates this object again when its executed. This situation can cause the *JUnit* assert to fail since the expected result obtained from *Oracle*, and the returned result obtained from materialized test case were provided with objects with different states.

The value in the column "Effective *RUTCs* in Table 5 corresponds to the number of runnable test cases that ran successfully (effective test cases).

6.1.1 Data analysis

We can verify by Table 5 that the amount of *RUTCs* generated for each $ATCs_{dist}$ was greater than the amount of *ATCs* into the respective $ATCs_{dist}$ group.This situation occurs because the *OPDdTe* generates the test cases for the methods of interest and overloads those methods when they exist. Table 6 exemplifies this situation. In the example shown by Table 6, the *OPDdTe Agent* generated one *ATC* file for the method of interest *load* of the *Configuration* class. *OPDdTe* $RUTC_{ger}$ generated three *JUnit* files. Each *JUnit* file tested an instance of the *Configuration* class's *load* method.

Inactive *RUTCs* corresponds to the test cases for which the *Oracle* could not execute the method of interest or perform, and *JUnit* cannot run it. In this situation, the *OPDdTe* creates and annotates the *JUnit* test case with the @Ignore tag. *OPDdTe* also writes a comment in the test case with the possible reasons that caused the test inactivation.

Failed *RUTCs* occurs because the *DP* of *OPDdTe* can assign a random value to test data whose data type is by reference. The reference data types generated randomly when the *Oracle* executes the interest method are regenerated when we perform the executable test case (*JUnit*). Thus, when we perform the test case, the tested interest method may generate a result different from the result obtained by the *Oracle* and used as the assertion. This situation does not occur when the *DP* of the *OPDdTe* can retrieve a dynamically obtained value for the test data whose data type is by reference. Instead, the test case uses the exact value retrieved by the *Oracle* to execute the method of interest. The same basic type of test data provided by the *DP* for the *Oracle* is also used in the executable test case regardless of whether it was generated randomly or obtained dynamically.

6.1.2 Results

OPDdTe generated 981,895 *ATCs* during the execution of the *SUT* by the participants, generating on average more than 430 distinct *ATCs* per participant for the provided methods of interest. *OPDdTe* generated 4,351 *RUTCs* from distinct *ATCs* generated during *SUT* operation by participants, generating an average of 435 *RUTCs* per participant. In this study, these methods correspond to methods that made up the operational profile of the *SUT* not included in the test profile, that is, methods contained in the parts of the operational profile of the software not tested yet by any test case. However, we can provide any list of methods from any source to *OPDdTe*. *OPDdTe* can generate an initial test set for software that does not have an existing test suite as an example. Thus we considered the results of $FES-AT_1$ satisfactory.

Table 6: Handling overloaded methods

Context	Example
Method of interest Entry in the interest methods file (*OPDdTe Agent* input argument)	org.argouml.configuration.Configuration|load
Generated ATC file	org.argouml.configuration.Configuration_load-2021-08-25-12-59-32-209.atc
Generated JUnit files	org_argouml_configuration_Configuration_load_2021_08 _25_12 _59 _32_209_atc_id0_e1___ds0_0011.java (method under test: *public static boolean load()*) org_argouml_configuration_Configuration_load_2021_08_25_12_59 _32_209_atc_id1_e1___ds9_0011.java (method under test: *public static boolean load(File file)*) org_argouml_configuration_Configuration_load_2021_08_25_12_59 _32_209_atc_id2_e1___ds0_0011.java (method under test: *public static boolean load(URL url)*)

6.2 FES-AT2

We performed *FES–AT*$_2$ to verify if *OPDdTe* could generate executable test cases using operational profile data as test data. We initially analyzed the $ATCs$'s test data generated from each participant. We check the status and type of data contained in the analyzed $ATCs$. Table 7 shows the data from this analysis.

The value in the column "Participant" in Table 5 corresponds to the identification of the participant that gave rise to the analyzed $ATCs$. The value in the column "Error" in Table 7 corresponds to the amount of data that the *OPDdTe* was not able to dynamically collect during the SUT operation by the participants. The value in the column "Basic" in Table 7 corresponds to the amount of basic type data collected by the *OPDdTe* for the analyzed $ATCs$. The value in the column "Reference" in Table 7 corresponds to the amount of type data per reference collected by the *OPDdTe* for the analyzed $ATCs$.

We also analyze test data from $RUTCs$ generated by each participant. For each analyzed $RUTCs$, we check the origin of the respective test data. Table 8 shows the data from this analysis. Table 8 shows the data obtained from all $RUTCs$ generated by *OPDdTe* and obtained from the effective and failed $RUTCs$. The value in the column "NTD" (No Test Data) in Table 8 corresponds to the number of $RUTCs$ whose tested methods do not receive arguments, that is, test cases without test data. The value in the column "TD" (Test Data) in Table 8 corresponds to the number of $RUTCs$ whose tested methods receive arguments, that is, test cases with test data. The value in the column "R" (Random) in Table 8 corresponds to the number of $RUTCs$ whose test data was randomly generated by the DP. The value in the column "P" (Production) in Table 8 corresponds to the number of $RUTCs$ whose test data was provided by the data provider from data collected dynamically while the participants operated the SUT. The value in the column "B" (Both) in Table 8 corresponds to the number of $RUTCs$ whose test data was provided by the DP from data collected dynamically while the participants operated the SUT and were also randomly generated by the data provider.

6.2.1 Data analysis

The *OPDdTe Agent* generated an ATC for each method of interest that the participants processed during the SUT operation. The test data corresponds to the input arguments of the methods under test covered by the ATC. For each participant, we found that *OPDdTe* collected on average 82.40% of the input arguments of the intercepted methods of interest. We also found that, on aver-

Table 7: Type of dynamically collected data

Participant	Error	Basic	Reference	Basic (%)	Reference (%)	Collected Data (%)
P_{01}	39	188	54	77.69	22.31	86.12
P_{02}	56	218	68	76.22	23.78	83.63
P_{03}	50	213	59	78.31	21.69	84.47
P_{04}	48	186	62	75.00	25.00	83.78
P_{05}	47	215	72	74.91	25.09	85.93
P_{06}	78	178	28	86.41	13.59	72.54
P_{07}	42	179	63	73.97	26.03	85.21
P_{08}	76	214	26	77.33	22.67	83.45
P_{09}	49	191	56	86.17	10.93	75.95
P_{10}	40	139	55	71.65	28.35	81.91
			Average	78.07	21.93	82.40

Table 8: Runnable test cases test data source

	All Runnable Test Cases					Active Runnable Test Cases (Failed and Effective)									
Participants	NTD	TD	R	P	B	NTD	TD	R	P	B	TD(%)	R(%)	P(%)	B(%)	P+B(%)
P_{01}	214	200	122	62	16	145	120	77	37	6	45.28	64.17	30.83	5.00	35.83
P_{02}	239	241	145	78	18	175	157	101	48	8	47.29	64.33	30.57	5.10	35.67
P_{03}	247	230	146	69	15	173	135	91	38	6	43.83	67.41	28.15	4.44	32.59
P_{04}	225	212	131	65	16	163	135	90	38	7	45.30	66.67	28.15	5.19	33.33
P_{05}	246	236	145	74	17	173	141	91	43	7	44.90	64.54	30.50	4.96	35.46
P_{06}	222	201	136	47	18	159	134	98	28	8	45.73	73.13	20.90	5.97	26.87
P_{07}	215	198	119	64	15	158	133	84	43	6	45.70	63.16	32.33	4.51	36.84
P_{08}	238	223	142	64	17	170	141	100	33	8	43.94	66.14	29.13	4.72	33.86
P_{09}	216	202	125	62	15	162	127	84	37	6	45.34	70.92	23.40	5.67	29.08
P_{10}	172	147	105	54	15	145	134	88	38	8	48.03	65.67	28.36	5.97	34.33
							Average				45.54	66.61	28.23	5.15	33.39

age, 78.07% of the data collected is of basic type, and 21.93% is data of reference type. Based on the active *RUTCs* group (effective and failures) generated by each participant, we found that, on average, 54.4% did not contain test data. For this same group, on average, 30.3% used exclusively random data, 12.8% used exclusively dynamically obtained production data, and 2.3% used random and production data.

6.2.2 Results

Errors when collecting data dynamically and the amount of random test data in *RUTCs* reflect the constraints and limitations of technology in collecting non-serialized reference data at runtime and retrieving that data after running the software. Despite the restrictions and limitations, the *OPDdTe* collected on average 82.40% of the input arguments of the methods of interest intercepted during the *SUT* operation by each participant. Furthermore, for each participant, on average, *OPDdTe* was also able to use this data as test data in 33% of the *RUTCs* that require test data. These evidences answers research question RQ_1.

We could establish some restrictions to improve *OPDdTe* performance, like to restrict the analysis of methods which demands only basic data types or to demands all the referenced objects to be serialized. We decided not impose such a restrictions to observe which obstacle we must to overcome and will demand future research. Thus, we consider that the results obtained about dynamically collected production data as test data were satisfactory in this initial evaluation of *OPDdTe*.

6.3 FES-AT3

We performed $FES-AT_3$ to assess the coverage provided by executable test cases generated by *OPDdTe* for the tested units and to check if a test strategy based on test suites obtained from different *SOP* contribute to expanding the coverage of the tested units. For this, we run the test cases generated by each participant and identify the methods exercised by the respective test cases. For each method exercised, we measure code covered by the execution of test cases. We measure code coverage through the method statements exercised by the test case execution.

6.3.1 Data analysis

We created a table that presents the methods covered by each participant's test cases and the code coverage we measured for the respective methods. Table 9 shows a fraction of the table created[8]. The value in the column "ID" in Table 9 corresponds to the identification of the covered method. The value in the column "Covered Method" in Table 9 corresponds to the description of the covered method. The value in the columns P_{01} to P_{10} corresponds to the code coverage (instructions) of the tested method promoted by the *RUTCs* generated by the respective participants. The value in the column "Average" corresponds to the average code coverage of the tested method provided by the *RUTCs* generated by all participants. At the end of the table, we present the number of methods covered by the *RUTCs* of each participant and the average, minimum, and maximum coverage provided by the respective *RUTCs*. For each participant (Table 9), we found that the *RUTCs* generated by *OPDaTe* provided an average of over 90% code coverage for the tested methods.

6.3.2 Results

The data presented in Table 9 concluded that the code coverage of the units tested by the *RUTCs* generated by *OPDaTe* was satisfactory. According to Cornett [2007], the code coverage commonly accepted percentage for system testing is set between 70% and 80%. However, for unit testing, the code coverage can be set between 10% and 20% higher than system testing. The median code coverage for all Google projects that have successful coverage computation in 2015 and 2018 varied between 80 and 85% [Ivanković et al., 2019]. The tested units averaged over 90% code coverage for each participant. For better visualization, we graphically represent these results using a box plot diagram [9] (Figure 6).

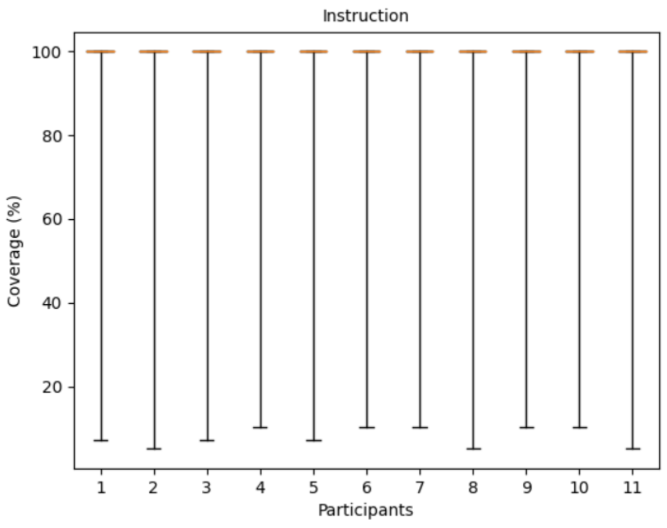


Figure 6: Methods coverage

Taking the Cornett [2007], Ivanković et al. [2019] studies as a reference, we consider the coverage obtained for the tested units to be satisfactory. The presented results provide an answer to research question RQ_2. We are aware that high levels of coverage do not necessarily indicate that a test set is effective on detecting faults [Inozemtseva and Holmes, 2014]. However, a good test set is the one with good coverage of the software parts related to the *SOP* [Cavamura Jr. et al., 2020b].

To answer research question RQ_3, we analyzed the data shown in Table 9 against the number of methods covered from the *RUTCs* generated by all participants. We found that test cases generated by participants P_{02} and P_{05} had the highest number of covered methods, resulting in 398 covered methods. In contrast, test cases generated by participant P_{10} had the fewest number of covered methods, resulting in 351 covered methods. The test cases generated by all participants had 433 covered methods, 8.8% greater than the number of methods covered by the test cases generated by participants P_{02} and P_{05} and 23% greater than the number of methods covered by the test cases generated by participant P_{10}. Figure 7 shows these results.

We also found that *RUTCs* generated by different participants can create variations in the coverage provided for the same method. The *id* 101 in Table 9 exemplifies this finding. The re-

[8]The complete table can be found at: `https://lcvm.com.br/artigos/anexos/opdate/tabela-cobertura-2022-01-18.pdf`

[9]`https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html`

Table 9: Code coverage provided by the generated test cases.

Id.	Covered Method	P_{01}	P_{02}	P_{03}	P_{04}	P_{05}	P_{06}	P_{07}	P_{08}	P_{09}	P_{10}	Average
0	ArgoVersion.init()V	100	100	100	100	100	100	100	100	100	100	100
1	LoadModules.<clinit>()V	100	100	100	100	100	100	100	100	100	100	100
2	Main.<clinit>()V	80.25	80.25	80.25	80.25	80.25	80.25	80.25	80.25	80.25	80.25	80.25
3	Main.addPostLoadAction(Ljava.lang.Runnable;)V	100	100	100	100	100	100	100	100	100	100	100
...	...	...	...	...	...	...	...	...	...	...	...	...
16	ResourceLoader.lookupIconResource (Ljava/lang/String;)Ljavax/swing/ImageIcon;	100	100	100	100	100	100	100	100	100	100	100
...	...	...	...	...	...	...	...	...	...	...	...	...
68	ToDoList.forceValidityCheck(Ljava/util/List;)V	36.47	36.47	36.47	36.47	36.47	36.47	36.47	36.47	36.47	36.47	36.47
...	...	...	...	...	...	...	...	...	...	...	...	...
101	ConfigurationHandler.saveDefault(Z)Z	57.14	57.14	35.71	71.43	0.00	16.67	0.00	57.14	57.14	30.95	
...	...	...	...	...	...	...	...	...	...	...	...	...
144	Translator.getName (Ljava/lang/String;)Ljava/lang/String;	88.24	88.24	88.24	88.24	88.24	88.24	88.24	88.24	88.24	88.24	88.24
...	...	...	...	...	...	...	...	...	...	...	...	...
283	BasicLinkButtonUI.createUI (Ljavax/swing/JComponent;) Ljavax/swing/plaf/ComponentUI;	100	100	100	100	100	100	100	100	100	100	100
...	...	...	...	...	...	...	...	...	...	...	...	...
286	ContextActionFactoryManager.getFactories ()Ljava/util/Collection;	100	0	100	0	100	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...
300	ProjectActions.jumpToDiagramShowing (Ljava/util/List;)V	0	5.08	0	0	0	0	0	5.08	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...
318	TargetManager.getInstance() Lorg/argouml/ui/targetmanager/TargetManager	100	0	100	0	100	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...
385	GeneratorHelper.generate (Lorg/argouml/uml/generator/Language; Ljava/lang/Object;Z) Ljava/util/Collection;	0	100	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...
430	AwtExceptionHandler.<clinit>()V	85.09	85.09	85.09	85.09	85.09	85.09	85.09	85.09	85.09	85.09	85.09
431	Tools.logVersionInfo()V	95	95	95	95	95	95	95	95	95	95	95
432	AwtExceptionHandler.<clinit>()V	100	100	100	100	100	100	100	100	100	100	100
433	AwtExceptionHandler.registerExceptionHandler()V	100	100	100	100	100	100	100	100	100	100	100
	Amount of covered methods	383	398	396	378	398	380	384	377	385	351	
	Average coverage	94.12	93.47	93.91	94.32	94.17	94.34	94.57	94.10	94.26	94.12	
	Minimum coverage	7.14	5.08	7.14	10.26	7.14	10.26	10.26	5.08	10.26	10.26	
	Maximum coverage	100	100	100	100	100	100	100	100	100	100	

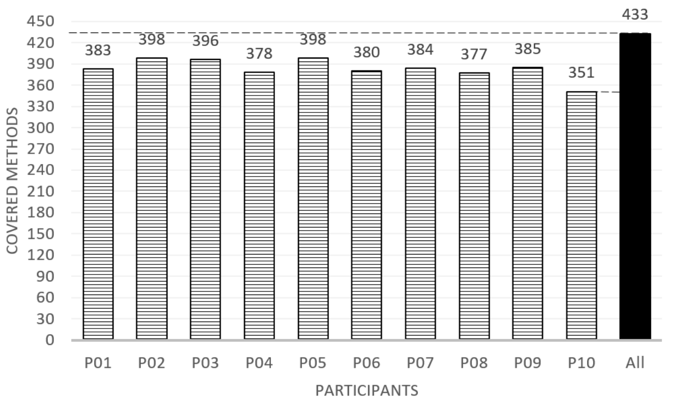


Figure 7: Number of methods covered by participants

sults shown in Figure 7 and the coverage variations for the same method by test cases generated by different participants show that a test strategy based on test suites obtained from different *SOP* contributes to increasing the number of tested units and expanding the coverage of the tested units. This evidence answers research question RQ_3.

7 Threats to Validity

We carried out the $FES\!-\!AT_1$ to obtain data that would allow us to carry out an initial evaluation of *OPDdTe*, providing answers to the defined research questions. We asked participants in the $FES\!-\!AT_1$ to perform the same task through the *SUT*. By requesting the same task from the participants, we induce the participants to similar operational profiles of the software. Using the *OPDdTe* from the *SUT* operation in a production environment, without restrictions on the operation to be performed, we would have more parts of the *SUT* tested. We are aware of this situation, but we emphasize that even inducing similar operational profiles, we obtained satisfactory results about research question RQ_3. *OPDdTe* tries to collect any data types and generate abstract test cases from them. However, just as an exhaustive test activity is impractical, it becomes difficult to predict and handle all combinations of characteristics associated with software submitted to *OPDdTe*, such as operating environments, applied technologies, and architectures. These characteristics can impact the effectiveness of *OPDdTe* in dynamically collecting data.

We performed an initial evaluation of the *OPDdTe* tool based on the results obtained when using it in a software product. However, the focus of this paper is on the "Feasibility Study" phase of the methodology applied. Our purpose was to investigate whether *OPDdTe* would generate the *ATCs* and derive the *RUTCs* from them. Then, we provided answers to the defined research questions with the generated data. Given the results presented in this paper, the "Observational Study" phase progresses, in which we are investigating the use of the tool in different software products.

8 Lessons Learned

First of all, it important to mention that *OPDdTe* is still a prototype and does not yet implements all the optimizations we intend to have.

In this way, depending on the volume of data to be collected, we have found that dynamically collecting this data can be costly for the software to operate, mainly due to the intensive input/output operations to persist the `atc` files. Even with some limitations that will be improved in a future version, by defining a strategy on which methods will be monitored makes the software operation feasible for users in a production environment. Thus, we must evaluate the cost-benefit ratio.

We also found that developing an automated tool for generating executable test cases is not a trivial task. These tools address many aspects related to software complexity, technology, and architecture that influence their generation strategies. Thus, each tool can approach the generation of executable test cases from different perspectives and principles, i.e., they have specific purposes. The results presented in our previous studies [Cavamura Jr. et al., 2020b, Cavamura et al., 2020] and the results presented in this paper show us that a strategy based on a set of tools with specific and differentiated purposes can increase the quality of the tested software.

9 Future Works

We presented in this paper the initial evaluates' results of *OPDdTe*. As described in Section 6, our strategy was to generate a set of abstract test cases ($ATCs_{dist}$) for each participant, composed by selecting an *ATC* by the method of interest from the *ATCs* generated for the respective method. Then, we generate the *RUTCs* test cases from each participant's generated abstract test cases. We are working on additional analysis to investigate the impact on code coverage provided by using different test data instances in the same *RUTC*, i.e., analyzing the results obtained by *OPDdTe*, using all *ATCs* generated for each method of interest.

We are also analyzing the failing *RUTCs* and the test cases classified as inactive by *OPDdTe*. This analysis will allow us to add knowledge to *OPDdTe* to handle these situations and to generate more effective *RUTCs*. For example, *OPDdTe Oracle* deals with the private methods specified in *ATCs*, but *RUTCs* (*JUnit*) cannot execute it. Therefore, *OPDdTe* generates unit test cases classifying them as inactive. We are working so that *OPDdTe* can also generate executable integration test cases using test data generated from dynamically collected *SOP* data. Thus, *OPDdTe* will identify the execution path and the test data needed so that the test case can exercise the private method indirectly, via a public method.

We are also working for the *OPDdTe* to generate the executable test cases online, right after the respective *ATC* is generated. The idea is that *OPDdTe* checks at *SUT* runtime if there are enough *RUTCs* for the processed method of interest. In addition, we are defining criteria such as code coverage to determine whether existing test cases are sufficient or not.

We also intend to allow the user to indicate if a given execution ends with a failure or success. Based on this information, we can figure out the set of methods involved in a failed execution and use such information to prioritize methods to be analyzed or to price a failure based on the number of affected uses which indicate it.

10 Conclusions

This paper presented the *OPDaTe* tool and the results obtained from an initial evaluation of its objectives' effectiveness. The results provided answers to the defined research questions, stating: a) *OPDaTe* was able to automatically generate executable test cases using real test data extracted from the *SOP*; b) the automatically generated runnable test cases provide the coverage for the respective tested units satisfactory; c) a test strategy based on test suites obtained from different *SOP* contribute to expanding the coverage of the tested units. The answers to the research questions provide the expected contributions to this work showing that *OPDaTe* can support to software testing activity providing: a) a set of initial executable test cases based on the *SOP* if the SUT does not have it; b) an extension of the existing test case set by adding executable test cases generated from *SOP*; c) the possibility of specifying at the method level the parts of the software for which test cases will be generated; d) support testers by providing the specification of test cases for the methods of interest.

We know that high levels of coverage do not necessarily indicate that a test set is effective on detecting faults [Inozemtseva and Holmes, 2014]. However, our previous work indicates that a good test set is the one with good coverage of the software parts related to the *SOP* [Cavamura Jr. et al., 2020b]. Therefore the *OPDaTe* provide support to build a good test set and also to reduce the possible mismatch between the *SOP* and test profile. The *OPDaTe* also contributes to a decrease the costs associated to the specification, project and development of test cases since the test cases are automatically generated and are based on real data extracted from *SOP*.

References

M. M. Ali-Shahid and S. Sulaiman. 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, apr 2015. doi: 10.1109/I4CT.2015.7219603.

Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. Encyclopedia of Software Engineering, volume 1, chapter The Goal Question Metric Approach, pages 528–532. John Wiley Sons, 2002.

Sergio Bittanti, Paolo Bolzern, and Riccardo Scattolini. An introduction to software reliability modelling, chapter 12, pages 43–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988. ISBN 978-3-540-46072-5.

Luiz Jr. Cavamura, Anderson Belgamo, Vinícius Rafael Lobo de Mendonça, and Auri Marcelo Rizzo Vincenzi. Warningsfix: A recommendation system for prioritizing warnings generated by automated static analyzers. In 19th Brazilian Symposium on Software Quality, SBQS'20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450389235. doi: 10.1145/3439961.3439987.

Luiz Cavamura Jr., Sandra Fabbri, and Auri M. R. Vincenzi. Perfil operacional do software: investigando aplicabilidades específicas. In Claudia P. Ayala, Leonardo Murta, Daniela Soares Cruzes, Eduardo Figueiredo, Carla Silva, Jose Luis de la Vara, Breno de França, Martín Solari, Guilherme Horta Travassos, and Ivan Machado, editors, Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020, Curitiba, Paraná, Brazil, November 9-13, 2020, pages 292–305. Curran Associates, 2020a. ISBN 978-1-7138-1853-3.

Luiz Cavamura Jr., Ricardo Morimoto, Sandra Fabbri, Ana C. R. Paiva, and Auri Marcelo Rizzo Vincenzi. Software operational profile vs. test profile: Towards a better software testing strategy. Journal of Software Engineering Research and Development, 8:5:1 – 5:17, Aug. 2020b. doi: 10.5753/jserd.2020.546.

Steve Cornett. Minimum acceptable code coverage. https://www.bullseye.com/minimum.html, 2007. Accessed: 2022-01-13.

Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. Journal of Systems and Software, 157: 110398, 2019a. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110398. URL `https://www.sciencedirect.com/science/article/pii/S0164121219301736`.

Benjamin Danglot, Oscar Vera Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. Empirical Software Engineering, 24, 08 2019b. doi: 10.1007/s10664-019-09692-y.

Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. 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 830–840, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338945. URL `https://doi.org/10.1145/3338906.3338945`.

Wei He, Jianyang Ding, Xiaomei Shen, Xinyu Han, and Longli Tang. A survey on software reliability demonstration. IOP Conference Series: Materials Science and Engineering, 1043(3):032008, January 2021. doi: 10.1088/1757-899x/1043/3/032008. URL `https://doi.org/10.1088/1757-899x/1043/3/032008`.

Laura Inozemtseva and Reid Holmes. 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, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568271. URL `https://doi.org/10.1145/2568225.2568271`.

Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 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, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3340459. URL `https://doi.org/10.1145/3338906.3340459`.

Abhinav Kashyap. 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, 2013.

D. Marijan, N. Teslic, T. Tekcan, and V. Pekovic. Multimedia system verification through a usage model and a black test box. In The 2010 International Conference on Computer Engineering Systems, pages 178–182, Pasadena, CA, United states, Nov 2010. IEEE.

J. D. Musa. Software reliability measures applied to systems engineering. In Managing Requirements Knowledge, International Workshop on(AFIPS), volume 00, page 941, S.I., 12 1979. IEEE.

J.D. Musa. Operational profiles in software-reliability engineering. IEEE Software, 10(2): 14–32, 1993. ISSN 07407459. cited By 396.

J.D. Musa and W. Ehrlich. Advances in software reliability engineering. Advances in Computers, 42(C):77–117, 1996. ISSN 00652458. cited By 1.

Glenford J. Myers, Corey Sandler, and Tom Badgett. The Art of Software Testing. Wiley Publishing, 3rd edition, 2011. ISBN 1118031962, 9781118031964.

S. Nakornburi and T. Suwannasart. A tool for constrained pairwise test case generation using statistical user profile based prioritization. In 2016 13th International Joint Conference on Computer Science and Software Engineering, JCSSE 2016, pages 1–6, White Plains, NY, USA, 2016. Institute of Electrical and Electronics Engineers Inc. ISBN 9781509020331.

Erik Rogstad and Lionel Briand. Cost-effective strategies for the regression testing of database applications: Case study and lessons learned. Journal of Systems and Software, 113: 257–274, 2016. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.12.003. URL `https://www.sciencedirect.com/science/article/pii/S0164121215002794`.

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

Forrest Shull, Jeffrey Carver, and Guilherme H. Travassos. 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, 2001. Association for Computing Machinery. ISBN 1581133901. doi: 10.1145/503209.503248. URL `https://doi.org/10.1145/503209.503248`.

Ian Sommerville. Software Engineering. Addison-Wesley, Harlow, England, 9 edition, 2011. ISBN 978-0-13-703515-1.

Tomohiko Takagi, Zengo Furukawa, and Toshinori Yamasaki. 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, 2007.

Auri Marcelo Rizzo Vincenzi, Márcio Eduardo Delamaro, Arilo Claudio Dias Neto, Sandra Camargo Pinto Ferraz Fabbri, Mário Jino, and José Carlos Maldonado. Automatização de teste de software com ferramentas de software livre. Elsevier, 2018.

Olzhas Zhakipbayev and Aisulu Bekey. Effectiveness of operational profile-based testing, 2021.

Sai Zhang. Palus: a hybrid automated test generation tool for java. In 2011 33rd International Conference on Software Engineering (ICSE), pages 1182–1184, 2011. doi: 10.1145/1985793.1986036.

Capítulo 7

Conclusão, Trabalhos Futuros e Lições Aprendidas

Este capítulo apresenta as conclusões obtidas com esta tese e ressalta uma lista de trabalhos futuros que ainda podem ser desenvolvidos a partir da mesma. As lições aprendidas também são apresentadas neste capítulo.

7.1 Conclusão

O problema de pesquisa abordado nesta tese refere-se ao descompasso entre o *POS* e as partes testadas do software. A pesquisa realizada obteve evidências que permitem concluir que:

a) há variações significativas na maneira como os usuários operam o software mesmo quando realizam uma mesma operação;

b) existe um descompasso entre o *POS* e o *perfil de teste* e que falhas podem ocorrer nas partes do *POS* não testadas;

c) uma estratégia de teste baseada na união de um conjunto de teste existente com um conjunto de teste gerado automaticamente por uma ferramenta geradora de dados de teste diminuiu o possível descompasso entre o *POS* e as partes testadas do software, porém, não foi capaz de evitar o descompasso;

d) a ferramenta *OPDaTe* permite, por meio da sua parametrização, identificar, em nível de método, a frequência de execução das partes do software discriminadas na sua parametrização para serem monitoradas e terem casos de teste abstratos gerados;

e) a ferramenta *OPDaTe* pode contribuir para diminuir o descompasso pois gera casos de teste de unidade executáveis, em nível de método, para as partes do software discriminadas na parametrização da ferramenta, as quais podem corresponder às partes do *POS* não testadas.

Por meio dos resultados obtidos pelo mapeamento sistemático e pela revisão sistemática da literatura, verificou-se que o *POS* é pouco explorado em outras áreas de pesquisa da engenharia de software. Assim, o *POS* pode ser explorado pelos pesquisadores para prover contribuições à essas áreas de pesquisa.

Além das evidências obtidas sobre o problema de pesquisa tratado nesta tese, são também contribuições desta pesquisa:

a) mapeamento sistemático e revisão sistemática da literatura que investigam o uso do Perfil Operacional do Software;

b) projeto e implementação da ferramenta *OPDaTe* juntamente com uma avaliação inicial referente à sua eficácia.

Os resultados obtidos pela pesquisa e as respectivas contribuições geradas evidenciam a relevância do *POS* ao teste de software, independentemente da estratégia de teste adotada. Assim, o *POS* alinha essas estratégias ao uso operacional do software, ou seja, faz com que estejam em consonância com as necessidades dos usuários.

7.2 Trabalhos Futuros

A partir das evidências e resultados apresentados nesta tese por meio das publicações realizadas, foi possível identificar as seguintes propostas de pesquisa diretamente relacionadas ao tema abordado pela tese:

1. Investigar o uso do *POS* como um critério de priorização para que, dado um conjunto de falhas, permitir identificar aquelas que causam o impacto mais significativo na experiência dos usuários ao operar o software, e assim considerar tal impacto na precificação da correção dessas falhas.

2. Investigar se há variações significativas nos resultados providos por um mesmo conjunto de casos de teste quando diferentes instâncias dos dados de teste são obtidas de diferentes *POS*, permitindo, assim, avaliar se diferentes dados de teste extraídos diretamente da operação do software pelos usuários podem prover variações significativas nos resultados fornecidos por um mesmo conjunto de casos de teste.

3. Por meio da ferramenta *WarningsFix* (CAVAMURA et al., 2020), investigar os tipos de *warnings* ou combinações de tipos de *warnings* que podem estabelecer diretrizes para realizar a análise dinâmica, ou seja, compor um critério de priorização para realizar as atividades de teste propondo uma estratégia que combine a análise estática e dinâmica e, assim, verificar se pode existir uma relação entre o *POS* e o *perfil de teste* com os tipos de *warnings* ou combinações de tipos de *warnings*.

Novas funcionalidades estão sendo analisadas e especificadas para serem integradas à ferramenta *OPDaTe*ermitindo que a ferramenta possa:

1. *Gerar dos casos de teste executáveis durante a execução do software sob teste, logo após o respectivo caso de teste abstrato ser gerado.* A ideia é que a ferramenta *OPDaTe*ere os casos de teste executáveis enquanto não tenham sido gerados casos de teste suficientes para o método de interesse monitorado. Os critérios que determinam se os casos de teste gerados são suficientes, como por exemplo a cobertura de código provida por eles, estão sendo analisados.

2. *Gerar de casos de teste de integração executáveis.* A ferramenta irá identificar caminhos de execução e dados de teste que permitam gerar casos de teste executáveis que consigam exercitar métodos de interesse cujo escopo é privado.

3. *Melhorar a eficiência, precisão e capacidade de coleta de dados do perfil operacional.* As bibliotecas utilizadas para monitoramento do software e que viabilizam a coleta dos dados do perfil operacional apresentam limitações que, acredita-se, possam ser superadas. Para isso, com base nos dados coletados dos tipos de erros identificados, pretende-se realizar uma análise qualitativa das bibliotecas sugerindo ou até mesmo implementando melhorias.

4. *Oferecer uma estratégia de minimização de conjunto de teste com base no POS.* Diversas execuções do software por parte de um mesmo usuário ou de usuários com perfis operacionais semelhantes podem produzir uma grande quantidade de casos de teste referentes à mesma área do produto de software e, desse modo, estratégias de minimização de conjuntos de teste podem ser aplicadas para selecionar um subconjunto dos testes mantendo as características fundamentais do *POS* desejado.

7.3 Lições aprendidas

Por ser decorrente da manifestação do caráter criativo do intelecto humano (ASSESC, 2012), o software é mutável e evolutivo. Adicionalmente, o software pode ser desenvolvido sob os mais variados padrões de projeto, arquiteturas, tecnologias e plataformas. Essas características, além das limitações de tecnologia inerentes ao uso da meta programação, tornam o desenvolvimento de ferramentas computacionais, capazes de gerar programas executáveis e coletar dados dinamicamente de outros softwares, uma atividade desafiadora uma vez que softwares desenvolvidos sob as mais variadas características podem ser submetidos à essas ferramentas. Assim, o desenvolvimento de ferramentas computacionais não é trivial.

Os experimentos, realizados durante a pesquisa, forneceram indícios que tais ferramentas possuem características e abordagens específicas. Com base nesses indícios, é possível que uma estratégia baseada no uso de diferentes ferramentas computacionais, que tenham um mesmo propósito, possa estender os resultados obtidos por cada ferramenta individualmente.

Cada resultado pode prover um complemento aos demais resultados, gerando um conjunto resultante maior que cada resultado individualmente. Embora essa estratégia possa prover contribuições, uma análise da viabilidade do uso de diferentes ferramentas é recomendada, uma vez que essas ferramentas podem gerar também resultados redundantes, demandar tempo de processamento e requerer esforço da equipe de desenvolvimento referente ao estudo e treinamento dessas ferramentas.

Sobre o *POS*, a identificação do *POS* e a coleta de dados dinamicamente por meio de instrumentação pode afetar o desempenho do software e produzir um grande volume de dados dependendo do nível de fragmentação adotado. No entanto, as informações obtidas sobre o *POS* podem contribuir para melhorar o teste de software independentemente da estratégia adotada.

Referências

ALI-SHAHID, M. b.; SULAIMAN, S. Improving reliability using software operational profile and testing profile. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2015. p. 384–388. ISBN 9781479979523.

ASSESC, F. F. **Propriedade Intelectual e Software - Cursos de Mídia Eletrônica e Sistema de Informação**. 2012.

BEGEL, A.; ZIMMERMANN, T. Analyze this! 145 questions for data scientists in software engineering. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 12–23. ISBN 978-1-4503-2756-5.

BERTOLINO, A. et al. Adaptive coverage and operational profile-based testing for reliability improvement. In: **Proceedings of the 39th International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE '17), p. 541–551. ISBN 978-1-5386-3868-2.

BITTANTI, S.; BOLZERN, P.; SCATTOLINI, R. An introduction to software reliability modelling. In: _____. **Software Reliability Modelling and Identification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988. cap. 12, p. 43–67. ISBN 978-3-540-46072-5.

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 34, n. 1, p. 135–137, jan. 2001. ISSN 0018-9162.

CAVAMURA, L. J. Operational profile and software testing: Aligning user interest and test strategy. In: **Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST) – Doctoral Symposium**. [S.l.: s.n.], 2019. p. 492–494.

CAVAMURA, L. J. et al. Warningsfix: A recommendation system for prioritizing warnings generated by automated static analyzers. In: **19th Brazilian Symposium on Software Quality**. New York, NY, USA: Association for Computing Machinery, 2020. (SBQS'20). ISBN 9781450389235.

_____. OPDaTe: A unit test case generator based on real data from software operating profile. ——, —, x, p. xxx–xxx(x), jan. 2022. ISSN -. Artigo submetido para avaliação.

CAVAMURA, L. J.; FABBRI, S.; VINCENZI, A. M. R. Perfil operacional do software: investigando aplicabilidades específicas. In: AYALA, C. P. et al. (Ed.). **Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020, Curitiba, Paraná, Brazil, November 9-13, 2020**. [S.l.]: Curran Associates, 2020. p. 292–305.

CAVAMURA, L. J. et al. Software operational profile vs. test profile. In: **Proceedings of the XVIII Brazilian Symposium on Software Quality**. New York, NY, USA: Association for Computing Machinery, 2019. (SBQS'19), p. 139–148. ISBN 9781450372824.

_____. Software operational profile vs. test profile: Towards a better software testing strategy. **Journal of Software Engineering Research and Development**, v. 8, p. 5:1 – 5:17, ago. 2020.

CUKIC, B.; BASTANI, F. B. On reducing the sensitivity of software reliability to variations in the operational profile. In: ANON (Ed.). White Plains, NY, USA: IEEE, Los Alamitos, CA, United States, 1996. p. 45–54. ISSN 10719458.

FALBO, R. A. **Engenharia de Software**. Vitória: [s.n.], 2005. 145 p.

FUKUTAKE, H. et al. The method to create test suite based on operational profiles for combination test of status. In: K., S. (Ed.). White Plains, NY, USA: Institute of Electrical and Electronics Engineers Inc., 2015. p. 1–4. ISBN 9781479986767.

GITTENS, M.; LUTFIYYA, H.; BAUER, M. An extended operational profile model. In: **Proceedings - International Symposium on Software Reliability Engineering, ISSRE**. Saint-Malo, France: IEEE, 2004. p. 314 – 325. ISSN 10719458.

HE, W. et al. A survey on software reliability demonstration. **IOP Conference Series: Materials Science and Engineering**, IOP Publishing, v. 1043, n. 3, p. 032008, jan. 2021. Disponível em: <https://doi.org/10.1088/1757-899x/1043/3/032008>.

KASHYAP, A. **A Markov Chain and Likelihood-Based Model Approach for Automated Test Case Generation, Validation and Prioritization: Theory and Application**. 106 p. Tese (Doutorado), 2013.

KOSCIANSKI, A. **Qualidade de Software**. São Paulo: Novatec Editora, 2006. 395 p. ISBN 85-7522-085-3.

LEUNG, Y.-W. Software reliability allocation under an uncertain operational profile. **Journal of the Operational Research Society**, Palgrave Macmillan UK, [S.l.], v. 48, n. 4, p. 401 – 411, 1997. ISSN 01605682.

LI, Q.; LUO, L.; WANG, J. Accelerated reliability testing approach for high-reliablity software based on the reinforced operational profile. In: . Pasadena, CA, United states: IEEE, 2013. p. 337 – 342.

MUSA, J. Operational profiles in software-reliability engineering. **IEEE Software**, Los Alamitos, CA, EUA, v. 10, n. 2, p. 14–32, 1993. ISSN 07407459.

MUSA, J. D. Software reliability measures applied to systems engineering. In: **Managing Requirements Knowledge, International Workshop on(AFIPS)**. S.I.: IEEE, 1979. v. 00, p. 941.

______. Adjusting measured field failure intensity for operational profile variation. In: . Monterey, CA, USA: IEEE, Los Alamitos, CA, United States, 1994. p. 330–333. ISSN 10719458.

MUSA, J. D.; EHRLICH, W. Advances in software reliability engineering. In: ZELKOWITZ, M. V. (Ed.). [S.l.]: Elsevier, 1996, (Advances in Computers, v. 42). p. 77 – 117.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. Hoboken, New Jersey: John Wiley & Sons, Inc, 2012. 240 p. ISBN 1118031962, 9781118031964.

NAMBA, Y.; AKIMOTO, S.; TAKAGI, T. Overview of graphical operational profiles for generating test cases of gui software. In: K., S. (Ed.). White Plains, NY, USA: Institute of Electrical and Electronics Engineers Inc., 2015. p. 1–3. ISBN 9781479986767.

POORE, J.; WALTON, G.; WHITTAKER, J. A constraint-based approach to the representation of software usage models. **Information and Software Technology**, ScienceDirect, [S.l.], v. 42, n. 12, p. 825 – 833, 2000. ISSN 0950-5849.

PRESSMAN, R. S. **Software Engineering A Practitioner's Approach**. 9. ed. New York, NY: McGraw-Hill, 2019. 704 p. ISBN 978-1-25-987297-6.

RINCON, A. M. **Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais**. Dissertação (Mestrado), 2011.

SHUKLA, R. Deriving parameter characteristics. In: . New York, NY, USA: ACM, 2009. p. 57 – 63.

SMIDTS, C. et al. Software testing with an operational profile: Op definition. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 46, n. 3, p. 39:1–39:39, fev. 2014. ISSN 0360-0300.

SOMMERVILLE, I. **Software Engineering**. 10. ed. Harlow, England: Pearson, 2015. 816 p. ISBN 978-0-13-394303-0.

TAKAGI, T.; FURUKAWA, Z.; YAMASAKI, T. An overview and case study of a statistical regression testing method for software maintenance. **Electronics and Communications in Japan (Part II: Electronics)**, Wiley Online Library, [S.l.], v. 90, n. 12, p. 23–34, 2007.

TRICENTIS. **Software Fail Watch**. 2018. Disponível em: <https://www.tricentis.com/software-fail-watch/>.

ZHAKIPBAYEV, O.; BEKEY, A. **Effectiveness of operational profile-based testing**. 2021. 40 p.