

PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

<https://pgcomp.ufba.br>
pgcomp@ufba.br

Although Software Product Lines (SPL) offer the potential for order-of-magnitude improvements in software engineering performance, the up-front cost, level of effort, assumed risk, and latency required to make the transition to SPL are prohibitive adoption barriers for many organizations that could otherwise benefit from reusing of their existing systems. The SPL adoption from an extractive model based on a reengineering process of existing systems into SPL is an active research topic with real benefits in practice. It allows software development companies to preserve their investment and aggregate knowledge obtained during the development of their portfolio of systems individually developed. Despite these benefits, adopting an extractive product line approach still requires a considerable up-front investment and is more complex to evolve than single systems. Because of these drawbacks, software companies refrain or delay the adoption of SPL, resorting to an ad-hoc practice of clone-and-own. To speed conversion to and maintenance of SPL, we present Foundry, a Software Transplantation (ST) approach that guides the process of transplanting and merging features in a product line from existing systems. It is the first approach for SPL that automates all stages of product line construction using the ST technique. We realized Foundry in prodScalpel, a software transplantation tool for SPL that automates the process of identifying, adapting and transferring features from existing systems to a common product base. Its code transfer mechanism between different systems allows it to be used not only for the generation of product lines but also as an alternative to the clone-and-own technique for system specialization. We compared our proposal with the existing reengineering solutions to demonstrate evidence that the ST is an alternative with potential for application in the field of reengineering of existing systems to SPL. In the search for more concrete evidence, we evaluated prodScalpel on two case studies where two products were generated by transplanting of features from three real-world systems. Moreover, we conducted an experiment comparing Foundry's feature migration with manual effort. We show that Foundry automatically migrated features across codebases 4.8 times faster, on average, than the average time a group of SPL experts took to accomplish the task. Although preliminary, our evaluation provides evidence to support the claim that ST for Software Product Line Engineering (SPLE) is a feasible and, indeed, promising new research direction.

DSC | 042 | 2023

An automated software transplantation approach
for reengineering of systems into product lines

Leandro Oliveira de Souza

An automated software transplantation approach for reengineering of systems into product lines

Leandro Oliveira de Souza

Tese de Doutorado

Universidade Federal da Bahia
Programa de Pós-Graduação em
Ciência da Computação

Junho | 2023

UFBA





Universidade Federal da Bahia
Instituto de Matemática e Estatística

Programa de Pós-Graduação em Ciência da Computação

**AN AUTOMATED SOFTWARE
TRANSPLANTATION APPROACH FOR
REENGINEERING OF SYSTEMS INTO
PRODUCT LINES**

Leandro Oliveira de Souza

TESE DE DOUTORADO

Salvador, Bahia – Brasil
30 de Junho de 2023

LEANDRO OLIVEIRA DE SOUZA

**AN AUTOMATED SOFTWARE TRANSPLANTATION
APPROACH FOR REENGINEERING OF SYSTEMS INTO
PRODUCT LINES**

Esta Tese de Doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Eduardo Santana de Almeida
Co-orientador: Prof. Dr. Earl Barr e Dr. Justyna Petke

Salvador, Bahia – Brasil
30 de Junho de 2023

Ficha catalográfica elaborada pela Biblioteca Universitária de
Ciências e Tecnologias Prof. Omar Catunda, SIBI - UFBA.

S729 Souza, Leandro Oliveira de
An automated software transplantation approach for
reengineering of systems into product line/ Leandro Oliveira de
Souza. – Salvador, 2023.

190 f.

Orientadora: Prof. Dr. Eduardo Santana de Almeida

Tese (Doutorado) – Universidade Federal da Bahia.
Instituto de Computação, 2023.

1. Computação. 2. Software. 3. Engenharia de Software. I.
Almeida, Eduardo Santana de. II. Universidade Federal da
Bahia. III. Título.

CDU 004

Leandro Oliveira de Souza

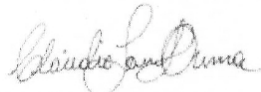
**An Automated Software Transplantation Approach For Reengineering Of
Systems Into Product Lines**

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da UFBA.

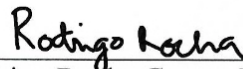
Salvador, 30 de junho de 2023



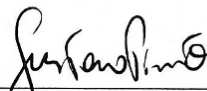
Prof. Dr. Eduardo Santana De Almeida (Orientador-UFBA)



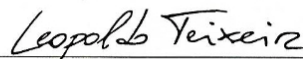
Prof. Dr. Claudio Nogueira Sant Anna (UFBA)



Prof. Dr. Rodrigo Rocha Gomes E Souza (UFBA)



Prof. Dr. Gustavo Henrique Lima Pinto (UFPA)



Prof. Dr. Leopoldo Motta Teixeira (UFPE)

"Dedico esta tese à memria da minha querida av Odete Guilhermina, que nos deixou há 5 anos. Agradeço-lhe imensamente por todo o amor, incentivo e apoio que sempre me ofereceu. Tenho a esperança de reencontrá-la um dia."

ACKNOWLEDGEMENTS

During my time pursuing my PhD, I had the privilege of being welcomed into the esteemed academic community at University College London (UCL), UK, by Professor Dr. Mark Harman, Dr. Earl Barr, and Dr. Justyna Petke. Their guidance, expertise, and mentorship have shaped me profoundly, both professionally and personally. UCL has been a transformative experience for me, and I will always cherish the knowledge and growth it has bestowed upon me.

I would also like to extend my deepest appreciation to Earl and Justyna, who have been incredible collaborators since my arrival in the UK. Their generosity, kindness, and partnership have been instrumental in my research endeavors. Furthermore, I would like to express my gratitude to the CREST research group, particularly my friend Chaiyong Ragkhitwetsagul, Alexandru Marginean, Giovanni Guizzo, Matheus Paixão e Carlos Gavidia, and Profir-Petru Pârtachi for the countless discussions, unwavering support during challenging times, shared laughter, and genuine friendship.

Lastly, but certainly not least, I want to express my heartfelt thanks to all my friends in London, with a special mention to Elizabeth Versari. Their presence and companionship was invaluable, making my adaptation to London life much smoother and more enjoyable. I am truly grateful for their friendship and the memories we have created together.

Agradeço a Deus do fundo do meu coração por tudo o que tens feito por mim. Minha fé em ti tem me feito galgar percursos ainda maiores e superar todos os obstáculos em minha vida. Obrigado por me guiar em cada passo do caminho.

Tenho que destacar que durante os anos do meu doutorado, enfrentei algumas das batalhas mais intensas da minha vida, perdi uma pessoa muito especial, minha querida avó Odete, à qual dedico este doutoramento. Superei também dois anos de pandemia, enfrentei sérios problemas de saúde e estive à beira de perder meu pai. Foram momentos tão desafiadores que precisei desacelerar por alguns meses e considerei desistir várias vezes. No entanto, também vivi momentos inesquecíveis e conheci pessoal as quais pretendo estar junto durante toda a vida.

No entanto, Deus me concedeu a força necessária para perseverar e superar os obstáculos que surgiram em meu caminho. Ao olhar para trás, percebo que cada luta valeu a pena. Foi no meio dessas dificuldades que encontrei momentos inesquecíveis, fiz grandes avanços como pesquisador e professor. Mas sei que não cheguei até aqui sozinho. Conteí com a ajuda de pessoas incríveis em cada etapa da minha jornada.

Gostaria de expressar minha sincera gratidão ao meu orientador no Brasil, o professor Dr. Eduardo Almeida. Obrigado por acreditar em mim e confiar em minha capacidade. Também agradeço a Paulo, Michele, Stefani, Tassio, Thiago, Iuri, Jaziel e Eduardo Lopes pelo apoio nesta reta final. Muito obrigado.

Quero agradecer de todo coração aos meus amigos e familiares, em especial à minha esposa Dayse, por apoiar-me desde o início desta minha desafiadora jornada acadêmica. Agradeço aos meus filhos Eloah e Leonardo por darem um novo sentido à minha vida, e peço desculpas por estar distante e não poder compartilhar todos os momentos felizes da vida de vocês durante esse período de estudos. Agradeço a minha mãe, Maria de Fátima, pelas orações e suporte. Do mesmo modo ao meu pai, Orlando Oliveira.

Agradeço aos meus amigos do apartamento 203 em especial Thiago, Jonatas, Jaziel, Tassio, Iuri, Marcos, Iuri e Josualdo, que compartilharam não apenas o mesmo espaço, mas também momentos de felicidade e dificuldade durante o doutorado.

Meus agradecimentos também vão para meus colegas de pesquisa e orientadores, que me ajudaram a crescer academicamente e me motivaram a perseguir em frente. Sou grato aos meus amigos mais próximos por me proporcionarem distrações e momentos felizes durante esse período.

Gostaria de expressar minha profunda gratidão ao incrível grupo de pesquisa e laboratório, RiSE Labs e Lab INES, que, sem dúvida, são e serão ótimos profissionais! A presença diária, o ambiente descontraído e os momentos compartilhados com essas colegas incríveis foram fundamentais nessa jornada de pesquisa. Agradeço de coração a cada um deles: Thiago, Jonatas, Jaziel, Tassio, Iuri, Magno, Alcemir, Larissa, Gau, Michele, Paulo, Renata, Rose, Tajara, Alberto e Crescêncio.

Quero também agradecer aos meus colegas de trabalho pelo suporte que a mim foi dado neste período de afastamento e retorno ao trabalho. Dentre eles destaco: Jeime, Robério, Rogério, Rafael, Gustavo, Cassio, Estéfani, Luciano, Tiago e Paulo.

RESUMO

Embora as Linha de Produto de Software (LPS) ofereçam o potencial de melhorias de ordem de magnitude no desempenho da engenharia de software, custo inicial, nível de esforço, risco assumido e a latência necessários para fazer a transição para a LPS são barreiras proibitivas à adoção por muitas organizações que poderiam se beneficiar da reutilização de seus sistemas existentes. A adoção de uma estratégia extrativa a partir de um processo de reengenharia de sistemas existentes em um LPS é um tópico de pesquisa ativo com benefícios reais na prática. Ela permite que as empresas de desenvolvimento de software preservem seus investimentos e agreguem o conhecimento obtido durante o desenvolvimento de seu portfólio de sistemas desenvolvidos individualmente. Apesar desses benefícios, adotar uma abordagem extrativa de adoção de linha de produtos ainda requer um investimento inicial considerável e é mais complexo de evoluir do que sistemas únicos. Por causa dessas desvantagens, as empresas de software evitam ou atrasam a adoção de LPS, recorrendo a uma prática ad-hoc de clonagem de código. Para acelerar a conversão e a manutenção de SPL, apresentamos o FOUNDRY, uma abordagem de transplante de software (TS) que orienta o processo de transplante e mesclagem de recursos de diferentes sistemas em uma linha de produtos. É a primeira abordagem com suporte ferramental para LPS que automatiza todos os estágios da construção de uma linha de produtos utilizando a técnica de TS. Automatizamos FOUNDRY no PRODSALPEL, uma ferramenta de TS para LPS que automatiza o processo de identificação, adaptação e transferência de recursos de sistemas existentes para uma base comum de produtos. Seu mecanismo de transferência de código entre distintos sistemas permite que a mesma seja utilizada não somente para a geração de linhas de produto, mas também como alternativa à técnica de clonagem de código para a especialização de sistemas. Comparamos nossa proposta com as soluções existentes a fim de demonstrarmos evidências de que o TS é uma alternativa com potencial de aplicação no campo de reengenharia de sistemas em LPS. Na busca por evidências mais concretas, avaliamos FOUNDRY em dois estudos de caso em que dois produtos foram criados a partir do transplante de código de três sistemas do mundo real. Além disso, conduzimos um experimento comparando a migração automática de recursos utilizando FOUNDRY com esforço manual realizado por especialistas em LPS. Mostramos que o FOUNDRY migrou automaticamente recursos entre bases de código 4,8 vezes mais rápido, em média, do que o tempo médio que um grupo de participantes levou para realizar a tarefa. Embora preliminar, nossa avaliação fornece evidências para apoiar a afirmação de que TS para Engenharia de Linha de Produto de Software (ELPS) é uma direção de pesquisa nova, promissora, e viável.

Palavras-chave: Transplante de software; Engenharia de software baseada em busca; Melhoramento genético; Programação genética; Linhas de produtos de software; Reuso

de software; Reengenharia de sistemas em linhas de produto de software;

ABSTRACT

Although Software Product Lines (SPL) offer the potential for order-of-magnitude improvements in software engineering performance, the up-front cost, level of effort, assumed risk, and latency required to make the transition to SPL are prohibitive adoption barriers for many organizations that could otherwise benefit from reusing of their existing systems. The SPL adoption from an extractive model based on a reengineering process of existing systems into SPL is an active research topic with real benefits in practice. It allows software development companies to preserve their investment and aggregate knowledge obtained during the development of their portfolio of systems individually developed. Despite these benefits, adopting an extractive product line approach still requires a considerable upfront investment and is more complex to evolve than single systems. Because of these drawbacks, software companies refrain or delay the adoption of SPL, resorting to an ad-hoc practice of clone-and-own. To speed conversion to and maintenance of SPL, we present FOUNDRY, a Software Transplantation (ST) approach that guides the process of transplanting and merging features in a product line from existing systems. It is the first approach for SPL that automates all stages of product line construction using the ST technique. We realized FOUNDRY in prodscalpel, a software transplantation tool for SPL that automates the process of identifying, adapting and transferring features from existing systems to a common product base. Its code transfer mechanism between different systems allows it to be used not only for the generation of product lines but also as an alternative to the clone-and-own technique for system specialization. We compared our proposal with the existing reengineering solutions to demonstrate evidence that the ST is an alternative with potential for application in the field of reengineering of existing systems to SPL. In the search for more concrete evidence, we evaluated prodscalpel on two case studies where two products were generated by transplanting of features from three real-world systems. Moreover, we conducted an experiment comparing FOUNDRY's feature migration with manual effort. We show that FOUNDRY automatically migrated features across codebases 4.8 times faster, on average, than the average time a group of SPL experts took to accomplish the task. Although preliminary, our evaluation provides evidence to support the claim that ST for Software Product Line Engineering (SPLE) is a feasible and, indeed, promising new research direction.

Keywords: Automated software transplantation; Autotransplantation; Search-based Software Engineering; Genetic improvement; Software product lines; Software reuse; Reengineering of systems into software product lines;

CONTENTS

List of Figures	xvi
List of Tables	xvii
List of Acronyms	xix

I Overview

Chapter 1—Introduction	3
1.1 Motivation	5
1.2 Problem Definition	6
1.3 Objectives	7
1.4 Research Questions	7
1.5 Research Design	8
1.6 Statement of the Contributions	10
1.7 Out of Scope	10
1.8 Organization of the Thesis	11

II Background

Chapter 2—Main Concepts and Foundations on Software Product Lines (SPL) and Reengineering of Systems into SPL	15
2.1 Software Product Line Engineering (SPLE)	15
2.1.1 Software Product Line Essential Activities	16
2.1.2 Strategies for Adopting Software Product Line	17
2.2 Reengineering of Systems into SPL	18
2.2.1 Reengineering process	19
2.2.2 Strategies to Perform the Reengineering	20
2.2.3 Input and Output Artefacts	21
2.2.4 Research Gaps and Limitations of Reengineering Approaches	22
2.3 Chapter Summary	22

Chapter 3—An Overview of Software Transplantation	25
3.1 Search Based Software Engineering (SBSE)	26
3.2 Software Transplantation	26
3.3 Automated Software Transplantation Process	28
3.3.1 Terminology	29
3.3.2 Challenges to Automate the Software Transplantation Process	29
3.3.3 Program Slicing	31
3.3.4 Genetic Programming (Genetic Programming (GP))	31
3.3.5 Using Program Slicing and GP Techniques for Automating Software Transplantation (ST) Process	34
3.3.6 Software Transplantation Experiences	35
3.4 Related Work	36
3.4.1 Re-engineering of Software Systems into SPL	36
3.4.2 Clone-and-own	37
3.4.3 Variability in SPL	37
3.4.4 Software Transplantation	38
3.5 Chapter Summary	39

III Approach

Chapter 4—Software Transplantation Approach for SPLE	43
4.1 FOUNDRY for Software Reuse	43
4.2 Motivating Example	44
4.3 The FOUNDRY Main Concept	45
4.4 FOUNDRY	46
4.4.1 Domain Engineering	47
4.4.1.1 Variability Analysis.	48
4.4.1.2 Donor Preparation.	49
4.4.1.3 Organ Test Suite Preparation.	50
4.4.1.4 Host Preparation.	50
4.4.1.5 Over-organ Extraction.	51
4.4.2 Application Engineering	52
4.4.2.1 Over-organ Selection.	53
4.4.2.2 Over-organ Reduction and Adaptation.	53
4.4.2.3 Organ Implantation.	54
4.4.2.4 Postoperative Stage.	57
4.5 Chapter Summary	59
Chapter 5—Implementation	61
5.1 PRODSICALPEL	62
5.1.1 Automating Feature Removal	63

5.1.2	Automating Vein and Over-Organ Extraction	64
5.1.3	Automating Over-organ Reduction and Adaptation	67
5.1.4	Automating Multiple Organs Implantation	68
5.2	Automated support for SPLE	69
5.2.1	Automating re-engineering of existing systems into SPL.	70
5.2.2	Automating clone-and-own technique.	70
5.2.3	Automating Reactive Product Line Adoption Process	71
5.2.4	Automating a <i>Symbiotic SPL</i>	71
5.2.5	Supporting Controlled Maintenance and evolution of SPL	72
5.2.6	Providing a Variability Mechanism through Organ Transplantation	73
5.3	Chapter Summary	73

IV Empirical studies

Chapter 6—Comparative Study on Automated SPL Reengineering Practices via ST 77

6.1	The Comparative Study	77
6.1.1	Objective and Research Questions	79
6.1.2	Selection Criteria	79
6.2	Tool Support for Reengineering of Systems into SPL	80
6.3	Results and Discussion	81
6.4	Chapter Summary	84

Chapter 7—The Case Studies 85

7.1	Case Studies Design	85
7.1.1	Objective and Research Questions	87
7.1.2	Subject Selection	88
7.1.3	Procedure and Execution	89
7.2	Results and Discussion	90
7.2.1	Results	90
7.2.2	Discussion	92
7.2.3	Threats to Validity	92
7.3	Chapter Summary	93

Chapter 8—Experimental Evaluation 95

8.1	Experiment design	95
8.1.1	Goal	95
8.1.2	Research Questions	96
8.1.3	Metrics	96
8.1.4	Instrumentation	96
8.1.5	Hypotheses	97
8.1.6	Methodology	98

8.1.7	Participants	98
8.1.8	Operation	98
8.2	Results and Discussion	100
8.3	Threats to Validity	104
8.4	Chapter Summary	104

V Conclusions

Chapter 9—Concluding Remarks and Future Work		109
9.1	Research Contribution	109
9.2	Future work	110
9.3	Concluding Remarks	113
Appendix A—Multiple case studies: Data and support material		131
A.1	PRODSCALPEL usage	131
Appendix B—Experiment: Data and support material		137
B.1	ONLINE PRE-SURVEY (BACKGROUND FORM)	137
B.2	Online post-survey (Feedback form)	142
B.3	Experiment tasks: Scenario I	150
B.4	Experiment tasks: Scenario II	155
B.5	EVALUATION: Results of post-execution survey	163
B.6	EVALUATION: THE TIME MEASURED FOR THE PARTICIPANTS AND TOOL	167

LIST OF FIGURES

1.1	Schematic overview of the thesis structure.	9
2.1	The software product line engineering framework [2].	17
2.2	Extractive model of software mass customization [44].	18
2.3	SPL reengineering process, based on [7].	20
3.1	GP process.	33
4.1	Product derivation process using FOUNDRY approach.	45
4.2	An overview of how new products are derived from a product line based on ST.	46
4.3	FOUNDRY lifecycle.	47
4.4	Call graph extracted from GEdit text editor.	55
4.5	Three validation steps.	58
5.1	Overall implementation of PRODSICALPEL.	62
5.2	The reconfigurator	64
5.3	Over-organ extraction process	65
5.4	Code clones detector.	68
5.5	Source code merging process.	70
6.1	Comparative Study Design.	78
7.1	Case Study Method	87
8.1	Time (in minutes) spent by participants and PRODSICALPEL on performing the three stages of SPL reengineering.	102
8.2	Time results grouping automated and manual in both scenarios.	103

LIST OF TABLES

6.1	Comparison of the PRODSICALPEL with existing reengineering solutions to SPL regarding the strategies used and open issues addressed based on [7].	81
7.1	Donors and hosts corpus for the evaluation.	89
7.2	Case studies results	91
8.1	Experiment instrumentation	97
8.2	Details of participants' expertise (in years) and division into groups. . .	99
8.3	Scenario I: Donor NEATVI - Host Product Base	101
8.4	Scenario II: Donor Mytar - Host Product Base	101
A.1	TRANSPLANTATION FOLDER STRUCTURE	133
B.1	Scenario I.	167
B.2	Scenario II.	167
B.3	Total time (in minutes) spent by on performing the three stages of SPL reengineering: feature extraction, adaption and merging.	168

LIST OF ACRONYMS

SPL	Software Product Lines
SPLE	Software Product Lines Engineering
AutoST	Automated Software Transplantation
ST	Software Transplantation
GP	Genetic Programming
SBSE	Search Based Software Engineering
GI	Genetic Improvement
SE	Software Engineering
SDG	System Dependence Graph
TXL	Turing eXtender Language

PART I

OVERVIEW

INTRODUCTION

Software reuse is an important approach for companies interested in increasing their productivity, minimizing development costs, and improving time-to-market [1]. SPL has emerged as a systematic way to achieve reuse based on a strategy that plans the use of assets in several products rather than conventional ad-hoc reuse approaches [2, 3].

Despite its benefits, in general, adopting SPL requires a considerable upfront investment before its benefits can be obtained. The cost of migrating existing products to SPL is lower than adopting SPL from scratch, making *extractive* adoption more common, especially in companies with many software system variants in production [4]. Two factors motivate this preference: (1) it is often hard to determine the upfront investment that will be needed because related products often emerge from a small set of initial products, and (2) starting from scratch discards considerable knowledge and investment in existing codebases when they exist [5, 6].

To re-engineer existing products, companies must solve four problems: they must analyze their products to (1) identify and (2) extract the features these products share, (3) learn their inter-dependencies, and finally, (4) define a variability mechanism for combining these features, subject to their inter-dependency constraints [7]. Currently, reengineering to adopt SPL remains largely manual [7] and costly [8]. Indeed, because of its cost, software companies delay or even refrain from adopting SPL[9]. Automating these tasks remains an open challenge [7].

In his keynote paper during the SPLC Conference, Harman demonstrated the potential application of search-based techniques, such as Genetic Improvement (GI) for SPL development [10], for instance, in feature model selection, architectural improvement, refactoring, and SPL testing [11]. Since then, the search for solutions to SPL problems has sparked a surge of research and application at the intersection of Search Based Software Engineering (SBSE) and SPL, which has manifested with an increasing number of papers connecting both research communities [12].

In 2013, Harman et al. [13] introduced ST as a new research direction and laid out its implications for SPL reengineering. Harman et al. defined ST as “*the adaptation of one system’s behaviour or structure to incorporate a subset of the behaviour or structure*

of another” [13]. In terms of automated ST, Petke et al. [14, 15] were the pioneers in transplanting code snippets from different versions of a system to enhance its performance using genetic improvement [16]. A year later, Barr et al. [17] introduced a theory, algorithm, and tool that could automatically transplant a feature from one program to another successfully. Another tool, CodeCarbonCopy (CCC), was proposed by Sidiroglou-Douskos et al. [18] which automatically transfers code from a donor to a host codebase by utilizing static analysis to identify and eliminate irrelevant functionalities that are not pertinent to the host system.

The idea of ST has the potential to be exploited as an alternative approach to the optimization of the reengineering process to obtain SPL reusing existing systems. The powerful essence of this idea is the focus on deriving new products automatically from the transplant features from different systems in a product line without each feature has been initially created to be reusable.

Inspired by that idea, we introduce FOUNDRY, the first ST approach for SPL reengineering. FOUNDRY is independent of the programming language and supports SPL’s *domain engineering* and *application engineering* [19] processes at the code level. It tackles each SPL reengineering task, easing some and automating others. FOUNDRY does not eliminate the manual labour of feature identification but reduces it to the task of annotating the entry points (the interface) of a feature, or its “organ” using transplantation nomenclature. FOUNDRY amortises this manual step across a sequence of transplantations. FOUNDRY automates feature extraction using slicing to overapproximate feature dependencies. It leverages transplantation to automate the variability mechanism and, simultaneously, tackle slice-imprecision.

Key to FOUNDRY is the mapping software transplantation’s “over-organs”, a conservative over approximates an organ [17], to become assets, or features of a product line. The use of *Program slicing* [20] to organ extraction means that FOUNDRY does not need specially prepared donors; the donor programs can even be unaware that they are participating in an SPL. A product line via ST is composed of multiple over-organs and a “product base”, a host that contains all features are shared across all products within the product line, so constructing a product entails transplanting a set of organs into a product base.

Because over-organs are conservative, self-contained slices, two organs may share features. For example, in an editor, two different features, like a spell checker or a plugin manager, might share a memory-resident database feature. FOUNDRY uses clone-aware genetic improvement [16] to adapt and specialise an over-organ to its implantation point and to detect and remove cross organ redundancies.

We implement FOUNDRY in PRODSALPEL, a tool that transplants multiple organs (i.e., a set of interesting features) from donor systems into an emergent product line. It is the first ST tool that can automatically produce product lines from existing systems implemented in C. We implemented PRODSALPEL to automate the ST idea [13] for reengineering of systems into SPL. Our solution exploits *Program slicing* [20], *SPL reengineering*, *GP* [21], and *Clone analysis* techniques for extracting, transforming, and implanting multiple features’s implementation from existing systems with the aim of generating product lines.

PRODSICALPEL also can be configured to support the use of existing variability mechanisms based on *feature toggle* [22], such as feature flags and *preprocessor directives* [23]. It can automatically encapsulate the code associated with each feature transplanted by surrounding it feature flags or preprocessor directives to control which features are active at compile or runtime. This allows product line to take advantage of the benefits of both techniques, while minimizing their drawbacks. Furthermore, FOUNDRY can also be exploited as an alternative to clone-and-own approach to optimise the product variants development process. Consequently, our solution brings a new way for software reuse by providing both a faster and automatic way of reuse of features from existing codebases. By using ST, software companies also could opt for an initial strategy of generating new products and then, based on the market demand, towards a product line adoption, both strategies possible by transplanting functionalities from existing products. That way, new products might be assembled from existing software only at the moment that there is a demand for them, reducing the up-front investment, one of the main barriers to SPL adoption.

We evaluate FOUNDRY as a feasible approach to the reengineering process for extracting SPL, by conducting a comparative study (Chapter 6), thereby demonstrating the potential of ST to address open issues in the current reengineering practices in comparison with existing automated solutions.

To evaluate PRODSICALPEL, we conducted two case studies and a controlled experiment (Chapters 7 and 8). We first generate products by transplanting features from three real-world systems — Kilo¹, VI² and CFLOW³ — into two product bases generated from VI and VIM⁴, used as hosts for the target transplantations. Next, we asked twenty SPL experts to conduct an experiment of feature migration to a product line. We provided to them the same inputs as those PRODSICALPEL requires. In all cases, PRODSICALPEL outperformed our experiment’s participants in the time taken to complete feature migration. On average, PRODSICALPEL automatically migrated features across codebases 4.8 times faster, on average, than the average time a group of SPL experts took to accomplish the task.

Our results show that ST does successfully automate SPL reengineering, by combining features extracted from existing, possibly unrelated, systems.

1.1 MOTIVATION

The most common scenario of reuse in practice is based on ad-hoc techniques [7]. For example, when there to demand for a new product that has some similar functionalities to an existing product, commonly, developers fork the new product from other already existing software and then adapt it to fit the new requirements. Seeking to minimize this rework effort, reuse techniques, such as reengineering of existing systems to SPL emerged as a systematic way to synthesize a set of systems in a product line.

¹<https://github.com/antirez/kilo>

²<http://ex-vi.sourceforge.net/>

³<https://www.gnu.org/software/cflow/>

⁴<https://www.vim.org/>

Numerous legacy systems, with a long history of versions and local variations, can be targets for reengineering, as highlighted in the work of Bayer et al. [24] on SPL. However, simply recovering existing assets from single systems and trying to reuse them for new developments is not sufficient for systematic reuse [25]. Reengineering can help in extracting reusable components from existing systems, but the efforts needed for understanding, extraction and adaptation of assets should be considered.

Many reengineering approaches with focus on SPL have been proposed in order to minimize the adoption barrier [7]. Some are based on the use of generic reengineering techniques [26, 27]. Others have focused on model and code transformation approaches [28, 29]. However, recovering assets from existing systems and trying to assemble new systems from them is not totally possible using the existing SPL reengineering practices [7]. Most of the existing solutions provide support for specific tasks such as feature detection and analysis, rather than the whole SPL generation process. In addition, they are not fully automated and lack the means to consolidate different features present in more than one system into a product line [7]. Even solutions like ECCO [9] that consolidates different features present in more than one product requires that these products must be based on the same family of products which limits the capacity of reusing assets.

The proposition of autonomously transferring features among unrelated codebases has the potential to establish ST as a burgeoning and promising trend in future software development, with minimal human intervention. The application of search-based techniques to automate the ST process can be leveraged to migrate existing features into a product line. Using ST, software companies can contemplate an initial strategy of generating product variants and subsequently transitioning towards a product line as market demand dictates. This adaptable approach enables the assemblage of new products when needed, thereby curbing upfront investments.

1.2 PROBLEM DEFINITION

Automated transplantation aims to identify and extract an organ (interesting behaviour to transplant), all code associated with the feature of interest, then transform it to be compatible with the namespace and context of its target site in the host [17]. The automatic transplantation of a single organ already faces significant challenges. The code from different systems is unlikely to compile, execute and pass test cases when relocated into an unrelated foreign system without extensive modification.

The utilization of ST as a means to successfully automate the reengineering of product lines presents even greater challenges. Firstly, the code's extraction involves the identification of all semantically relevant code to maintain the organ functional even outside the donor environment. The codebase of the donor may have been developed using diverse architectures, dependencies, and implementation details, which necessitates the analysis of considerable amount of code and identification of all dependencies and interactions across the organs present in their respective codebases.

Secondly, adapting the extracted features to work with the host system can also be challenging. The host system may have different architectures or dependencies than the donor system, which can require significant modifications to the extracted organs. This

adaptation process needs to be done carefully to ensure that the feature behavior is preserved and that the organ can be integrated with the host system without causing conflicts or errors.

Thirdly, the successful implantation of the extracted organ into an emergent new product is hindered by potential dependencies among it and other transplanted organs. A single feature in a program may interact with another feature or a set of features, resulting in complex interactions that must be taken into account during the Automated Software Transplantation (AutoST) process to achieve new products that incorporate a set of useful features from a product line. Upon successful implementation and overcoming all challenges, ST for SPL reengineering can be adopted as an approach to meet the demands of the software industry.

1.3 OBJECTIVES

Face to the additional challenges described in the previous section; our general goal is to propose an approach and a tool that can be used for optimizing the process of generating both product lines as new products from existing codebases with minimal human involvement.

From this general goal, the following specific research goals are defined:

Research Goal 1: Define which transplantation stages are necessary to identify, extract, transform, and implant features from a set of donors programs to a product line;

Research Goal 2: Implement an AutoST tool for generating product lines and product variants from the transplant of multiple features;

Research Goal 3: Demonstrate the potential of automated ST to address the gaps and limitations in the existing SPL reengineering practices; and

Research Goal 4: Evaluate the proposed approach and tool regarding successful product line and product variants generation by conducting two case studies with real-world systems and a controlled experiment with SPL experts.

1.4 RESEARCH QUESTIONS

Based on such defined goals, we established the following research questions that guide this investigation:

- **How to evolve the current state of practices of the reengineering of existing systems to obtain an SPL?**

Rationale: There is a variety of approaches that supports reengineering of systems variants into SPL [7]. Thus, we need to discuss how our proposed approach supports the SPL reengineering process by considering existing solutions and their limitations.

- **How does a multi-organ transplantation approach automate existing reengineering practices for extracting an SPL from a codebase?**

Rationale: As a recent idea in the SBSE field, the ST is emerging as a new research area with many exciting opportunities for software reuse and SPL [13]. However, although the idea of AutoST has been proposed, it has not been explored yet as an alternative applied in the SPL reengineering process. Thus, a proposal that exploits such potential is needed.

- **Are the approach and tool effective to create software product lines?**

Rationale: To achieve a comprehensive research rigor and relevance, the tool should be empirically evaluated, and results have to be reported, detailing benefits and drawbacks of the approach.

1.5 RESEARCH DESIGN

This section describes the research design employed in this work. We split this investigation in three main parts: *Background*; *Proposal*; and *Evaluation studies*. Figure 1.1 shows a diagram with these macro parts and an overview of the sub-activities, which we detail next.

Background. The first part presents an overview of the basic concepts that guide this thesis, such as Software Product Lines Engineering (SPLE), reengineering for SPL, SBSE, and ST.

We first present an overview on SPLE, basic concepts, essential activities, and models for adopting. Then, we provide an overview on reengineering processes to migrate systems to SPL. Next, we discuss an overview of SBSE. Finally, we present an overview of the state-of-the-art of the ST main aspects, steps, the AutoST process, terminology, and a summary of some promising application of ST that already been tried. Such concepts provide the foundations for devising our research questions and narrowing down the possibilities to be included in this investigation.

Proposal. The second part comprises the proposal of the AutoST approach for product line generation. As a means of better understanding our approach (FOUNDRY), we first discuss the main implications of SPL reengineering via AutoST. Next, we introduce the idea of using FOUNDRY for reuse, including a motivating example for the illustration of the potential of application of ST for generating SPL. Then, we detail each stage of the FOUNDRY approach, including all techniques used. Finally, we give an overview of PROSCALPEL's features and describe how it automates the process of reengineering of existing systems into product lines using ST. For each feature, we describe the main challenges solved by our automated solution.

Evaluation studies. The third part encompasses three studies. Firstly, a comparative study, comparing our AutoST solution with existing reengineering practices and their limitations. Secondly, an empirical evaluation conducted from two case studies where we generate products by transplanting features from three real-world systems into two product bases, used as baselines for the target product lines. Thirdly, we conduct an experiment that reflects a real-world process of product line migration from existing codebases. The goal of this experiment is to analyse the effectiveness and efficiency of our

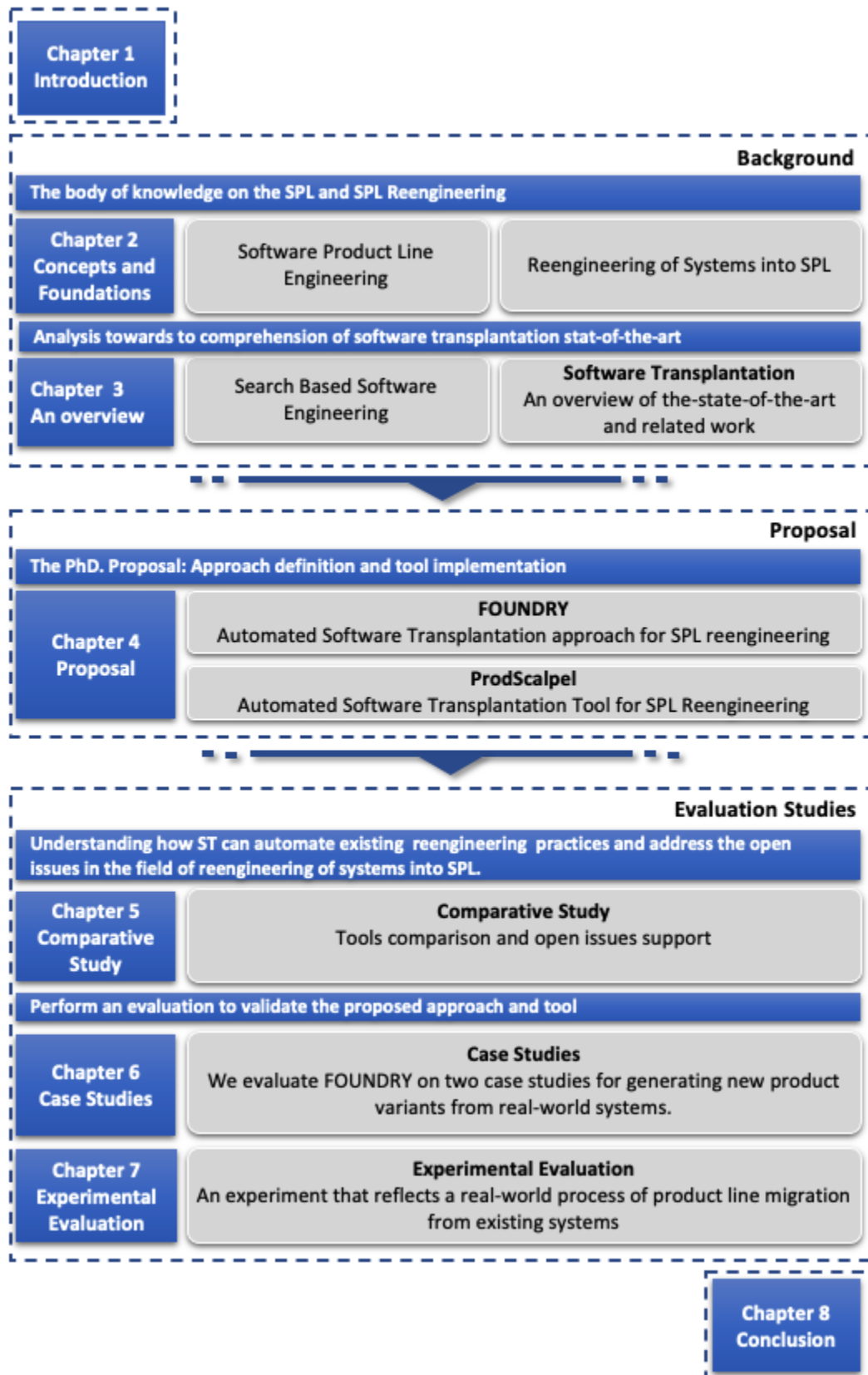


Figure 1.1: Schematic overview of the thesis structure.

approach compared with the manual process of generating a product line from existing systems, performed by SPL experts.

1.6 STATEMENT OF THE CONTRIBUTIONS

We propose that employing ST for generating product lines can be a viable and promising research direction in the field of SPL reengineering. We outline the main contributions of this work as an initial step towards exploring this research direction. They are listed in the following:

1. **FOUNDRY**, a novel SPL reengineering approach that leverages ST to define a new way of software reusing to the SPL area in which existing codebases can provide features to a product line without they have been built to be reused;
2. **FOUNDRY**'s implementation for C in **PRODSICALPEL**, a tool that transplants multi-file organs and uses clone detection to prevent implanting redundant features;
3. A set of evidence on how the ST process (as realised in **FOUNDRY**) evolves the existing reengineering practices for extracting an SPL by addressing the gaps and limitations of existing approaches; and
4. An evaluation of **PRODSICALPEL** in two case studies that demonstrates **FOUNDRY**'s promise. We use **PRODSICALPEL** to generate two product lines, and two new products, composed of features transplanted from three different real-world codebases.
5. An experimental evaluation where **PRODSICALPEL** was able to migrate features on average 4.8 faster than SPL experts performing the same task.

1.7 OUT OF SCOPE

It is rather important to define the scope of this thesis proposal. Given all the described before, we consider as out of the scope the following topics:

- *Test case suite*: there are several studies discussing how to obtain the best coverage for a test case suite [30, 31, 32]. We do not address this issue in our investigation to reduce the scope. We assume that we already have the test cases for the systems used on the running example and evaluated systems;
- *Automatic variability analysis*: our approach introduces a variability analysis stage to discover variability and commonality in the donor systems with the potential to be reused in a target product line. However, the automatic creation of a variability model, such as feature models, to express the valid combinations of features is out of the scope of this thesis proposal;
- *Implementation of a new GP algorithm*: our tool uses the same GP algorithm implemented by Barr et al. and reported in [17] that is well-recognized by the community to execute organ adaptation. Thus, we assume it is reasonably well developed and, could be evolved to support the adaptation and reduction of multiple organs.

1.8 ORGANIZATION OF THE THESIS

This PhD thesis is structured into six parts. Figure 1.1 shows a schematic overview of the thesis structure. Apart from the Introduction part, the remainder can be outlined in the following way:

Part II - Background. It presents the underlying concepts involving the key research areas of this thesis, as introduced in the Section 1.5.

Chapter 2 presents the underlying concepts involving the key research areas of this thesis proposal: SPLE and reengineering for SPL, with special focus on the reengineering process;

Chapter 3 provides an overview of the state-of-the-art of the ST, including a briefly outlines related work.

Part III - Automated reengineering of systems into SPL via ST. This part motivates and defines in detail how FOUNDRY handles SPL reengineering from existing systems.

Chapter 4 It introduces our proposed approach and tool by giving details on how new products can be incrementally derived with the support of our tool.

Chapter 5 It gives an overview of PRODSICALPEL's features and describe how it automates the process of reengineering of existing systems into product lines using ST.

Part IV - Empirical studies. This part presents evaluation studies conducted to collect evidence regarding the usefulness of ST as a strategy to generate product line from transplanting of features originally implemented in the existing systems.

Chapter 6 discusses how multi-organ transplantation (as realised in FOUNDRY) can be used to automate existing SPL reengineering practices.

Chapter 7 presents two validation case studies, including a discussion on the results and the threats to validity.

Chapter 8 describes and discusses the results of an experiment to analyses the effectiveness and efficiency of our approach compared with the manual process reengineering of SPL performed by SPL experts.

Part V - Conclusion and Future Work. This part concludes this work, with a summary and the main outlook on future research.

Chapter 9 The last chapter presents the concluding remarks of this work and describes a research agenda to continue exploring the application of ST for automating the SPLE area.

PART II

BACKGROUND

MAIN CONCEPTS AND FOUNDATIONS ON SPL AND REENGINEERING OF SYSTEMS INTO SPL

SPL defines a set of systems that share common features and artefacts to achieve high productivity, market agility, quality, low time to market, and cost [19]. SPLE strives to achieve systematic reuse of a managed set of features satisfying the needs of a particular market segment [19]. Consequently, in comparison to single system engineering, SPLE requires a considerable initial investment to establish a product line [33].

An alternative to reduce such upfront investment is to use existing systems as a baseline for building an SPL [34, 4] by using an *extractive* approach [35] to SPLE adoption. From those systems, similarities are identified, and features are reused based on a reengineering process to obtain an SPL. This process allows software companies to design core assets starting from existing systems artefacts and compose a product line. The overall approach is based on real-world system artefacts individually developed over the years, whose combinations in a product line result in a huge number of possible product configurations.

The goal of this chapter is to present the basic concepts in the context of this thesis proposal. Since the topics of the sections are broadly encompassing SPLE and reengineering of systems into SPL, along with this chapter, we provide background information rather than introducing all the existing literature. The remainder of this chapter consists of three main sections. **Section 2.1** provides an overview of the basic concepts from SPLE. **Section 2.2** presents the main idea of the reengineering process, as a mainstream strategy to deliver SPL. Moreover, it describes challenges and limitations to be overcome within current reengineering practices of systems into SPL. **Section 2.3** presents the chapter summary.

2.1 SOFTWARE PRODUCT LINE ENGINEERING (SPLE)

The need for faster development of software and for introducing new products into the market led to the concept of reuse of software assets from the existing systems [36]. The software reuse process involves the use of assets from existing software, finding the

appropriate ones that are needed at present for reuse and integrating them with a new one [37].

An efficient software reuse process facilitates the increase of productivity, reliability, and the decrease in costs and implementation time [38]. However, these benefits are not assured by the application of ad-hoc reuse approaches. These are generally opportunistic and not systematic reuse. For this reason, SPL emerges as a systematic way to achieve reuse [2]. It applies a strategy that plans the use of assets in multiple products rather than ad-hoc approaches that reuse assets only if they happen to be suitable [39]. Its basic idea is the reuse of systems functionalities or domains that can be used by a family of systems with similar needs [40].

2.1.1 Software Product Line Essential Activities

SPL exploits commonalities and manages variabilities among related products, in which it is possible to establish a common platform on top of software assets that can be systematically reused and assembled into different products. It covers processes for building, managing, and using SPL by predicting two main life cycles: *core asset development* and *product development* [19]; or *domain engineering* and *application engineering* [2]. In this thesis, we refer to the SPL phases as domain engineering and application engineering. Figure 2.1 illustrates the product line engineering framework representing the domain and application engineering phases and their development disciplines, as defined by Pohl et al. [2].

- **Domain engineering** corresponds to the process of establishing a reusable and customizable platform, by defining what will be shared among the products, as common parts, as well as by defining the variation points expressed in the artefacts, that will enable customizations. This activity is iterative and should consider the creation of assets that are generic enough to fit different environments and domains. During domain engineering, the common platform is designed and implemented. It also involves the evolution of the assets in response to product feedback, new market needs, and so on [19];
- **Application engineering** corresponds to the phase where components previously developed are assembled to compose a product. This is the phase where variability is realized so that artefacts customizations come into place. According to [2], “*the main goal of application engineering is to derive a SPL application by reusing as many domain artefacts as possible*”. This is achieved by exploiting the commonality and variability of the product line established in domain engineering.

In practice, domain engineering focuses on the creation and maintenance of reuse repositories of functional areas, while application engineering makes use of those repositories to implement new products. The separation into domain engineering and application engineering enables the distinction between the platform building process and specific-product building. However, these processes must interact in a way that is beneficial to both.

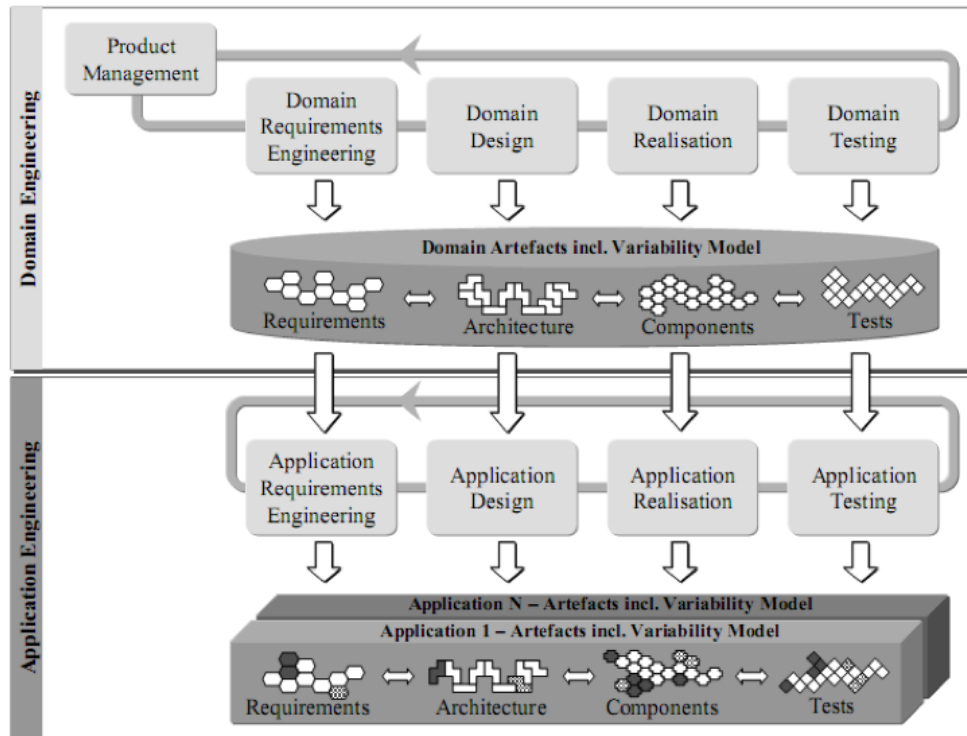


Figure 2.1: The software product line engineering framework [2].

The basic insight of SPL is that most software systems are not new. Rather they are variants of systems that have already been built. Software companies repeatedly create similar systems in a given domain, with variations to meet different customer needs. Instead of creating each new system variant from scratch, significant savings can be achieved by reusing parts of previous systems to build new ones belongs to a common domain [41]. This insight can be used to improve the quality and productivity of the software development process [42].

2.1.2 Strategies for Adopting Software Product Line

The transition to SPLE can be conducted using three software reuse adoption strategies: *proactive*, *reactive*, and *extractive* [35]. The proactive approach aims to develop a product line in a top-down approach, starting with the analysis and design activities and then implementing a complete set of common and varying source code, feature declarations, and product definitions. The adoption of a proactive approach is often assumed as a cost-saving strategy since it aims to initially implement a complete set of reusable assets, but that corresponds to a heavyweight adoption due to the required and massive up-front investment [19, 2].

By adopting a reactive approach, the organization incrementally grows software mass customization production line when the demand arises for new systems or new requirements on existing ones [43]. It offers a quicker and less expensive transition into an SPL.

In particular, the extractive approach illustrated in Figure 2.2 takes advantage of existing software systems by extracting the common and varying source code into a single production line. Such a scenario enables an organization to quickly adopt software mass customization, and it is appropriate when an existing collection of systems can be reused [35]. It makes the extractive approach the most common way to systematize the software reuse with SPL by encompassing the reengineering of existing systems, leading to a systematic reuse [4].

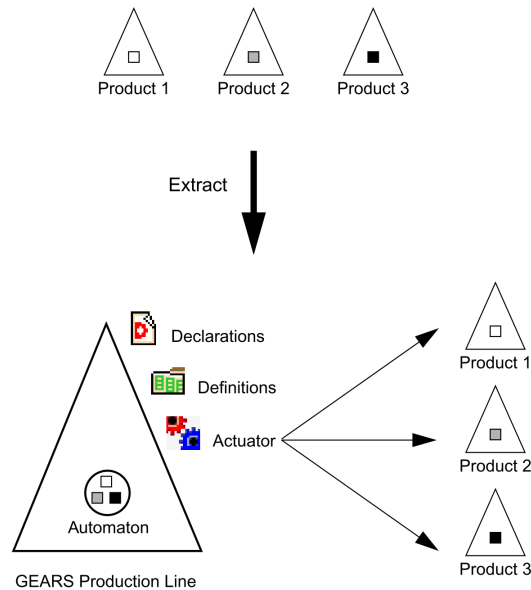


Figure 2.2: Extractive model of software mass customization [44].

The risks associated with possible out-of-date analysis and design artefacts inherent to adopt the proactive and reactive models, they are worth to consider the existing systems source code as a starting point to develop the production line in a reengineering process. Besides these technical benefits, the reengineering of existing systems into an SPL allows companies to preserve their investment and aggregate knowledge obtained during the development of individual systems. The preference for this approach is justified since the existing systems represent considerable company knowledge and investment [5].

It is important to observe that these approaches are not necessarily mutually exclusive. A common scenario, for instance, is to bootstrap building a product line effort adopting the extractive model and then move on to a reactive model to incrementally evolve the production line over time [35].

2.2 REENGINEERING OF SYSTEMS INTO SPL

Reengineering is *“the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form”* [45]. Unlike reverse engineering, which is concerned with understanding a system to create representations

of it in another form or at a higher level of abstraction, reengineering is concerned with restructuring/refactoring the systems.

In the SPLE, however, the reengineering of systems into SPL (or extractive SPL reengineering) has focused on transforming a set of existing products into an SPL. For instance, exploiting reengineering techniques for identifying and extracting software components from existing systems with the aim of populating repositories of reusable modules [46].

Extractive SPL reengineering is an active research topic with real benefits in practice [7]. It allows software development companies to preserve their investment and aggregate knowledge obtained during the development of their portfolio of systems individually developed. Because of this, the extractive approach realized from reengineering process has attracted interest from companies, with a considerable number of systems in production [47], and researchers in the SPLE field [7].

Reengineering has been the subject of constant studies, such demonstrated by Laguna et al. [48] and Fenske et al. [49] works. These works provide an overview of the reengineering activity, answering questions about existing approaches, techniques, open challenges, and suggesting a taxonomy for existing approaches. More recently, Assunção et al. [7] conducted a systematic mapping to provide an overview of the current research on reengineering of existing systems to SPLs, identify the community activity in regarding of venues and frequency of publications in this field, and highlight trends and open issues that could serve as references for future researches. Together, these works provide a coarse-grained overview of the reengineering activity and were used as the main source of information for our overview of extractive SPL reengineering.

2.2.1 Reengineering process

In the context of SPL reengineering process, there is not an established or widely accepted set of phases [7]. The most closely initiative for this is the systematic mapping of reengineering legacy applications into SPL performed by Assunção et al. [7]. In general, the main tasks associated with the phases of the reengineering process considered by the approaches are: (i) to identify the features existing in a set of systems or map features to their implementation, (ii) to analyse the available artefacts and information to propose a possible SPL representation, and (iii) to perform the modification in the artefacts to obtain the SPL. These phases are respectively called *detection*, *analysis*, and *transformation*, recalling the terminology proposed in [50, 7], and structured as presented in Figure 2.3.

- **Detection:** the first phase of the process, it is responsible by detection of variabilities and commonalities among existing systems. Such as illustrated in Figure 2.3, the variabilities and commonalities are represented in terms of features. Relevant information is extracted from the input artefacts, e.g. source code, to understand the existing structure, data flow, relationships, existing features, etc. Common support in this phase is given by feature location techniques, which aim at locating the artefacts responsible for implementing the system functionalities [7].

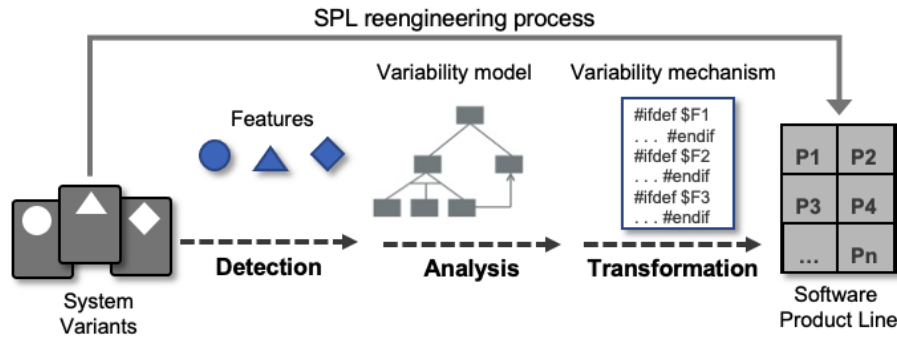


Figure 2.3: SPL reengineering process, based on [7].

The solid arrow represents the entire reengineering process, however, it is usually composed by phases, shown by dashed arrows.

- **Analysis:** this phase is devoted to the analysis and organization of discovered variability and commonality. In this phase, the variability models are created. Such model expresses the valid combinations of features of an SPL generally represented by using a feature model [51].
- **Transformation:** the last phase of the process, transformations are performed on the considered artefacts (such as source code) aiming at enabling the systematic reuse [7]. Artefacts that implement the features and the variability model are used to create the SPL, using a variability mechanism [52] or considering design models [53].

2.2.2 Strategies to Perform the Reengineering

In SPL reengineering, a strategy is defined as a technique or method applied to obtain an SPL from existing systems. They can be grouped into five categories [7]:

- **Expert-driven:** is a strategy based on the expertise of specialists such as software engineers, developers, software architects, stakeholders, etc;
- **Static analysis:** relies on following or analysing structural information of static artefacts, in other words, without their execution [54]. *Clustering, Graph-based, Heuristics, Overlaps, Structural Similarity, Model Transformation, Dependency Analysis, Rule-based, Aspect Programming, Data Flow Analysis, Program Slicing, Propositional Logic,* and *Reflection Method* are example of static analysis strategies;
- **Dynamic analysis:** when the approach makes use of tools to collect and analyse information about the artefact's execution, in general considering a low-level of abstraction, such as source code [55]. *Execution Tracing* and *Data Access Tracing* are examples of dynamic analysis strategies used;
- **Information retrieval:** this strategy leverages the fact that identifiers and comments represent domain knowledge. Commonly this strategy considers the textual

similarity [56]. *Formal Concept Analysis, Latent Semantic Indexing, Natural Language Processing Techniques, Vector Space Model, Word Frequency, Data Mining, and Ontology* are example of information retrieval strategies;

- **Search-based:** this strategy applies search-based algorithms from the optimization field [21]. Some example of search-based strategies are *Genetic Algorithm, Genetic Programming, Non-dominated Genetic Algorithm II, Hill Climbing, and Random Search*;

Most of the existing solutions use only one type of strategy, with static analysis being the category with the largest attention [7]. However, a complete solution for re-engineering process may require the use of a hybrid approach [57], i.e., the use of a combination of different strategies. For instance, dynamic analysis can be combined with static analysis such as suggested by Eisnbarth et al. [58] and Frenzel et al. [59] or static analysis combined with information retrieval [28].

Hybrid approaches can improve the results when compared with the application of only one type of strategy [7]. The lack of approaches that propose a combination of strategies is one of limitations of the SPL re-engineering literature handled by using our ST technique. We discuss it with more details on the Chapter 6 of this thesis proposal.

2.2.3 Input and Output Artefacts

Different types of artefacts provided as input and produced as output by existing approaches which can be grouped into four categories:

- **Requirement artefacts:** this type of artefacts encompass documents containing feature descriptions, customer requests, test sets generated, implementation and operation aspects, etc.
- **Domain Information:** an example of this category is a high-level description of systems in specific domain and domain analysis. Products description, user comments, documentation of systems in a specific domain, and domain analysis are the most common artefact used.
- **Design models:** design artefacts include models such as class diagrams, state machines, or entity-relationship database model. Class diagrams, state machines, and entity-relationship database models are examples of design models artefacts.
- **Source code:** corresponds to the system implementation in a programming language. Java, C, C++, and C# are the programming languages generally used.

Regarding the most common type of artefacts produced as output during the process we can group them as following:

- **Features discovered:** are in general outputs of the detection and analysis phases. Features identified or mined from artefacts, where they are not well-modularized

or spread in multiple implementation units. When the features are known and well defined, it is only necessary to obtain the mapping of features to the elements, commonly in source code. When these features are disorganized or spread across many code units, it is necessary to discover the features and its elements.

- **Features mapping:** are often generated as traceability links between known features and artefacts related with them, for example, from requirements to source code.
- **Reports:** are reports generated with information such as the variability among the systems, impact on the reengineering to SPLs, and potential reuse in legacy system variants.
- **Source code refactored:** is the most common output of the transformation phase. After generating a feature-to-code traceability, the source-code elements associated to a feature can be: clustered into a Java package (in case of object-oriented programming, e.g. [60]), migrated to an aspect (in aspect-oriented programming, e.g. [43]), or reformulated as components assets (e.g. [61]). Source code refactored is an output provided to allow a better organization of the features with the SPLE [7].

2.2.4 Research Gaps and Limitations of Reengineering Approaches

The number of solutions available for evolving existing software into a SPL can be really extensive, encompassing generic re-engineering techniques as well as model and code transformation approaches [7]. However, automated solutions fail in automate in the entire process of re-engineering of existing variants to an SPL [7]. Consequently, the current state of the practice in obtaining a product line from a codebase ends by requiring that developers employ a collection of tools for different stages of the process whose outputs require manual composition, which is translated into an up-front investment that needs to be considered in the SPL adoption process [8, 7].

The transformation phase, which enables systematic reuse of artifacts, has received limited attention in the context of SPLs as highlighted in previous research [7]. Furthermore, researchers point out the labour-intensive task of manually annotating feature entry points to adapt it to the SPL to [60] context. Rubin et al. [62, 63] have emphasized the need for sophisticated techniques for refactoring model variants to generate SPLs. Maâzoun et al. [64] have identified the use of semantics in refactoring SPLs as a potential area for future work. Therefore, new refactoring techniques should be proposed.

Additionally, researchers report limitation in the existing approaches such as provide an automation and tool support, exploiting multiple sources of information for re-engineering, feature management, implementing hybrid approaches, using refactoring, and provide a more robust empirical evaluation [7].

2.3 CHAPTER SUMMARY

In this chapter, we presented an overview of the basic concepts in the context of this thesis. We started by introducing SPLE, its essential activities and adoption models. We

also presented SPL reengineering from existing system as a common strategy to obtain an SPL.

Next chapter introduces SBSE and provides an overview on ST, the challenges to automate the process and describes some promising application of ST that already been tried, including a brief outline related work.

AN OVERVIEW OF SOFTWARE TRANSPLANTATION

Clone-and-own methodology was identified by [65] as the most common reuse scenario in practice. It is a reuse method where new variants of a software family are created by copying and adapting an existing variant [65, 9]. Clone-and-own offers companies with a considerable number of systems in production a simple way to reuse its software artefact. In many cases, artefacts of an existing product are cloned and modified to fit the new requirements. For instance, when there is a demand for a new product that has some similar functionalities to an existing product, developers usually fork the new product from another already-existing program and then adapt it to fit the new requirements

The primary reason for transferring code from a pre-existing source to their own project is to use it as a base for the new code [66]. All of this transferring process occurs by copying, modifying and pasting the necessary portions of code, being this the most common reuse process in practice. However, the process of manually searching for necessary and relevant code is tedious, stressful, and time-consuming [67]. Moreover, since code modification is performed manually by the developer, unexpected and simple errors may occur, such as omitting some variables [68].

The idea of a ST process was introduced by Harman as a new research direction in the field of SBSE. It was defined as the process of adaptation of one system's behaviour or structure to incorporate a subset of the behaviour or structure of another [13]. From this idea, initiatives to automated the process have emerged with different application possibilities [14, 69, 17, 18, 15]. Nevertheless, ST has not been explored yet as an alternative to SPLE.

The goal of this chapter is describing the basis for understanding our proposal of application of ST to achieve SPL reengineering, as well as, the related work linking reengineering for SPL, clone-and-own, SBSE, and ST. The remainder of it is organized as follows. **Section 3.1** provides a brief summary of SBSE with focus on its application in code transplantation. **Section 3.2** provides an overview of state-of-the-art in ST. **Section 3.3** presents the AutST idea, including the terminologies, main challenges and techniques proposed to automated the code extraction and transformation process and some promising applications of ST that already been tried. **Section 3.4** brings a brief outline of related work. **Section 3.5** presents the chapter summary.

3.1 SEARCH BASED SOFTWARE ENGINEERING (SBSE)

Software Engineering (SE) often considers problems related to finding a proper balance between competing and potentially conflicting goals [70]. There is often a massive set of choices, and finding suitable solutions can be hard. In scenarios like that, perfect solutions are often either impossible or impractical, and the nature of the problems often makes the definition of analytical algorithms problematic [71]. It is precisely these factors which make search-based optimization techniques readily applicable.

SBSE emerged as a discipline that focuses on the application of search-based optimization techniques to SE problems. It converts an SE problem into a computational search problem that can be tackled with a metaheuristic [72]. This involves defining a search space or a set of possible solutions. Such search-based techniques can provide solutions to the complex problems where perfect solutions are either theoretically impossible or practically infeasible using an automated approach [71].

In [13], Harman et al. explored the possibilities for new directions in research using search-based techniques with a particular emphasis on the exciting possibility of automating ST process by using GI [14, 69, 17, 15] to transfer code among different codebases. Following this idea, the initial research performed by Barr et al. [17] has achieved an interesting outcome demonstrating that search-based techniques may be used to achieve a automated ST process.

3.2 SOFTWARE TRANSPLANTATION

Different systems can share many common resources. In fact, it is quite common for software developers to copy a specific feature's code from one codebase and paste it into another. They perform this task by extracting the desired code, modifying it and manually inserting it in the new location. Kim et al. [66] observed in their study that an average of 16 code fragments per hour is copied by one program developer and pasted into another, showing that code cloning is a widely used mechanism.

During the development of a new product using existing code as a baseline, the developers need to find the code fragments they are looking for in existing code base; extract them from existing products that will be reused; compose the extracted code fragments to form the new environment; and adjust them, if needed, by adding for instance features/interactions that did not yet exist in any existing codebase. In practice, these activities are proposed in an ad-hoc and undisciplined manner, but the real problem is that these steps are presently done manually and are errors prone.

The manual extraction of relevant code is also a time-consuming task that requires detailed knowledge of existing codebase. Code can be easily missed or misidentified-leading to extracted code fragments with missing or unnecessary implementation. What makes the extraction task especially difficult is the identification of code fragments that are responsible for interactions among features. The composition is another particular and complex aspect in code reuse from existing products. It requires the merging of all relevant code with a new location while remaining faithful to structure and feature dependencies.

The process of reusing code structure can offer benefits in terms of avoiding the need for new development. However, the tasks involved in searching, extracting, composing, and adapting relevant code are known to be laborious, stressful, time-consuming, and error-prone [67]. Nonetheless, if it were possible to automatically integrate desired features from one program into another, specifically transferring the precise code segments required to ensure the functionality of the target feature in the recipient program's environment, a significant advancement would be achieved. In 2013, Harman et al. introduced the concept of ST as a novel research direction within the field of Search-Based Software Engineering SBSE [13]. ST was defined as the process of adapting the behavior or structure of one system to incorporate a subset of the behavior or structure of another system [13].

In their initial insight, Harman et al. [13] introduced several steps that could be taken into consideration when developing a transplantation algorithm [13]. They called these seven-steps by LATIIV process: *Localise*, *Abstract*, *Target*, *Interface*, *Instantiate* and *Verify*

1. *Localise*: identify the location of organ (interesting behaviour to transplant) in donor system. Over the years, many approaches have been proposed for feature identification [46, 73, 74]. Extracting [75, 76, 77] of system's component, a component of a system, given the identification of a suitable feature was also proposed, through work on *slicing* and *dependence analysis* [78, 79, 80, 81].
2. *Abstract*: create a transplantation template for feature, a template that reflects feature in its behaviour but which does not contain specific components related to the donor. Component-based software engineering [82] provides the conceptual idea of a transplantation template for organ abstraction since a component is built as a self-contained portion of code that addresses or provides a focused amount of functionality. By definition, it should contain everything it needs to work properly and developed independently from each other or the target system.
3. *Target*: find candidate locations that can host the transferred feature. In theory, this step could be completely automatic, with a search-based algorithm trying out many possible locations in the target system to host the transplant. In practice, feasibility will likely require a combination of test cases and manual annotations for the host system, akin to those used to localise the feature in the source system [13].
4. *Interface*: create an interface between the transplant and the host that can contain a candidate transplant, then add it into the host and evaluate the candidate transplant. During the modification process, genetic programming can be used to search for bindings from the host's variable in scope at the implantation point to the organ's parameters.
5. *Insert*: apply the created template from the previous step into the target location in the host system. Given a destination location and the abstract template for the feature, we want to graft the template into an existing system. Barr et al. [17] suggest the use of annotation to define an insertion context in the host in a way

that can be possible to find the host's variables at the implantation point from it. Thus, GP can be applied, using the donor's test suite, adapt the organ to execution environment at implantation point and may be inserted.

6. *Validate*: validate the transplanted candidate and its compatibility with other functions in the host. Harman et al. [13] highlighted some kind of tests which the quality of a transplant might be evaluated during the validation stage. Following this idea, Barr et al. [17] suggested three validation steps using *Regression tests*, *Argumented regression tests* (a manually augmented version of the host's regression test suite) and *Acceptance tests*.
7. *Repeat*: if the transplanted candidate does not perform effectively, or if it causes adverse side-effects with other functions, then it will be rejected, and the previous steps will be repeated to find a transplant that the host can tolerate. These steps will identify the overall challenges faced when building a transplant algorithm; however, not all approaches need include every one of these steps [13].

According to Harman et al. [13], not every approach to ST necessarily needs to follow the LATIIVR process. Nevertheless, it is important to highlight the challenges that will be encountered in localization, abstraction, targeting, interfacing, insertion, and validation tasks, all of which are likely to be critical to any such approach.

3.3 AUTOMATED SOFTWARE TRANSPLANTATION PROCESS

The transplantation process can be perceived as a distinctive variant of the common copy and paste practice. What sets this process apart is the precision in which the copied code, referred to as the "organ," is specified. In traditional copy and paste actions, the developer's focus might not necessarily be on the exact functionalities of the system. In such cases, the developer could merely select and transfer a few lines of code, which may or may not encompass a complete function. However, in the context of transplantation, the code being transplanted invariably encapsulates a precise functionality within a system. This distinction underscores the meticulous nature of the transplantation process in capturing and transferring specific system functionalities.

There have already been some attempts at the AutoST process; Petke et al. [14] were the pioneers in transplanting code snippets from different versions of a system to enhance its performance using genetic improvement [16]. A year later, Barr et al. [17] introduced a theory, algorithm, and tool that could automatically transplant a feature from one program to another successfully. This process meant automatically extracting a function from the donor, modifying it, and inserting it into the host. Using their technique, the programmer needs to identify the donor, an entry point in which the function of interest exists, the host, equality (equating the performance of the function in the donor and the host) and the process of adapting the acceptance testing in the donor function to the transplanted function in the host.

Barr et al. [17] proposed a new concept, *organ*, which refers to all code associated with a feature of interest, bringing a new idea for software reuse. Different from components

that highlight a relationship between several classes, organs emphasize on the integrity of a functionality. Organs do not have to be several classes. It can be some lines of code, a function, or one class, as long as it implements a specific functionality independently [83]. Besides the organ concept, they introduced some other terminologies to be used in the AutoST process.

3.3.1 Terminology

Some concepts in the ST area are borrowed from the field of transplantation medicine. As in medicine, a human—a developer in this case—performs the donation operation of an organ from a donor to a host. Thus, we have the three basic concepts of ST: *donor*, *organ* and *host*. The Donor is the program containing a functionality of interest called *Organ*. Donor donates the organ to a host program. To clarify the process it is important to define more terminologies stated in it. Following the concept defined by Barr et al. [17], in this thesis, we use the following terminology regarding code transplantation process.

- ***Transplantation*** - is the process whereby all code related to a feature of interest is transferred from one system into another system. Transplantation process consists of the extraction of an organ and one of its veins from a donor followed by their implantation into a host.
- ***Vein*** - a vein is a feasible execution path from a program's entry to an organ's entry point. It builds and initializes an execution environment that the organ expects [17].
- ***Organ*** - an organ represents a specific and useful feature that also considers all existing dependencies required to an organ could implement a useful functionality, for example, the features `Find` that locate a specific fragment of text in a text editor.
- ***Donor*** - is the program containing a feature of interest called the organ. Donor donates the organ to the Host program.
- ***Host*** - is the target program that needs the functionality from the donor program and accepts it.
- ***Organ's entry*** - is the entry point of code that implements a feature of interest provided by the programmer.
- ***Organ's insert point*** - is the target location(s) in the host where the feature has to be implanted. It also is supplied by the programmer.

3.3.2 Challenges to Automate the Software Transplantation Process

Barr et al. [17] automated some tasks needed to move a portion of code from a donor to a host system. The automation process exposed many challenges and difficulties associated with the process, which some were partially solved by them. These challenges are spread across the different stages of the process: in the preliminary stage, we need to define the

fragment of code to be transplanted; in the intermediate stage, we have to reduce and adapt the extracted code; and, in the final stage, we have to implant it into a non related system. These challenges are mainly related to organ extraction and organ modification activities.

- ***Dependency Analysis:*** Identifying and capturing all dependencies of the extracted code organ is crucial. It requires a thorough understanding of the interdependencies between different code components, such as libraries, functions, and variables. Failure to capture all dependencies accurately can result in broken functionality or compatibility issues when integrating the code organ into the host system.
- ***Granularity Selection:*** Determining the appropriate level of granularity for code extraction poses a challenge. Choosing a granularity that is too coarse may result in unnecessary code being extracted, leading to bloated or inefficient transplanted code. On the other hand, selecting a granularity that is too fine-grained may increase the complexity of integration and reduce the reusability of the extracted code.
- ***Preservation of Semantics:*** Ensuring that the semantics of the extracted code organ are preserved during the transplantation process is a critical challenge. The extracted code should retain its intended behavior and functionality even after modifications or adaptations are made to incorporate it into the host system. Failure to preserve semantics can lead to errors, bugs, or unintended consequences.
- ***Compatibility with Host Environment:*** Adapting the extracted code organ to work seamlessly within the host environment poses another challenge. The host system may have different libraries, frameworks, or architectural constraints that require modifications to the extracted code. Ensuring compatibility while maintaining the desired functionality of the transplanted code can be complex and time-consuming. This is a challenging vision of transplantation because code from one system is unlikely to even compile and correctly execute when it is re-located into an unrelated foreign system at least without an extensive modification [17]. Some organs will be simply untransplantable because the two systems are just too different [13]. For example, we cannot meaningfully transplant a video encoding functionality into a text editor; they are simply incompatible “species” of software. According to Harman et al. [13], even “intra-species”, transplants might be rejected, perform poorly or cause side effects by the host does not provide an execution environment that the organ expects. A transplant is *rejected* if the resulting host-plus-transplant simply fails to compile. The transplant performs poorly if it fails some of the tests which check the new desired functionality. It causes *side effects* if some regression test fails.
- ***Interactions and Side Effects:*** Consideration of the interactions and side effects of the transplanted code organ with the existing system is essential. The integration of the code organ should not disrupt the behavior of other features or introduce

conflicts. Analyzing and resolving potential interactions or side effects is necessary to ensure the overall stability and functionality of the host system.

Effectively addressing these challenges during the organ extraction stage is crucial to ensure the successful integration of the extracted code organ into the host system. Many of these challenges can be effectively tackled using well-established approaches [13]. For instance, in the code extraction process, the utilization of *program slicing* and feature extraction techniques [84, 17] can aid in identifying and extracting specific features from the donor code. Additionally, modifying and re-implementing the code into the host system can be facilitated by leveraging advancements in the GP field [13]. By leveraging these established approaches, we can enhance the efficiency and efficacy of the organ extraction stage in automated ST.

3.3.3 Program Slicing

Program slicing is a source code analysis technique proposed by Mark Weiser [85] for automatic program decomposition. By focusing on the selected aspects of semantics, program slicing is capable of automatically identifying the set of program statements, named the *slice*, which might directly or indirectly affect the values of the selected variables at a program point of interest, called the *slicing criterion* [86].

Technically, a slice is an executable subset of program statements that preserves the original behaviour of the program in relation to a subset of variables of interest and at a given point in the program [87]. Program slicing uses dependence analysis that examines the source code to trace the flow of control and data to determine the statements that belong to the slice. The process of slicing deletes parts of the program that can be determined to have no effect on the semantics of interest [86].

Slicing has many application areas [88]. Basically, any area of SE and development in which it is helpful to extract subprograms based upon semantic criteria have a potential of application of slicing [88]. Harman et al. [13] relate SBSE and program slicing with applications in many areas of software engineering, including comprehension, reuse, testing and reverse engineering.

There has also been work on locating dependence structures in a program using slicing that may be of interest to the reverse engineering [89]. In particular, there have work on the ST field [17, 15] that suggest the combination program slicing and GP techniques for code extraction and transformation. The process of slicing discards those parts of the donor program that can be determined to have no effect upon the feature. For instance, Barr et al. [17] use a new kind of GP, augmented by a form of *dynamic observational slicing* [17], guided by *Test suite observation* [90, 91]. Thus, it is possible to obtain an executable subset of program statements that preserves the original behaviour of the feature in its donor program.

3.3.4 Genetic Programming (GP)

Base on the model of biological evolution, GP is an evolutionary computation technique that allows the exploration of the space of computer programs to generate programs [92].

Its application in the software engineering process represents a way to automate one of its most expensive and time-consuming aspects, the code development. GP extends *Genetic algorithm* [93], wherein the population comprises not fixed-length character strings encoding candidate solutions, but rather actual computer programs that serve as the candidate solutions to the problem when executed.

Technically, GP is a special evolutionary algorithm in which the individuals in the population are computer programs [94]. Similar to other evolutionary algorithms, GP defines a quality criterion, typically referred to as fitness, which guides the evolution of a population of candidate solutions (individuals), following the fundamental principles of Darwinian evolution [92].

Each individual (program) in the population is assigned a fitness value based on its interaction with the environment. The program's input values are compared against the expected response values, with a higher proximity to the desired outcome indicating a better program. The fitness function determines how much a program will be able to solve the problem.

GP breeds the solutions to problems using an iterative process involving the probabilistic selection of the fittest solutions and their variation by means of a set of genetic operation, usually *selection*, *crossover* and *mutation*.

- **Selection:** corresponds to the asexual reproduction process, which selects an individual according to his aptitude and copies him to the new generation without modification;
- **Crossing:** corresponds to the process of sexual reproduction. Two programs are selected and recombined to generate two other programs based on their fitness values. A random crossing point is chosen in each parent program, and the abstract trees below these points are exchanged;
- **Mutation:** corresponds to the process of generation of one new offspring program by randomly changing a randomly chosen part of one selected program [95]. The mutation process begins by selecting a point at random within the tree. This point can be internal (function) or external (terminal). Then the selected point and the one below it are removed, and a randomly generated subtree is inserted at this point.

These genetic operations, performed iteratively, drive the evolution of solutions to problems in GP. The fittest solutions are selectively preserved and combined through crossover, while occasional random changes are introduced through mutation to explore new areas of the solution space. By leveraging these genetic operations, GP enables the generation of diverse and potentially superior computer programs to tackle complex problems in software engineering. Figure 3.1 illustrates the GP process.

Initially, an initial population of randomly configured individuals is created. Then a sequence of iterations then starts with the evaluation of the *objective functions* on the individuals in the population. Based on their results, a relative *fitness* is assigned to each solution candidate in the population. These fitness values are the criteria on

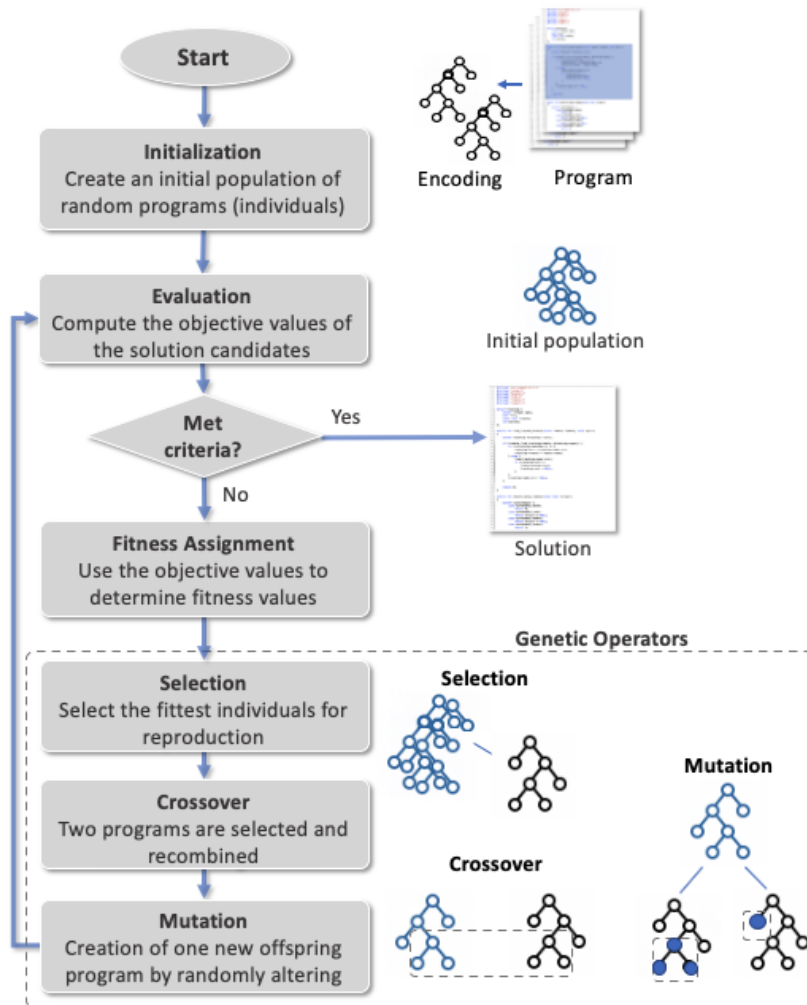


Figure 3.1: GP process.

which selection algorithms operate to pick the most promising individuals for further investigation while discarding the less successful ones. The solution candidates which managed to enter the so-called mating are then reproduced, i. e., combined via crossover or slightly changed by mutation operations. After this is done, the cycle starts again in the next generation. Then, the programs are generated through continuous improvement of an initially random population of programs [92]. Thus, generation by generation GP iteratively transforms populations of programs into other one. The execution of the GP algorithm usually ends when a criterion is satisfied. The most common is to limit the maximum number of generations or to run until a satisfactory solution is found.

Since its idealization, GP has been attracting the interest of a lot of researchers around the globe. The recent advances in Genetic Improvement (GI) field have been moving SBSE researchers to explore new directions in research using GP to improve existing programs rather than evolving them from scratch. Its versatility of use makes with the changes a constant thing in this research area, as investigators discover new ways of doing things

using GP [95]. One particular application was suggested by Harman et al. [13]. Where they proposed the use of GP to achieve automated or semi-automated software transplants by analyzing and combining all valid statements and variables of the desired functionality from the donor into a host.

3.3.5 Using Program Slicing and GP Techniques for Automating ST Process

As discussed in Section ??, two of the most challenging aspects of automating the AutoST process are extracting an organ and adapting its behavior or structure to incorporate and bind a subset of the behavior or structure of another [17]. Barr et al. [17] developed a tool for software transplanting called μ Scalpel that implements genetic programming, extended by program slicing. Together, the program slicing and GP techniques, as implemented in μ Scalpel, make the automatic transfer of functionality between two unrelated systems possible.

In order to transplant code from a donor program to a different host, the tool captures the code upon which the chosen functionality depends on the donor using program slicing [17]. The extraction process (as implemented by Barr et al. [17]) is closer to dynamic slicing [96], but it is guided by *Test suite observation* [90, 91], rather than dependence analysis, and with only a limited form of *Control dependence* [91]. Their slices also only capture the particular features of interest and not the entire computation on the slicing criterion, thereby resembling ‘barrier’ slicing [97] and feature extraction.

To modify the organ, μ Scalpel re-constructs the extracted portion of code that implements the functionality of interest to transplant and modify it to execute in the host program. A set of tests are used to guide the search for feature code modifications required to make it fully executable (and pass all test cases) when deployed in the host. During the search process, GP selects a type compatible binding from the host’s variables in scope at the implantation point to each of the organ’s parameters. Then, it selects one statement from the organ, including its vein, and add it to the individual. GP also records which statements have been selected and favours statements that have not yet been selected. At the end of the process, the organ is implanted into the host environment that is tested.

As a quite recent field of study, much more work is required to develop the idea of ST, but we argue in this thesis that it may be a valuable application for SPLE field. The capability to automatically extract from an existing system the code implementing a feature and transform it to execute in a distinct of its original code base environment has an enormous value to developers of SPL [79], no explored yet.

As an initial step in this direction, we introduce FOUNDRY, the first ST approach for SPL re-engineering. FOUNDRY is independent of the programming language, and supports SPL’s *domain engineering* and *application engineering* [19] processes at the code level. Thus, it is possible to generate a product line from the migration of features belong to existing systems. We realise FOUNDRY in PRODSALPEL, a tool that transplants multiple organs from donor systems into an emergent product line for codebases written in C. Our solution exploits *Program slicing* [20], *GP* [21], *SPL reengineering* [45], and *Clone analysis* [65] techniques for identifying, extracting, transforming and implanting multiple organs’ implementations from existing codebases with the aim of generating both product

line and product variants. More details on how FOUNDRY and PRODSICALPEL automates process of reengineering of existing systems into SPL are provided in Sections 4 and 5.

3.3.6 Software Transplantation Experiences

ST is an emerging research area with few research and empirical studies concerning the transplantation process and its practical application. Being a relatively new field, the exploration of ST presents opportunities for novel applications to emerge. By conducting rigorous and effective empirical studies, researchers can attain consistent and plausible conclusions, thereby facilitating improvements and guiding future research directions. This iterative process of empirical investigation not only enhances our understanding of the transplantation process but also uncovers potential avenues for innovation and advancement in ST [98].

Although it is a relatively new field, it is possible to highlight some promising applications of ST that have already been tried:

Code Transplants to Specialise a C++ Program to a Problem Class:

Petke et al. [14] were the first one to implement and evaluate the AutoST process. They did so by transplanting a code between two versions of the same system (MiniSAT). Their goal was to improve the execution time for a particular task (Combinatorial Interaction Testing). The results showed that their approach achieved an improvement of 17% and proved faster than the versions written by human experts [14].

Using SBSE to grow and graft entirely new functionality into a real-world system:

In 2014, Harman et al. [69] proposed a new approach called "grow and graft" to GI, which helped transplant new features into an existing system. Two steps are required: grow and graft. In the first step, a new feature is grown in isolation using GP and guidance from the developer. Then, at the grafting stage, the created feature is inserted into the host system. Using this approach, the authors successfully grew a linguistic translation feature "Babel Fish" and inserted it into the Pidgin instant messaging system [69].

Automated Software Transplantation: Recently, Barr et al. [17] introduced a theory, algorithm, and tool that could automatically transplant a feature from one system into another entirely unrelated system. This process meant automatically extracting a function from the donor, modifying it, and inserting it into the host. The algorithm they built was implemented to extract the organ and insert it into the host. Barr et al. conducted a successful case study in which they extracted a function for supporting H.264 media format from x264 encoder into a VLC media player [17].

Genetic Improvement and Code Transplantation for Software Specialising: In the domain of software specialisation, Petke et al. [15] demonstrated the application of genetic improvement techniques for automated program specialisation. Specifically, they employed genetic improvement to evolve optimized versions

of a C++ program called MiniSAT, a Boolean satisfiability solver, as well as ImageMagick, an image processing tool. They utilized code derived from GraphicsMagick, an alternative image processing tool that originated from a fork of ImageMagick. By leveraging genetic improvement, the authors successfully achieved specialisation of the aforementioned programs for three distinct applications, each possessing unique characteristics. This work showcases the potential of combining GI with code transplantation techniques to tailor software towards specific use cases and optimize its performance.

CodeCarbonCopy: Another tool, CodeCarbonCopy (CCC), was proposed by Stelios Sidiroglou-Douskos et al. [18], which automatically transfers code from a donor to a host codebase by utilizing static analysis to identify and eliminate irrelevant functionalities that are not pertinent to the host system. They evaluated CCC on eight transfers between six programs. Their results show that CCC can successfully transfer donor functionality into recipient programs.

3.4 RELATED WORK

We position our work within the existing body of knowledge in areas of re-engineering of systems into SPL, clone-and-own, variability in SPL and ST.

3.4.1 Re-engineering of Software Systems into SPL

Diverse academic proposals and industrial experience report addressing re-engineering of legacy systems into SPL are present in the literature [7]. However, this number decreases considerably when we are interested in proposals that automate the lifecycle of the re-engineering process [99].

Martinez et al. [27] introduced *But4Reuse*, a generic and extensible open source tool to facilitate extractive SPL adoption. But4Reuse is a tool that aims to extract SPLs from legacy systems by identifying a set of reusable assets and representing them in a modular way. The tool uses a variety of program analysis techniques, including clone detection and feature location to identify commonalities and variabilities in the code. Once the SPL is extracted, But4Reuse generates a set of variability models, which can be used to configure the SPL for different product variants.

In contrast, FOUNDRY does not assume an existing set of product variants. SPL can be created from a single codebase, and only requires feature entry point annotation, and a set of tests. The needed code is automatically extracted using slicing, and modified to run in the given product base via automated over-organ adaptation.

IsiSPL [100] is a reactive approach [44] to SPL adoption. IsiSPL automates the integration of new products into an existing SPL and thus generation of a new SPL with the new features. In particular, whenever a new product is added, a list of all features needs to be provided. IsiSPL then analyses SPL to only insert new features, annotating them with conditional directives. However, with a large number of products inserted over time, the list of conditional directives will grow, hindering code comprehension, maintenance, and ease of derivation of new products.

In FOUNDRY approach, the incorporation of directives for variability management is not obligatory. Developers have the flexibility to automatically introduce conditional directives within the evolved product line as they see fit. This flexibility allows them to generate new products without the requirement of including preprocessor directives in the product line. Instead, they can achieve this by transplanting organs from the transplantation platform, affording them the freedom to introduce variability without the explicit need for preprocessor directives. This approach grants developers greater flexibility in customizing and expanding the product line while maintaining the overall structure and integrity of the system.

3.4.2 Clone-and-own

FOUNDRY can be exploited as an automated alternative to clone-and-own, where, instead of creating a product line, products are cloned and amended, based on demand. Although there exists automated support for feature detection using code clone detection, its adaptation for reuse still requires manual work [101, 102].

Fischer et al. [9], for instance, present *ECCO* to enhance clone-and-own. The tool finds the proper software artifacts to reuse and then provides guidance during manual adaptation phase, by hinting which software artifacts may need to be migrated and adapted. Moreover, *ECCO* requires that the features' source code must be extracted from the same family of products, which limits its ability to reuse assets. In contrast, FOUNDRY stores over-organs that can be automatically adapted to different product bases. These do not need to come from the same family of products as the product base. Once extracted, features implemented by stored over-organs can be adapted, and implanted into a product base in a fully automated way.

3.4.3 Variability in SPL

The capacity of providing variability in a software development process is a key aspect of modern software development, enabling software products to be customized and adapted to meet the needs and requirements of different stakeholders.

Several work have already identified the frequent use of the *variability mechanisms*, like preprocessor directives [103] and feature flags [104, 105], as strategies for allowing the inclusion or exclusion of specific code blocks or features in the product line at compile time or runtime. Both are annotation-based implementation techniques for SPL require explicit annotations often scattered across multiple code units (e.g., preprocessor annotations such as `#IFDEFs`) [106]. These annotations establish a mapping of code portions to features defined in a variability model. This mapping serves as input to configurator tools, which then uses the information to select and configure the appropriate features for a given software product.

Despite its error-proneness and low abstraction level, the preprocessor directives are still widely used in present-day software projects to implement variability, maintain, evolve, reuse, or re-engineer a software system [104]. Liebig et al. [103] present a study of 40 SPL that use preprocessor-based variability mechanisms. The study analyzes the variability mechanisms used in the product lines and the impact of these mechanisms on

the codebase, in terms of code size, complexity, and maintainability. Kästner et al. [107] also discussed the concept of variability based on preprocessor directives in SPLs and how it affects the granularity of features. They present a case study of the Linux kernel to illustrate how the different levels of granularity in feature implementation can affect product line evolution and maintenance.

Other authors, such as, Jezequel et al. [108] present a case study of how feature models and feature toggles can be used in practice to manage variability in software systems. The authors describe how they used feature models to capture the commonalities and variabilities of a SPL and then translated them into feature toggles that could be used to enable or disable specific features at runtime.

Although useful, these traditional variability mechanisms have limitations [109, 103]. Preprocessor directives can lead to code bloat and reduced maintainability, while feature flags can add complexity and overhead to the codebase. Rahman et al. [22] analyzed feature flag usage in the open-source code base behind Google Chrome, finding that feature flags are heavily used but often long-lived, resulting in additional maintenance and technical debt. Meinicke et al. [105] also discovered that despite the temporary nature of feature toggles and developers' initial intention to remove them, they tend to remain in the codebase unless compelled by policy or technical measures.

Particularity, building an SPL, where the number of options can grow considerably [110, 111], the use of feature flags and/or preprocessor directives can lead to a large codebase in the emergent product line [105]. Foundry can be an interesting alternative to those traditional variability techniques. In contrast to the existing approaches where lots of annotations that are permanently added and their number increases over time, our solution requires only an annotation of the feature entry point and its insertion point in the product base. Thus, Foundry's approach has the potential to reduce code complexity and increased readability, as such any extra annotations are not required. Furthermore, Foundry keeps all reusable features of a product line (so-called organs) functional and physically separated, integrating them into the product base only when required for composing a new product.

Overall, ST technique can potentially offer several advantages over the use of traditional feature toggles, including simplified maintenance, reduced code complexity, and reduced risk of conflicts. Furthermore, it can be used in conjunction with existing variability mechanisms, such as preprocessor directives and feature flags, allowing developers to take advantage of the benefits of both approaches such as the possibility of enabling or disabling organs at runtime or compile-time.

3.4.4 Software Transplantation

Petke et al. [14, 15] were the first to transplant code snippets from various versions of the same system to improve its performance, using genetic improvement [16]. One year later, Barr et al. successfully transplanted a feature from one program into another [17].

Stelios Sidiroglou-Douskos et al. [18] proposed another tool, CodeCarbonCopy (CCC), which can also transplant code automatically. CCC is a code-transferring tool from a donor into a host codebase. It implements a static analysis that identifies and removes

irrelevant functionalists that are irrelevant to the host system. It has performed well in eight code transfers across six applications. However, the code redundancy problem still persists. CCC is thus unable to handle multiple feature transplantation. Foundry on other hand, addresses this significant problem by exploiting code clone detection. Additionally, CCC inherits the limitations of its static analysis technique [112] to identify the feature code. It typically only looks at the code in isolation and does not consider the broader context in which the code is used, such as external dependencies or interactions with other features [112, 113].

Liu et al. [114] introduced a method to transplant code from open source software. The validation results indicated that their method can substantially reduce the workload of programmers and is applicable to real-world open-source software. However their idea is also based on program slicing, it does not support the transplant of multiple organs to re-engineering of systems into SPL. Furthermore, they still not provide any tool that support their method.

In contract to those work, we are the first one to use automated ST to automate SPL engineering tasks. Our approach and tool can transplant multiple organs to compose an SPL from existing donor systems. In the process, we have solved several issues, previously not considered in the ST literature, such as code redundancy, multi-file organ transplantation, organ dependence and duplication. Furthermore, we provide a systematic approach, independent of the programming language that supports SPL's domain engineering and application engineering processes at the code level.

3.5 CHAPTER SUMMARY

In this chapter, we presented an overview of the main areas in context of this thesis. We started by summarizing SBSE as an emergent discipline that focuses on the application of search-based optimization techniques to SE problems. Then, we present ST idea, some steps, the main terminologies, the challenges to automated the process and discussed some promising application of ST that already been tried. Moreover, we presented research that address some related work in the areas of re-engineering of systems into SPL, clone-and-own, variability in SPL and ST.

Next chapter introduces FOUNDRY approach as realized by PRODSALPEL. We give details on how each stage is performed, the required inputs, and outputs produced.

PART III

APPROACH

SOFTWARE TRANSPLANTATION APPROACH FOR SPLE

The migration from legacy code to an SPL is a time-consuming process that still requires a significant amount of expert manual effort, which hinders the broader adoption of SPLE[115]. The goal of this chapter is to introduce FOUNDRY, an ST approach for the SPL reengineering process. It optimizes this process by automatically transforming existing systems into a product line. FOUNDRY is language-independent and supports the *domain engineering* and *application engineering* processes of SPLs[19] at the code level.

The remainder of this chapter is organized, as follows. **Section 4.1** provides a motivation example. **Section 4.3** presents the main concept of FOUNDRY. In **Section 4.4**, we introduce FOUNDRY approach, giving details on how each stage in *domain engineering* and *application engineering* [19] processes is performed. **Section 4.5** concludes the chapter.

4.1 FOUNDRY FOR SOFTWARE REUSE

We have idealized FOUNDRY as approach to guide the process of generating individual product variants or SPLs. By using it, software companies could opt for an initial strategy of generating variants of their products and then, based on the market demand, towards a product line adoption. That way, new products might be assembled from other already existing software only at the moment that there exists a demand for them, reducing the up-front investment.

In the first scenario, product variants generation, features are transplanted and merged into a product base on which the target product variant is assembled to compose an individual product. As a result, faster time is expected in terms of time to market of product variants generated, but the flexibility of the derived product decreases. This strategy can help software development companies to balance the initial investment and required variability. One practical example is to transplant only the features that are required for the new variant. Afterwards, those features can be used as a base for a product line that is incrementally expanded also using FOUNDRY. Thus, a company can

estimate the demand and cost for every feature separately and decide which are suitable for composing a product line.

In the second scenario, the application of FOUNDRY can be used to compose a product line through the extraction and integration of organs. In this case, existing donor systems contribute over-organs that are extracted and maintained within a *transplantation platform*, which acts as a repository of transplantation assets. Additionally, existing systems serve as product bases where organs are automatically integrated, resulting in the derivation of new product variations. This approach enables the creation and evolution of a product line by leveraging existing codebases.

4.2 MOTIVATING EXAMPLE

The open-source GNOME project¹ encompasses a large portfolio of individual programs that evolve as independently as possible from the rest. These programs share features, but because they are separately developed, their constituent features cannot be easily reused across its portfolio to provide mass customization, at least without much manual effort. The combination of mass customization and a common platform, principles of SPLE [2], would allow the GNOME team to reuse a common base of technology and, at the same time, to bring out products tailored to individual customers. Without a common platform and a software development process based on mass customization, it may be more difficult for the GNOME project to provide customized products and effectively manage the commonality and variability of its features.

The GNOME project is a natural candidate for SPL, but the significant re-engineering investment of time and resources has prevented it from adopting SPL. FOUNDRY is transformative in this case because it can be used to reduce this cost. By using PRODSICALPEL for automated support, the GNOME team can iteratively and incrementally reengineer the codebases of GNOME's application portfolio for SPL.

Suppose project collaborators want to build a product line in the domain of text editors. This product line would allow GNOME to produce text editors that augment its current text editor, *GEdit*², with additional features. Since they have decided to augment Gedit, the GNOME team would select it as the product base, the shared substrate of a product line that, for FOUNDRY, serves as the host for transplanted features. Assume that the GNOME team targets the following three features: (1) **side-panel**, (2) **split pane**, and (3) **presentation**. They then identify two donors from which to transplant these features: *NEdit*³, a multi-purpose text editor that is not part of the GNOME portfolio, and *Evince*⁴, a document viewer for multiple document formats that is part of GNOME, not an editor.

Once defined all possible donors and a host, the GNOME engineers can start the process. In the donors, they need to demarcate all feature entry points into transplants; a single annotation is sufficient for PRODSICALPEL to extract a feature. To prepare the

¹<https://wiki.gnome.org/Projects>

²<https://wiki.gnome.org/Apps/Gedit>

³<https://sourceforge.net/projects/nedit/>

⁴<https://wiki.gnome.org/Apps/Evince>

host, GNOME engineers use PRODSICALPEL to extract a product base from *GEdit* by removing all features not shared across all products within the built product line.

To demarcate the inserting point in the product base, the engineers must annotate it to indicate the implantation point for each target feature, or “organ” using transplantation nomenclature. GNOME engineers then run PRODSICALPEL that automatically extracts all of the specified feature’s source code and its dependencies, or “over-organ” using transplantation nomenclature.

Section 4.2 illustrates all transplantation iterations performed to generate a new product. It shows a new text editor derived from the transplant of features from different donors and using *GEdit* as a product base. Using feature models [51] to represent each donor system and the product base evolution, PRODSICALPEL first transplants the `side-panel` feature, extracted from *GEdit* itself. This transplantation demonstrates that PRODSICALPEL can transplant features into a product base that comes from the same codebase. Next, PRODSICALPEL transplants the `split-pane` feature from *NEdit*. It shows how PRODSICALPEL manages to transplant features from distinct codebases, which is not possible without manual effort using the current state-of-art to SPL reengineering. Finally, PRODSICALPEL transplants the `presentation` feature from GNOME’s Evince renderer.

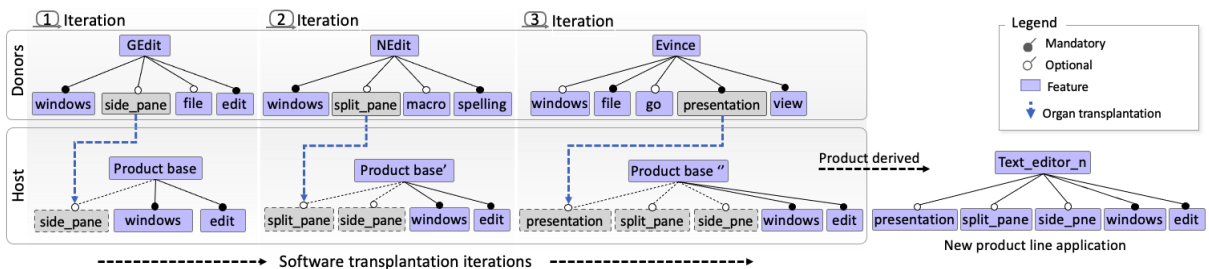


Figure 4.1: Product derivation process using FOUNDRY approach.

PRODSICALPEL *performs three sequential feature transplants into the GEdit’s product base, resulting in a new text editor after three iterations of organ transplantation.* The gray boxes represent the features selected for transplantation.

FOUNDRY facilitates transplanting features from any program into a product line, opening the door to large scale feature reuse. Open-source projects, like GNOME, are an especially promising source of code for FOUNDRY, so long as the donors and target hosts share compatible licenses.

4.3 THE FOUNDRY MAIN CONCEPT

In our idea of SPLE via ST, a feature in the code level is implemented by an over-organ that implements a functional feature or attribute of a software product. For example, an organ can be a specific functionality or a user interface element.

Based on ST idea, FOUNDRY treats *product base* and *over-organs* (representing features) as product line assets. A product base is a host that contains all features that will be shared among the products. It is adapted from an existing system which already

provides a set of solutions (features) so close to the target products that it can be used as a baseline for the assembly of products. For example, a text editing program could provide a baseline for new programs for text translation, presentation or rendering, since they could have a considerable number of common features between them. The idea is to take simultaneous advantage of commonality to reduce effort to maintaining the product line and creating new products by transplanting only specific features on demand.

An over-organ, in turn, is a completely functional and reusable portion of code extracted from a donor system that conservatively over-approximates the target organ [17]. An over-organ can be specialized to become an organ that preserves the original behavior of the feature in a different host codebase [17].

Conceptually, in FOUNDRY, while the product base provides commonalities (i.e., common features) to the target product line, the variability (i.e., variant features) are provided by the organs transplantation process, as illustrated in 4.2. This idea opens new ways for SPLE area by automated construction of different products by transplanting multiple organs into a product base.

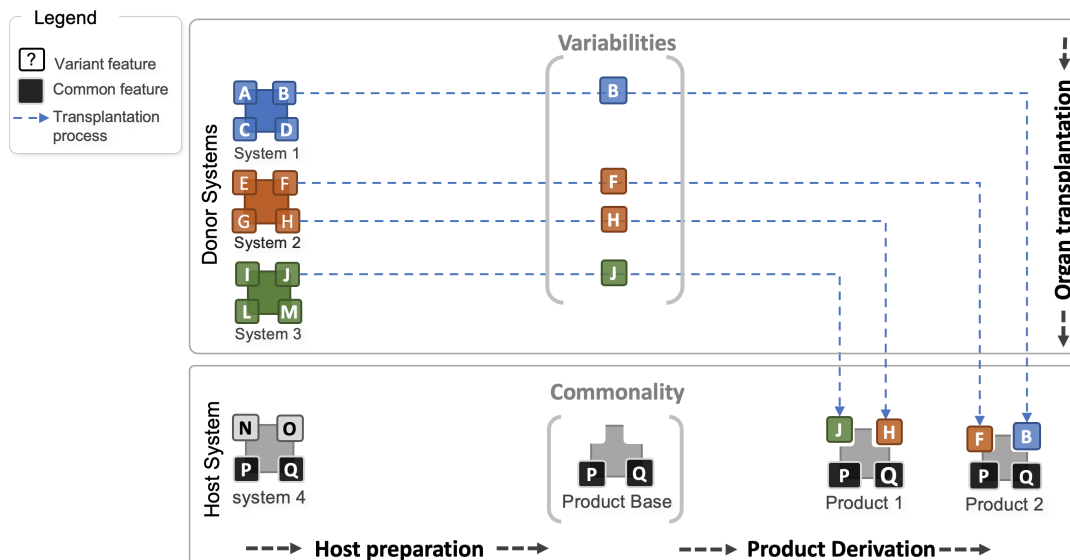


Figure 4.2: An overview of how new products are derived from a product line based on ST.

These concepts highlight how FOUNDRY leverages ST to facilitate the automated construction of SPL, providing both commonalities and variabilities through the transplantation process.

4.4 FOUNDRY

The FOUNDRY approach is a novel method for reengineering systems into SPL based on the principles of Software Transplantation (ST). It aims to automate and streamline the process of transforming existing systems into SPLs, allowing for greater variability and customization while minimizing the effort and complexity typically associated with

traditional reengineering approaches.

FOUNDRY is an approach independent of the programming language. In this way, it can be applied to a wide range of programming languages, making it adaptable to various software development contexts. FOUNDRY offers two ways of creating products through transplantation. The first approach involves using a pre-established transplantation platform. The second approach allows for the direct extraction and transplantation of an organ from a donor system into the product base, even if the corresponding over-organ is not present in the transplantation platform. These two approaches can also be combined to create specialized products.

Figure 4.3 provides an overview FOUNDRY’s workflow. The approach provides support to SPL’s *domain engineering* and *application engineering* [19] processes at the code level. By supporting both domain engineering and application engineering processes at the code level, FOUNDRY offers benefits throughout the SPL lifecycle. Here’s a breakdown of each stage defined by the approach, detailing some challenges to be faced by an automated solution.

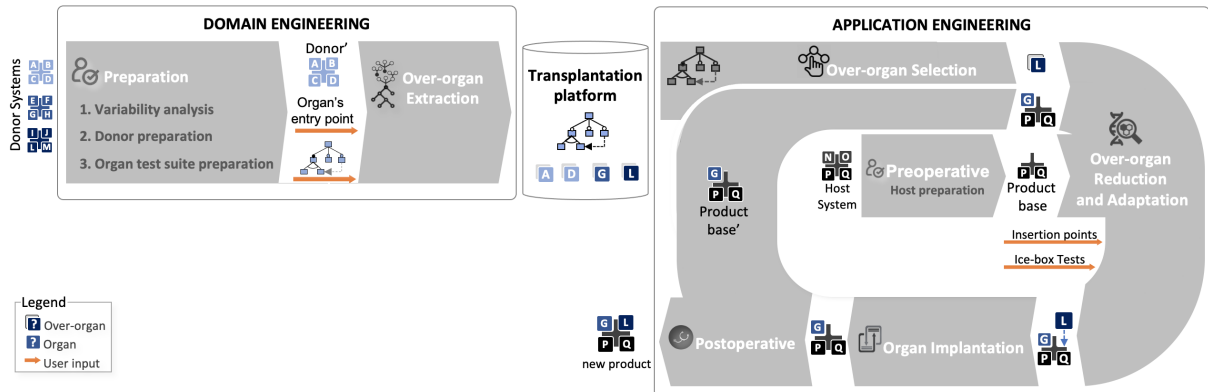


Figure 4.3: FOUNDRY lifecycle.

In the domain engineering phase, four over-organs (A, D, G, L) are extracted from three donor systems and kept in the transplantation platform, with product base consisting of 2 features (P and Q) shared across all products (P and Q). In the application engineering phase, a new product is derived from the product line after two ST iterations (organs G and L).

4.4.1 Domain Engineering

In SPL’s lifecycle the domain engineering corresponds to the process of establishing a reusable platform of core assets [19]. The process defines what will be shared among the products derived from it, i.e., commonalities. It also specifies the possible variations expressed as artefacts, that will enable the customization of product line applications, i.e., products.

In FOUNDRY, domain engineering refers to the process of establishing a product line composed of a product base and a set of reusable over-organs. This process involves the extraction and maintenance a set of over-organs, including the capture and management of

commonalities and variabilities across the product line. FOUNDRY provides a mechanism to extract these over-organs into a *transplantation platform*, facilitating the automated integration of shared functionalities and ensuring the consolidation and maintenance of common aspects within the SPL. The *transplantation platform* plays an important role in structuring the product line, bringing together the product base and reusable over-organs to establish a foundation for creating specialized products.

In analogy to medical procedures for transplantation, FOUNDRY encompasses two stages *preoperative* and *postoperative*. The preoperative stage involves preparatory tasks for both the donor and host systems in anticipation of the transplantation process, while the postoperative stage focuses on evaluating the success of the transplantation procedure. For a detailed exploration of the postoperative stage, please refer to Section 4.4.2.4, specifically focused on the evaluation and assessment of the transplantation outcome.

Specifically, the preoperative stage encompasses essential pre-transplantation activities, including *variability analysis*, *organ's test suite establishment*, and *preparation of both the donor and host systems*. These tasks are integral to ensuring the smooth execution and effectiveness of the transplantation process. The variability analysis process assists in identifying and managing the variabilities inherent in the donor organs, contributing to the customization and adaptation of features within the product line. The establishment of the organ's test suite for each organ make possible the automated adaptation of the organ to different product bases environment.

4.4.1.1 Variability Analysis. The preoperative stage starts with a variability analysis process to identify and analyze the commonalities and variabilities among a set of donor systems with potential to provide features to the product line. The goal of this analysis is to discover features in existing systems that can be used to create a product line with multiple variants extract from existing codebase. This process involves creating a variability model that expresses the valid combinations of features that can be extracted from donor systems. By creating a variability model, SPL engineers can better understand the relationships between features and create a more efficient and effective software development process.

The variability model can be represented using a feature model [51] that is a graphical representation or formal structure that captures the relationships and dependencies between features in an SPL, such as mandatory relationships, exclusive or inclusive relationships, and cross-tree constraints. It typically consists of a tree-like structure where features are organized based on their relationships .

FOUNDRY augments the traditional feature model representation to incorporate the input required for ST process. Thus, each over-organ representation in the feature model is annotated with its corresponding entry point in the donor codebase or an *organ's entry point* [17], considering the code level. An organ's entry point is a function in the donor system that belongs to the organ, defines an execution environment expected for its initialization, and provides access to the organ's test suite.

In order to create a feature model, SPL engineers need to identify the common and variable features across the donor systems. This can be done by analyzing the source code, documentation, and user requirements of the donor systems. Once the features have been

identified, they can be organized in the feature model.

In FOUNDRY, the common features are represented in a feature model as mandatory features that are shared across all product variants. These kind of features can be implemented by one or more features present into the product base or incorporated into it from the transplantation process. Variable features, on the other hand, are represented as alternative or optional features implemented by one or more over-organs into the donor systems that introduce variability among different product variants.

Determining the organ's entry point requires the SPL engineer to provide the name of the function implemented in the donor codebase. SPL engineers can use existing search tools, ranging from basic tools like *grep*⁵ to advanced information-retrieval mechanisms like *SNIFF* [116], *FLAT3* [117], and Portfolio [118], as well as configuration analysis tools [119], to aid in finding the appropriate function. Even tools such as *Doxygen* [120] for generating source code documentation can be helpful.

We can, in the future, improve our approach with *Dynamic Analysis* technique [55] specifically focused on identifying the entry point of features in software codebases. This technique could be integrated with our automated solution and approach workflows to provide SPL engineers with a more efficient and automated way to identify and analyze feature entry points. Additionally, the technique could be extended to support the identification of interactions between features, which could further improve the efficiency of ST process.

4.4.1.2 Donor Preparation. The organ in the donor code base can contain code portions that will never be used in the target products. For example, codebases written in C, in general, have code fragments guarded by `#ifdef` C-preprocessor directives [23], commonly used to control code toggle or extensions related to features. Although useful for the donor program, such code, if transplanted as part of the target organ, will generate *dead code* [52] that will never be executed in any transplanted product.

Previous work [14, 17, 15, 18] with focus on transplantation of a single organ did not concern donor clean-up. However, even when transplanting a single organ, dead code, if not removed apriori, can lead to unnecessary bloat and lower efficiency of the over-organ adaptation process (see Section 4.4.2.2).

The donor preparation task in the preoperative process consists of cleaning up the donor codebases to remove any code that is not needed in the target product line. In FOUNDRY, the donor clean-up can be performed in a manual or automated way, since the process maintains the source code structure of the program needs to be preserved (indentation, spacing, number formats, etc.), to prevent future bugs and maintain the organ's long-term viability post-extraction.

One way to perform donor preparation is to use a combination of automated tools and manual inspection to identify and remove potential dead code from the donor codebase. Automated tools, such as static and dynamic code analysis tools, can be used to identify potential dead code in the donor codebase. These tools can analyze the code and identify fragments that are never executed or that are unreachable.

⁵<https://www.geeksforgeeks.org/grep-command-in-unixlinux/>

The results of the analysis can be presented to the SPL engineer, who can then manually inspect the code to confirm whether it is a dead code or not. An inspection empowers the SPL engineer to validate the results produced by automated tools and identify any portion of code that may not be necessary for the target products. During the manual inspection process, it is of utmost importance for the SPL engineer to exercise caution and preserve the original source code structure.

Our automated solution (PRODSICALPEL) provides support to the donor preparation process by using its *Reconfigurator* and a textual list of preprocessor directives provided by the user. It avoids these collateral effects by cleaning up unused directives and associated dead codes from the donors. Thus, conditional directives belonging to the target organs are not transplanted to the host. This is done in a way that preserves as much of the source code structure belonging to the organ.

4.4.1.3 Organ Test Suite Preparation. For product derivation, an SPL engineer must supply test suites, called *ice-box tests* [17]. They are used to guide the GP algorithm in the over-organ adaptation process to create an organ that is fully executable when implanted in the product base, and that it satisfies the constraints and requirements of the product line.

Ice-box tests are typically easy to be implemented, as proposed by [17], generated using existing test generation tools, or even adapted from the donor's unit tests, when available. The SPL engineer must select or implement a set of ice-box tests that adequately cover the functionality of the over-organ. Although easy, this process requires expertise and an understanding of the donor and target systems. Once implemented, these tests must be integrated into the transplantation platform to be used in new transplantation processes of the target organ.

4.4.1.4 Host Preparation. This task can be performed before or after domain engineering phase. The objective of it is to ensure that the host codebase is ready for the transplantation process. This involves modifying the host codebase to its basic form by removing all optional features and performing tasks such as dead-code removal. The host preparation step sets the foundation for integrating the extracted organs from the donor systems into the host codebase, facilitating the successful transplantation process and enabling the generation of new product variants.

In the initial stage, the SPL engineer is tasked with selecting an appropriate product base for the target product line. If required, the chosen product base can undergo a feature removal process to reduce it to its fundamental form, retaining only the mandatory features or those that are pertinent to all products derived from the target product line. The reduction process entails the removal of code segments responsible for implementing features that are not essential for creating the target product. In cases where removal is necessary, extra care must be exercised to identify and eliminate all code portions associated with each unwanted feature while ensuring the integrity and functionality of the product base remain intact.

Although possible, doing it manually is not trivial. We advise assigning this task to

an engineer with system-specific experience, if possible, as careful attention is required to make sure that the remaining code base is properly adjusted and still executing correctly.

We have provided an automated solution in cases where the product base has been implemented using C's preprocessor directives. It uses the *Reconfigurator* module that handles C's preprocessor directives by using as input a list of all features directives which must be kept or removed. By handling preprocessor directives, the reconfigurator removes from the product base both directives and code that they delimit, while otherwise keeping the source code structure belonging to the product base unchanged. Thus, it removes optional features creating a product base ready to receive the transplanted organs. We provide more details on how the re-configurator is implemented in Section 5.1.1.

The selection of an appropriate product base, combined with efficient removal of unnecessary features, can significantly reduce the effort required for subsequent product derivation from the target product line. Although its preparation may require considerable effort for localising and removing all unnecessary code, it can be compensated with the benefits achieved through using the product base as a baseline to build products belonging to the same or similar domain, as illustrated in Figure 4.2.

It is important note that even though the preparation process may require manual effort for identifying features of interest, localising and removing dead code from the donor and preparing all test suites for the organ, it can be amortised across multiple transplantation and reuse of a single over-organ more in than one software product.

4.4.1.5 Over-organ Extraction. Once the donor is prepared, it is possible to start the automated transplantation process by construct an over-organ, a vein and an organ, that contains all the code in the donor that implements the organ, given the organ's entry. An over-organs is a conservative, self-contained slices that over-approximates the actual organ with but with all resource (code, files, library, components, etc) required to maintain it functional [17]. An organ implements the functionality that is being transplanted. An over-organ's vein and, in turn, is a pathway in the donor codebase for building and initializing the execution environment for the organ [17]. In order to ensure the successful transplantation of an over-organ, it is crucial to capture all the relevant code associated with both the organ and its vein. By carefully managing code structure, dependencies, and adaptations, the over-organ extraction process enables the successful integration of donor functionality into the product line, supporting the reengineering efforts and facilitating the creation of specialized products within the SPL.

In practice, the over-organ extraction process can consist of several lines, one or more classes, as long as they fully implement a specific functionality [13]. Thus, it can involve a considerable amount of code at different levels of granularity, from moving required files and libraries to entire functions and individual statements, both potentially not confined to a single class, file or library [121]. For instance, the feature `FEAT_DIFF` implemented in VIM has more than 5k LOCs scattered across 33 of its 166 source files.

Here we have a challenge that needs special attention, ensuring the integrity of the over-organ when the extracted code is distributed across multiple files, a concern also acknowledged by Wang et al. [121]. In such cases, it becomes essential to preserve the original multi-file structure of the extracted organ. Failing to do so could result in difficul-

ties maintaining it functional outside of the donor’s environment, which can lead to errors and failures in an SPL. Additionally, it can be challenging to propagate those changes to all product line applications, if any changes need to be made to the transplanted over-organ, such as bug fixes or enhancements,

Because over-organs are conservative, self-contained slices that over-approximates the actual organ to maintain it still functional, an extracted over-organ may itself contain multiple small features, which its functionality depends on. For example, a `spell_checker` feature might depend on a memory-resident database feature. Hence, the extraction process has to implicitly learn the feature’s dependencies, by including them in its over-organ.

Program slicing technique implemented with System Dependence Graph (SDG) [81] are promising solutions for automating the extraction process of features in ST [17]. They can potentially address the challenge of preserving the original multi-file structure of the extracted code. The process of slicing discards those parts of the donor program that can be determined to have no effect upon the organ. Hence, it allows us to obtain an executable subset of program statements that preserves the original behaviour of the organ from its entry point in the donor program.

By using program slicing, the code can be extracted in a way that preserves the dependencies between different parts of the code while SDG can help identify all the dependencies between features and components in the donor codebase. However, it is important to ensure that such techniques are used in conjunction with techniques for identifying code duplication, tackling slice-imprecision, and automated test execution, ensuring that the extracted feature performs as intended, even outside of the donor environment.

Our realization of `FOUNDRY` in `PRODSICALPEL` is able to handle all these issues. It uses program slicing and *clone-aware genetic improvement* to extract, adapt and specialise an over-organ to its implantation point, detecting and removing cross organ redundancies (see Section 5.1.2).

Upon completion of the over-organ extraction process, the source code of the extracted organ is preserved within the transplantation platform, alongside other over-organs that constitute product line assets. By storing the extracted over-organs in the transplantation platform, they become readily accessible for reuse during the application engineering phase. This enables the software engineers to leverage the existing repertoire of over-organs to integrate the desired functionality, implemented by a specific over-organ, into the target products

4.4.2 Application Engineering

In SPLE, application engineering corresponds to the phase where features are reused in new products. This is the phase where the inherent variability of the SPL is realized, facilitating the customization of artifacts to meet specific requirements.

In `FOUNDRY`, the application engineering phase encompasses the development of customized products using the organ transplantation process. Through multiple iterations of organ transplantation, a new product emerges as additional organs are transplanted. This dynamic process of transplantation enables the desired level of flexibility for product customization. This flexibility is achieved by integrating extracted over-organs with a

product base, thereby providing the necessary foundation for deriving products according to specific needs and preferences.

As highlighted in ??, the application engineering process is supported by FOUNDRY through the execution of four stages of software transplantation: (i) over-organ selection, (ii) over-organ reduction and adaptation, (iii) organ implantation and (iv) postoperative stage.

4.4.2.1 Over-organ Selection. The application engineering process starts with the target feature selection and the subsequent identification of the corresponding over-organ from the transplantation platform. The selection process is guided by the feature model generated during the earlier phase of variability analysis. The feature model provides a systematic representation of the various features and their inter-dependencies within the SPL, which helps the engineer make decisions which features to include in the target product.

The selection process involves considering factors such as feature dependencies, compatibility with the product base, and the overall goals and requirements of the target product. By aligning the target feature with the feature model, the over-organ that encapsulates the desired functionality can be traced and retrieved, facilitating its posterior integration into the customized product.

4.4.2.2 Over-organ Reduction and Adaptation. During this stage, the over-organ undergoes pruning and adaptation to ensure its compatibility with the host environment, specifically the chosen product base. The adaptation process involves the specialization of the over-organ to align with the target product base, resulting in an organ that can integrate into a specific implantation point within the product base.

An SPL engineer must select the target product base with an annotated implantation point where a call to the organ will be grafted to initialize and execute it. An automated solution must identify the insertion point within the target product base, where the over-organ will be integrated. Then it needs to reduce and specialize the over-organ to initialize and execute its functionality from the host command.

Barr et al. [17] automated the over-organ reduction and adaptation process using a GP algorithm. Its GP algorithm reduces an over-organ and specialises it to the host environment. By utilizing a context-insensitive slicing over the donor’s call graph and implements observational slice [90] in GP, it reduces the organ, transform it to execute in the host.

In the ST context for SPL, it becomes necessary to handle organs that consist of multiple files. Hence, the GP algorithm also must be capable of creating multi-file individuals, allowing the donor code statements even scattered by multiple files is available for mutation of the organ instance. However, Barr et al.’s approach does not support the adaptation of an organ that contains multiple files. Moreover, it does not support organ maintenance tasks but rather provides a one-off transplantation approach.

To address these limitations, FOUNDRY introduces an innovative concept known as the ”organ-host wrapper,” which functions as a type of organ-host interface. This wrap-

per serves as an intermediary layer between the organ and the host, facilitating their integration and maintenance. It is automatically constructed on demand by the GP algorithm, utilizing the specified implantation point within the product base and the code provided by the organ's vein.

The automated organ-host wrapper construction frees developers from the burden of manually writing the code to convert the host's data structures into parameters to the organ's entry point whenever a new product is demanded from the product line.

We optimized the GP algorithm proposed by Barr et al. [17] to handle multi-file over-organs. By combining the clone-aware genetic improvement for over-organ adaptation and the use of the organ-host wrapper, FOUNDRY offers a comprehensive and scalable approach to SPLE. This approach automates the customization of organs within the product line, providing flexibility in handling organs implemented in multiple files and enabling maintenance and re-implant of transplanted organs.

In practice, PRODSALPEL uses the GP algorithm implemented in Barr et al.'s automated solution to prune one or more program elements within the boundaries of the target organ while keeping the organ still functional and passing on the icebox tests. In the wrapper, PRODSALPEL abstracts variable names so that GP can select a type-compatible binding. It selects different combinations of all valid statements, variables and function calls mapped from the organ's vein to initialise an execution environment that the organ expects before executing it.

PRODSALPEL then uses GP to search for matching between variables in the organ and the product base. The matches found are inserted into the organ-host wrapper. By a mutation operation, a new version of the organ (i.e., a new individual) is created while PRODSALPEL makes several changes in the organ-host wrapper and pruning the over-organ. Each such mutation operation is either an `INSERT`, `REPLACE` and `DELETE` of code into the individual and the wrapper at the level of statements. PRODSALPEL then synthesises a call to the extracted organ to execute and test it from the wrapper constructed.

At the end of the adaptation process, an organ that successfully passes all the icebox tests is uniformly achieved and can be posteriorly implanted into the product base during the implantation stage. The over-organ reduction and adaptation try to ensure that the transplanted organ is fully functional and aligns with the desired behaviour and requirements. We provide more technical details on how PRODSALPEL reduces and adapts the over-organ in Section 5.1.3.

4.4.2.3 Organ Implantation. After the organ has been successfully adapted to continue functional in the host environment, it becomes ready for implantation into the product base. So far, the existing ST literature has discussed the transplantation of a single organ into a host system. However, to make the application of this technique in SPLE feasible, we have to consider the transplantation of multiple organs into a single host, a product base, including ones extracted from the same donor. As a consequence, the process is no longer concerned with a single but multiple organs and its consequent dependency and interactions.

Feature dependency is a well-known problem in software reuse [122]. Dependencies

among features are established by means of structural dependencies in the source code shared between elements of different features [123]. In practice, an organ implementing a feature in a system often shares elements, such as variables and functions with one or more organs that belong to the same donor. For instance, a structure that stores data that are manipulated by more than one function or file; or a function call between the code that belongs to organ A and B.

This situation can lead to the presence of overlapping organs within the postoperative product base. Such overlapping organs can pose challenges in terms of code redundancy, maintainability, and potential conflicts between different organs or versions of the same one. Therefore, it is essential to carefully manage and reconcile these overlapping organs during the transplantation process to ensure the integrity and coherence of the resulting product base. This may involve identifying and resolving code conflicts, making decisions about which version of a particular organ should be retained, and ensuring that the final product base remains consistent and free from redundant or conflicting code.

Figure 4.4 shows a real-world example of two call graphs from the same donor, GEdit text editor, sharing several functions. If we consider them as part of two unrelated organs, all common functions (highlighted by blue boxes) will be duplicated during their corresponding implantation processes. As a consequence, the transplantation process can insert code that is duplicated from multiple organs transplantation. Such a problem, if it is not managed, will lead to unwanted duplicated code, possibly breaking the postoperative product.

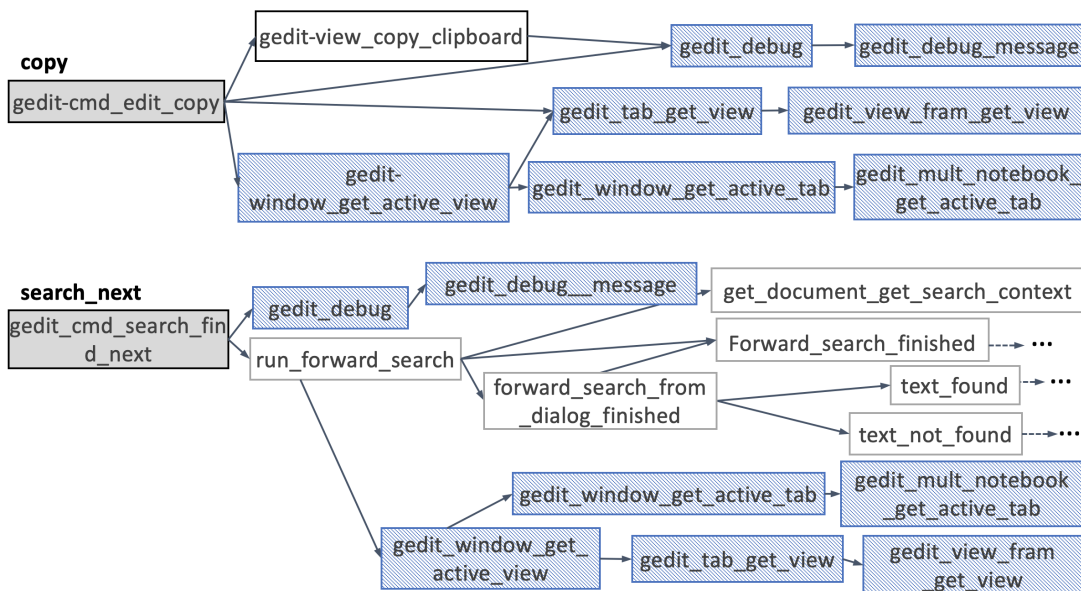


Figure 4.4: Call graph extracted from GEdit text editor.

*An example of connection points among call graphs from organs **copy** and **search_next**. Highlighted with blue boxes are functions belonging to both organs.*

Although the significant part of these overlaps is pruned from over-organ during GP-refinement, in the source code, one or more program elements within the boundaries of

an organ depend on elements external to that organ, such as a function defined in one organ and called by another organ. In other words, organ dependencies are established through structural dependencies in the source code shared between elements of different organs [123]. When this happens, the *organ collision* problem occurs, and our transplantation process would add duplicated code. Thus, it is necessary to go beyond the simple act of inserting foreign code (self-contained) into the product base without establishing any connections among the organs previously transplanted or product base elements. The objective is to enhance the functionalities of the postoperative product base by incorporating new behaviour that replicates the software organ extracted from the donor while avoiding code duplication and facilitating the sharing of common code among the organs.

It is important to carefully identify and manage all shared code elements to handle this challenge. This can be done through techniques such as code clone identification or code refactoring, where redundant code is identified and consolidated. Another approach is to use Program slicing implemented with SDG for identifying code redundancies and facilitating code consolidation. By analyzing the dependencies between different code segments, program slicing can help identify areas of redundancy and facilitate the redundant code consolidation.

To correctly work, the implantation process needs to identify all duplicated code elements and insert them only once into the product base, avoiding code duplication. Nevertheless, hosts tend to have large input spaces into which codes are inserted. In this way, finding the conflict points in the host can be difficult. For instance, functions can have the same namespace but not be identical. Thus, it is necessary to check whether a specific code element is already present in the host, considering not only its namespace but its structure and context at a fine level of granularity to make sure that two portions of code are not clones.

Identifying code clones is still more complex than a simple code duplication. Organs can have fragments of source code that are identical or very similar to those found in another organ transplanted into the product base. Clones can occur at different levels of granularity, ranging from individual lines or blocks of code to entire functions or modules. They are typically the result of copy-and-paste programming or code duplication, common in donor systems that are part of the same portfolio of systems, such as existing in the GNU Project - for example. Code clones can introduce maintainability issues, increase the risk of bugs, and hinder target product line evolution and understanding. Detecting and managing code clones during the transplant process is important to improve the transplanted organ quality, promote its reusability, and facilitate maintenance and refactoring activities product line. Hence, it is also important to ensure that the implantation technique is used in conjunction with techniques for code clone identification. Together these techniques can help to guarantee that not only the code duplication is identified but eventual code clones are effectively managed and potential errors are avoided.

The utilization of the clone detection technique is also valuable for managing the evolution of SPLs. When a new version of a donor is introduced in the SPL, it may provide organs already transplanted to the product line but with improvements or bug fixes. Similarly, over-organs within the transplantation platform may undergo changes or evolution during its life-cycle in the product line. When attempting to re-implant the new

version of an organ into the product line, there may be a portion of code that remains the same or exhibits minimal changes. To address this, clone detection techniques can be employed during the transplantation process to identify these code fragments and replace them with the updated code from the evolved organ. This helps ensure the consistency and coherence of the products while accommodating changes and improvements in the evolving SPL.

The presence of these specific aspects represents novel challenges in the application of the ST idea in SPLE, which are addressed by PRODSICALPEL. To overcome this challenge, we leverage code clone detection to prevent the insertion of duplicated code. We augmented PRODSICALPEL with a code clone detector, based on NiCad [124]. This clone detector finds exact clones over arbitrary program fragments in the organ and host source code by using Abstract Syntax Trees (AST). Thus, we exploit the benefits of *Unix DIFF command* [125] and TXL [126] to identify and compare potential syntactic code duplication.

PRODSICALPEL also supports the use of existing variability mechanisms [127] based on *feature toggle* [22] or *preprocessor directives* to facilitate its integration into an existing SPL codebase that uses them. To achieve this, PRODSICALPEL can incorporate feature flags surround implanted organs, allowing for the enablement or disablement of specific features as required. By encapsulating the organ's code within these conditional blocks, the code becomes subject to variation based on the specific configuration settings.

4.4.2.4 Postoperative Stage. As in medicine, FOUNDRY incorporates a postoperative stage aimed at assessing the potential side effects of the transplantation operation. Building upon the validation process proposed in previous works [13, 17], FOUNDRY introduces three distinct validation test suites, as illustrated in Figure 4.5. These suites, namely *Regression*, *Regression++*, and *Acceptance* tests, serve as measures for evaluating the quality and functionality of the transplant.

1. **Regression:** the test suite utilizes the host's existing set of regression tests. This suite aims to verify that the transplantation process does not introduce any defects in the product base, ensuring the preservation of its pre-existing functionalities.
2. **Regression++:** it is the **Regression** test suite expanded to achieve broader test coverage. Given the introduction of foreign code into the product base by the transplantation process, it is unreasonable to rely solely on the host's original test suite for achieving high statement coverage. Therefore, in the *Regression++* suite, additional tests are created and integrated into the host's existing test suite, augmenting its effectiveness in detecting potential issues.
3. **Acceptance:** This test suite is tailored specifically for the postoperative product, focusing on testing the newly transplanted functionality. If the transplanted organ already possesses a set of tests that adequately cover its intended behavior, those tests can be reused as part of the acceptance test suite. On the other hand, if the transplanted organ lacks comprehensive test coverage, new tests can be cre-

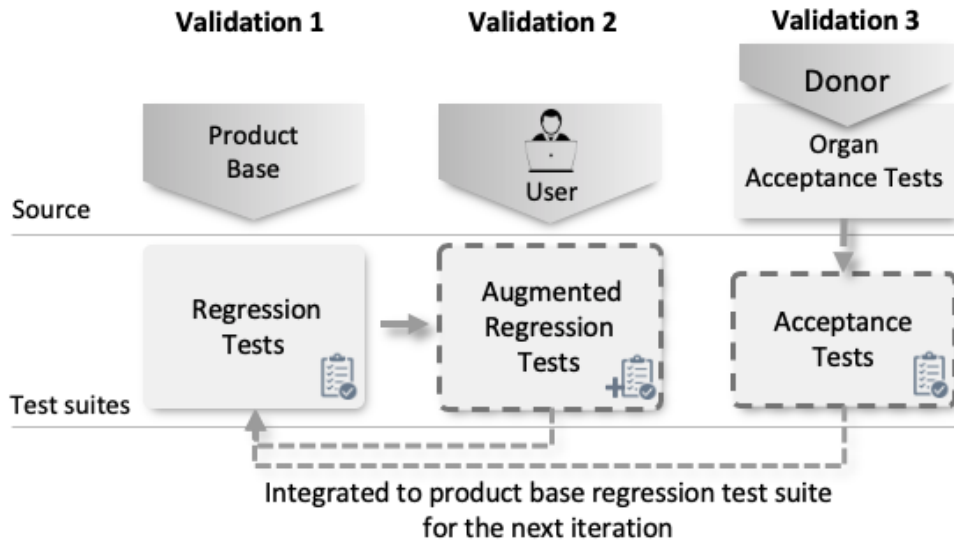


Figure 4.5: Three validation steps.

The dashed boxes are test cases added into the host regression test suite after each transplant iteration.

ated specifically for the acceptance test suite. These tests should target the desired behavior of the organ in the context of the postoperative product.

The chosen approach depends on the availability and adequacy of existing tests and the need for comprehensive evaluation of the transplanted functionality. At the end of the process acceptance tests also are incorporated to the product base's regression tests.

By implementing these three test suites, FOUNDRY establishes a robust postoperative stage that addresses the various aspects of transplantation validation, safeguarding the functionality and reliability of the transplant while identifying potential side effects.

Following the completion of the postoperative stage, the reengineering process can continue with subsequent iterations of organ transplantation. In the application engineering phase, the transplantation process enables the derivation of a new product as organs are introduced into the product base. Through sequential organs transplantation, the product base undergoes a gradual transformation, incorporating new features and capabilities while preserving existing functionality.

The iterative nature of the approach in the domain engineering phase allows for a controlled and manageable evolution of the product line, facilitating the systematic refinement and expansion of its offerings. By employing this methodology, the SPL reengineering process remains flexible and adaptable to accommodate evolving requirements and feature enhancements.

By comprehending the intricacies of each stage automation and the capabilities of PRODSALPEL, researchers and practitioners can effectively employ FOUNDRY for the reengineering and evolution of SPL, ultimately enhancing their flexibility, maintainability, and adaptability.

4.5 CHAPTER SUMMARY

This chapter has provided an overview of the fundamental concepts and stages of FOUNDRY for reengineering existing codebases into SPL. Each stage of the approach was discussed, highlighting the challenges that may arise and how PRODSICALPEL, the support tool of FOUNDRY, addresses them. Among the challenges encountered in the application of FOUNDRY, we have emphasized donor and host preparation, the management of over-organs, code duplication, extraction of multi-file over-organs, and the validation of post-operative results.

For each challenge, PRODSICALPEL's capabilities were elucidated, illustrating how the tool effectively addresses these issues through techniques such as program slicing, code clone detection, feature toggling, and comprehensive testing. This included a discussion on the use of GP for organ adaptation, the handling of multi-file organs, and the introduction of feature toggles when necessary.

The next chapter describes the implementation details. It introduces PRODSICALPEL giving an overview of its architecture and main features while describing the challenges for SPL reengineering via ST solved by our automated solution.

IMPLEMENTATION

Providing an automated solution for the transplantation of multiple organs remains an open issue in the field of ST. This is justified by the relatively recent emergence of this field and the inherent complexity of the transplantation process.

In this context, this section introduces PRODSALPEL, which is the first automated solution for multi-organ transplantation within the realm of SPLE. PRODSALPEL implements the FOUNDRY approach, tailored specifically to handle codebases developed in the C programming language. Building upon the groundwork established by $m\mu$ Scalpel [17], PRODSALPEL expands and incorporates new capabilities to enable the transplantation of multiple organs into a single host codebase (i.e., product base). This feature is particularly essential for effectively supporting SPLE, as product lines often involve the integration of diverse features to derive new products.

PRODSALPEL can be employed to support FOUNDRY for both *extractive* or *reactive*[35] product line adoption. Additionally, it can be utilized as a part of a systematic *Clone-and-own* strategy to specialize existing products.

In extractive adoption, PRODSALPEL enables the extraction and transplantation of organs from donor codebases, facilitating the integration of desired features into a product line. This process empowers SPL engineers to selectively extract and incorporate functionality from external sources, enabling the realization of feature-rich and versatile software systems

In reactive adoption, PRODSALPEL automates the adaptation and transplantation of organs in response to evolving requirements or market demands. By utilizing PRODSALPEL, SPL engineers can modify existing organs or introduce new organs into the product line, maintaining its continued relevance and competitiveness in dynamic environments.

Furthermore, PRODSALPEL supports a systematic clone-and-own strategy, allowing for the specialization of existing products. By cloning a base product and applying selective modifications through organ transplantation, software engineers can create tailored versions that cater to specific customer needs or niche markets. This approach provides the flexibility to customize products according to specific requirements.

PRODSALPEL provides solutions for problems often cited in the SPL reengineering literature [7]. It uses program slicing and *clone-aware genetic improvement* techniques to extract, specialise and implant organs to their implantation points, preserving feature behaviour while detecting and removing potential cross-organ redundancies.

PRODSALPEL also supports the use of existing variability mechanisms [127] based on *feature toggle* [22] or *preprocessor directives* [23]. It can surround implanted organs with feature flags, which permit enabling and disabling features, to facilitate its integration into an existing SPL codebase that uses them.

In the rest of this chapter, we provide an overview of the key features of PRODSALPEL, focusing on its contributions and advancements in the field of SPLE. **Section 5.1** details the specific features offered by PRODSALPEL, highlighting the challenges it addresses and the corresponding automated solutions it provides. **Section 5.2** discusses how PRODSALPEL can offer automated support for SPLE. **Section 5.3** concludes the chapter.

5.1 PRODSALPEL

The implementation of PRODSALPEL comprises five distinct modules, as depicted in Figure 5.1. These modules serve as integral components of the PRODSALPEL solution, implementing the different stages of FOUNDRY for the reengineering of systems into SPL.

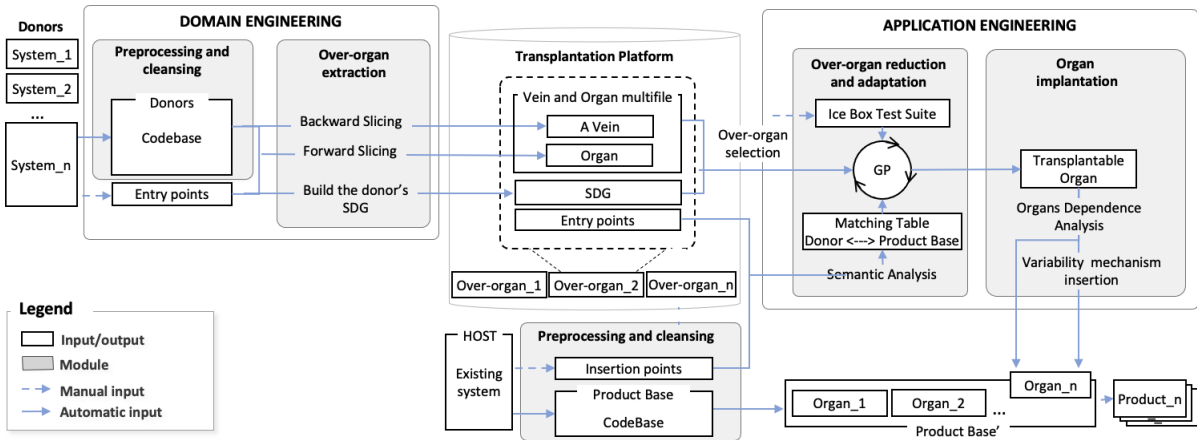


Figure 5.1: Overall implementation of PRODSALPEL.

An SDG is a system dependency graph that the GP algorithm uses to constrain its search space.

The domain engineering process is automated by the *Preprocessing and cleansing* and the *Over-organ extraction* modules. The first one is responsible for cleansing the donor codebases, eliminating any extraneous preprocessor directives that may be present. It also used to host preparing process by removing undesired features delimited by preprocessor directives from the product base. On the other hand, the *Over-organ extraction* module employs the program slicing technique implemented with SGD to extract an over-organ from the donor codebase, thereby isolating the relevant code fragments for further adaptation.

The application engineering process is automated by the *Over-organ reduction and adaptation*, and the *Organ implantation* modules. The first one utilizes genetic programming (GP) to reduce the size and complexity of the selected over-organ obtained from the transplantation platform. This reduction process is carried out while ensuring that the over-organ is effectively adapted to function within the target product base. GP enables the systematic exploration and optimization of the organ's code structure, achieving an organ specialized to a specific product base. The *Organ implantation*, in turn, employs a clone detector to identify code elements duplication and dependencies during the process of implanting the organ into the product base.

5.1.1 Automating Feature Removal

The presence of preprocessor conditionals (cpp) in large-scale systems often introduces complexities, such as intricate Boolean definitions encoding configuration dependencies and nested structures enabling code sharing across configurations. From a maintenance perspective, compile-time configurability poses significant challenges [52]. One of these challenges pertains to the synchronization between the configuration model presented to the user and the configurability implemented at the code level, which, when performed manually, becomes a tedious and error-prone task [52]. Furthermore, once a configuration is defined and a new product is generated, the portion of code consisting of unselected configurations can be considered dead code. Ideally, all dead code should be removed from the organ and product base, retaining only the code required by the current system configuration of them.

PRODSICALPEL implements a *Reconfigurator* in the *Preprocessing and cleansing* module to automate the feature and dead code removal processes required for cleaning both the donor and host systems. This implementation also addresses one of the initial limitations of the $m\mu$ Scalpel algorithm [17], namely, the default C grammar's inability to handle preprocessor directives.

To initiate the process, PRODSICALPEL requires the SPL engineers to provide a textual list of all possible configurations and the guidelines for switching between them as input. Leveraging this input, the tool performs a targeted search within the codebase, identifying and removing the code sections that implement the features delimited by these directives. However, it retains the required portion of code defined by the current configuration file, preserving the unchanged source code structure associated with the organ. When applied for streamlining the product base, it removal of all code related to the specified features by the SPL engineer. Figure 5.2 gives a example of a portion of code after PRODSICALPEL cleaned up unused directives.

Our automated solution for FOUNDRY utilizes the command line tool *unifdef*¹, which selectively processes conditional directives. While this approach proves useful, selectively processing directives alone may not be the optimal solution, as it does not fully comprehend the lexical and grammatical syntax of programming languages. For instance, *unifdef* does not handle defined or `#elif` preprocessor directives, nor does it address more complex issues, such as determining when both sides of a conditional assign the

¹<http://freshmeat.sourceforge.net/projects/unifdef>

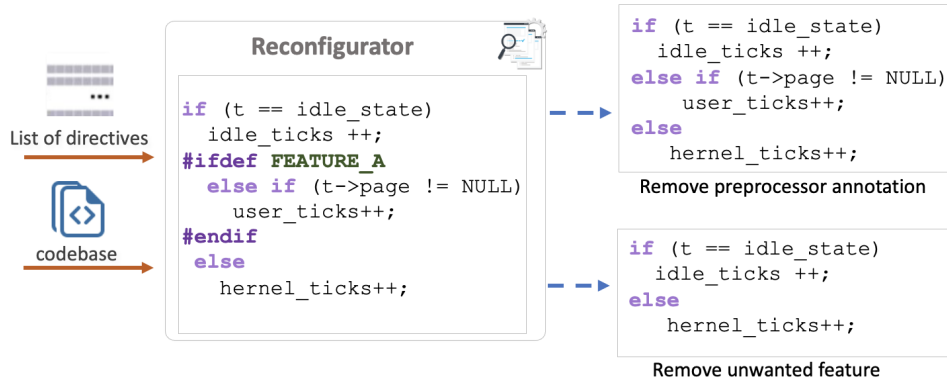


Figure 5.2: The reconfigurator

Used to automated removal of preprocessor annotation from donor codebase and unwanted features from product base.

same value to a preprocessor variable. To overcome this limitation, PRODSALPEL is also implemented in Turing eXtender Language (TXL) [126] to generate abstract trees. By utilizing TXL, PRODSALPEL gains enhanced capabilities for removing dead code and compilation directives with more precision.

5.1.2 Automating Vein and Over-Organ Extraction

The extraction of each selected organ captures a considerable amount of code not confined in a single file or library. Thus, an important issue, which must be considered during organ extraction, is how to organize the code belonging to a particular organ in terms of its file structure. Although it is possible to implement modifications to organ's files structure and even introduce a new one, we have chosen to maintain it in its original form, as implemented in its donor, without any redesign of the system besides those ones performed by GP during GP-refinement. This is justified when we consider the inherent complexity of the process, which often decreasing readability and maintainability while introducing the potential for additional programming errors.

Program slicing technique implemented in *m_uscalpel* algorithm was extended to produce a multi-file over-organ. We had to implement an organs slicer that computes slices in multi-files, keeping the original file structure of the features rather inline all their functions calls in a single file, as implemented by initial algorithm. Thus, each slice constructed includes either the files which each statement has been originally captured, necessary to produces an organ composed by multiple files. Producing a multi-file over-organ became it more self-contained and more understandable, facilitating future maintenance of the organ in the new product. Additionally, by computing multi-file over-organs, we avoid code duplication by handling dependencies between organs when transplanted from the same donor.

The process of slicing discards those parts of the donor program that can be determined to have no effect upon the organ. Hence, it allows us to obtain an executable subset of program statements that preserves the original behaviour of the organ from its

entry point in the donor program. Figure 5.3 illustrates the slicing process performed by PRODSICALPEL.

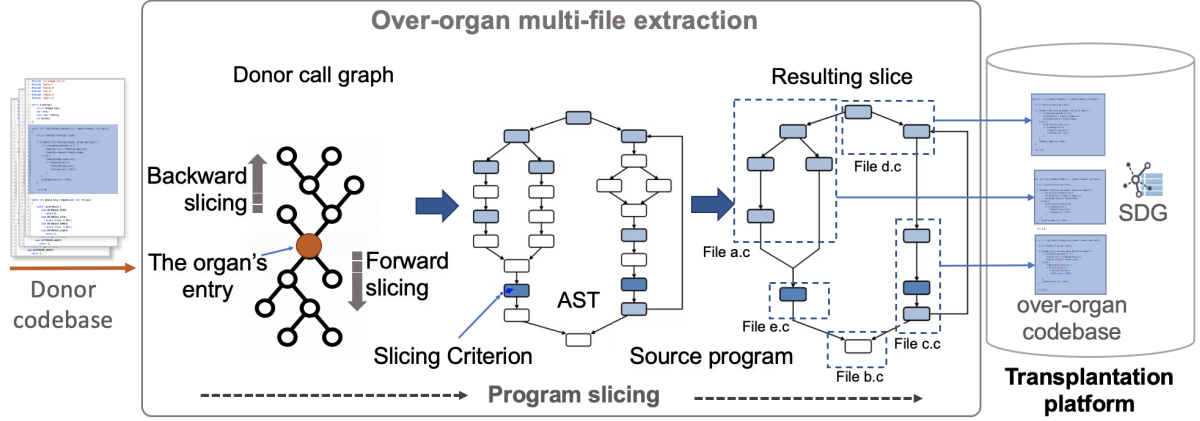


Figure 5.3: Over-organ extraction process

Similarly to previous work [17], given an entry point in the donor provided by the user, PRODSICALPEL automatically extracts an over-organ, an executable slice from the donor codebase.

Given the organ's entry point provided in the preoperative stage, PRODSICALPEL extracts an *over-organ*, an executable slice from the donor codebase, and a *vein*, constructing an over-organ, that contains all portions of code in the donor that implements the target functionality [17].

Initially, PRODSICALPEL generates a call and caller graphs for each function implemented in the donor system. Then, it selects the call graphs corresponding to the organ's entry point. The organ's entry point is used as a starting point for automated organ slicing. Using the observation-based slicing approach [20], PRODSICALPEL reaches all functions from the organ's entry point. To find the over-organ, it slices forwards by isolating the donor's call graph edges that are particularly relevant to the organ under consideration. To compute a slice, it context-insensitively traverses the donor's call graph and transitively includes all the functions called by any function whose definition it reaches.

To handle transplantation of organs spread in multiple files, PRODSICALPEL records the name of the files where the slice is and its location into the donor codebase. Then, it records the related statements in an Abstract Syntax Tree (AST), according to their order of appearance in the file. This is done to preserve the same structure in the transplanted organ as it appeared in the donor. Then, PRODSICALPEL computes the resulting slices in copies of its original files in the transplantation platform, without breaking the over-organ.

To compute a vein for an organ, PRODSICALPEL slices backward from the given organ's entry point traversing the call graph in reverse until it reaches the donor's entry point. Once it reaches the donor's entry point, PRODSICALPEL prunes the slice to retain only the shortest path, under the assumption that all paths to the organ are equivalent.

Following the solution implemented by Barr et al. [17], PRODSICALPEL inlines all functions, then maps the over-organ's statements to an array. Each array index uniquely

identifies each statement required by the GP algorithm to execute the process of pruning and adapting the over-organ.

Inlining is a technique employed by compilers to optimize code by replacing a function call with the body of the function itself. Through this technique, the GP algorithm can manipulate the code and generate new combinations that are better suited for the new host environment. However, inlining a vein can lead to the issue of vein redundancy, wherein a significant portion of the vein is shared among multiple organs transplanted from the same donors. If transplanted into the host environment, each inlined vein will contain a considerable amount of its statements duplicated.

As solution for vein redundancy, PRODSALPEL also saves each function belonging to the vein retained in a copy of its source file in the transplantation platform. Then, it also includes calls for functions existing into vein source code in the array of over-organ statements used by GP to achieve a multi-file vein.

The vein duplication is discovered during the implantation stage. In the extraction stage, both veins belonging to two organs (A and B) are kept duplicated in the transplantation platform in different directories, since it is important to keep the over-organ functional. However, when the second organ(organ B) is implanted after the transplant of organ A, the code clone detector, implemented in the *Organ implantation*, module looks for duplication also in the vein code of organ A. For example, imagine that a vein has a function `fx()` implemented in file `F.c` and belonging to both over-organs A and B. When PRODSALPEL tries to transplant organ B after organ A, it checks if the function `fx()` already exists in file `F.c` in the post-operative environment. In this way, if the function `fx()` is already in the host post-operative, as part of organ A, PRODSALPEL does not insert it into the host again. Instead, it introduces a calling from the vein belongs to the organ B. In this way, the function `fx()` already transplanted also is used by organ B. Thus, a function/method in the vein shared between two or more organs is not duplicated when transplanted.

PRODSALPEL also identifies occurrences of mutually recursive functions, even an occurrence of an indirect recursion. Technically, PRODSALPEL inlines the vein code while puts each function found in a stack of functions. When PRODSALPEL finds an occurrence of a recursive function it recovers the beginning of the recursive call and does not inline it. Instead, PRODSALPEL extracts the function to a file with the same name of the where the function is implemented. Then, it inserts a calling from the vein to the recursive function. This solution provides a finite interpretation for not inlining mutually recursive functions in the array of over-organ statements. Thus, PRODSALPEL can produce a search space for GP with a finite amount of program statements even from recursive functions.

As the donor code is slicing, PRODSALPEL also generates an SDG for both the organ and its corresponding vein, which is subsequently retained within the transplantation platform. The SDG serves as a graphical representation of the relationships and dependencies among various code elements encompassed within an over-organ, including functions, variables, and statements. This SDG guides the GP algorithm and constrains the search space for over-organ reduction and adaptation.

The GP algorithm utilizes the SDG to navigate the organ's complex web of depen-

dencies among statements, ensuring that the selected code elements for reduction and adaptation retain the essential functionality while mitigating any potential negative impacts on the overall system behaviour. The SDG acts as a roadmap for the GP algorithm, guiding its exploration and enabling it to make informed decisions based on the interdependencies among code elements.

5.1.3 Automating Over-organ Reduction and Adaptation

As proposed in FOUNDRY, PRODSICALPEL improves the process of over-organ reduction and adaptation by being able to handle over-organs containing multiple files. It also introduces a layer in the organ that works as an organ-host wrapper.

In practice, PRODSICALPEL uses GP, as in previous work [17], to prune one or more program elements within the boundaries of the target organ while maintain the organ still functional and passing on the icebox tests. Furthermore, the GP algorithm is used to search for matching between variables in the organ and the product base during the over-organ adaptation process. The matches found are inserted in the organ-host wrapper.

By a mutation operation, a new version of the organ (i.e., a new individual) is created while PRODSICALPEL makes several changes in the organ-host wrapper and prunes the over-organ. Each such mutation operation is either an `INSERT`, `REPLACE` and `DELETE` of code into the individual and the wrapper at the level of statements.

To create individuals for the next generation, a crossover operation concatenates two individuals from the current population by appending one list to another. The first parent is chosen based on its fitness value, while the second parent is uniformly selected from the breeding population, following a similar approach to previous implementations [14].

At each generation, PRODSICALPEL selects the top 10% most fit individuals and adds them to the new generation. Tournament selection is employed to choose 60% of the population for reproduction. Parents must be compilable, and if the proportion of possible parents is less than 60% of the population, the GP algorithm generates new individuals and starts a new refinement loop.

During the GP process, for each individual, a type-compatible binding is uniformly selected from the host's variables in scope at the implantation point for each of the organ's parameters. Then, one statement from the over-organ, including its vein, is uniformly chosen and added to the individual. The GP system keeps track of the selected statements and favors those that have not been selected yet.

In the organ-host wrapper, PRODSICALPEL abstracts variable names so that GP can select a type-compatible binding. It selects different combinations of all valid statements, variables and function calls mapped from the organ's vein to initialise an execution environment that the organ expects before executing it. In the end, PRODSICALPEL synthesises a call to the individual to execute and test it from the interface constructed.

At the conclusion of the adaptation process, achieves an organ that passes all the icebox tests. This organ is subsequently implanted into the product base during the implantation stage. The over-organ reduction and adaptation processes aim to ensure that the transplanted organ is fully functional and aligns with the desired behavior and requirements.

5.1.4 Automating Multiple Organs Implantation

Once the organ is adapted to correctly work on the host, it can be automatically implanted into the product base.

In our approach, code elements already belonging to the beneficiary or more than one organ can be characterized as a simple code redundancy or *implicit connection points* since they can represent a connection or dependence points among two or more organs. To correctly insert an organ PRODSCALPEL needs to identify and handle potential connection points and code redundancy, avoiding the insertion of code duplication into the product base.

To automate the implantation process, we have implemented in PRODSCALPEL a code clone detector based on the *NiCad* [124]. Our implementation of it combines *Program differencing* [125] implemented in a *Clone detection* technique to identify individual textual differences at a line level, even if it reflects changes in the organs already transplanted.

Figure 5.4 illustrates our implantation process and the solution to avoid the *organ collision*, the problem described in the previous chapter. In essence, the clone detector analysis whether a specific code element already exists within the beneficiary's environment. To do this, it constructs two lists, one with elements in the organ and the other with elements in the host. Then, it checks if there is a code element in the target organ which is also in the list of the host's elements.

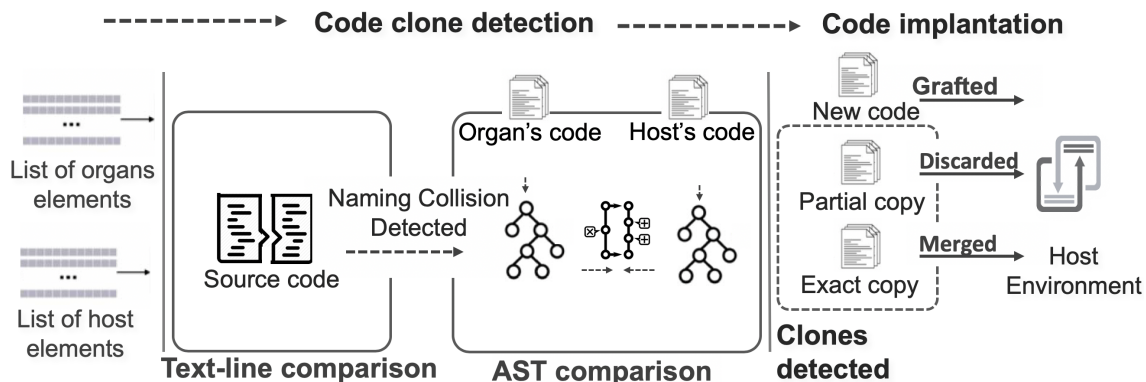


Figure 5.4: Code clones detector.

Once a potential element duplication is identified, they are compared line-by-line using a program differencing implementation. It checks if the code element is already present in the beneficiary by looking for *name collisions* [128] in the host codebase.

Although useful, the use of program differencing technique is not enough solution yet, because a line-by-line code comparison does not really understand the programming language's lexical and grammatical syntax. For example, *Program Unix DIFF* command [125] used by PRODSCALPEL does not provide enough information to PRODSCALPEL can handle more complex problems, such as determining high-level software changes such as refactorings [129] and crosscutting modifications [130], which often consist of a group of changes that share similar structural characteristics.

PRODSCALPEL exploit the benefits of TXL [126] to decompose both organ and host

elements in ASTs. Thus, our automated solution finds clones over arbitrary program fragments in organ and host source code by comparing abstract syntax trees. Additionally, by decomposing the code elements, PRODSALPEL can handle feature dependencies using a context-free parsing mechanism implemented in TXL.

Using TXL, PRODSALPEL can flexibly select the granularity of an input, such as a parse tree, under the control of a context-free grammar [124]. This allows PRODSALPEL to fine-tune the existing code of potential clones by introducing additional line breaks. By doing so, potential variances within statements and other structures can be accurately inserted using sub-abstract tree comparison.

As each code element from the organ is analysed, it is either completely **grafted** into the host codebase, **discarded**, or **merged** with existing code. The decision to graft, discard, or merge a code element is made based on some factors. For example, a code element is discarded when it conflicts with the existing one in the codebase, keeping the codebase version.

In cases where a code element needs to be merged, PRODSALPEL incorporates additional line breaks into the code and encapsulated by a feature toggle mechanism (provided the current transplantation’s input configuration specifies it). By incorporating feature toggle and line breaks, PRODSALPEL can re-implant new versions of organs into a product base. This approach enables the SPL engineers to manage the product evolution and organ changes. For example, consider the following function (shown in Figure 5.4 (a) and (b)) belonging to the organs `DIFF` and `SPELL_CHECK`, both extracted from VIM. Using a line-by-line code comparison of the segments (highlighted in blue), including the function’s signature, we can accurately determine that it is shared between the two organs. Considering an appropriate similarity threshold for the code segments, we can see that `prepare_help_buffer`, if transplanted again, in a new iteration, would generate the clone pair. In this case, it should not be transplanted again. However, it is possible to see that there is a variance present when the code fragment is used by the `SPELL_CHECK`, (line 22 in Figure 5.5 (c)). Thus, PRODSALPEL only catches this variance in the code segments, encapsulates it with a feature toggle mechanism and inserts it into the host (shown in Figure 5.5 (c) line 21).

It is important to highlight that PRODSALPEL is an initial FOUNDRY implementation for the SPLE field, which emerged during the execution of our multiple case study. As with all initial solutions, it still has some limitations inherited from external tools used in the process, such as the imprecise call graphs when dealing with function pointers, which affect the extraction of larger organs, see Section 7 for more details. This highlights the need for future research and improvements in refining the implementation of program slicing techniques and overcoming that limitation.

5.2 AUTOMATED SUPPORT FOR SPLE

Automated ST, as proposed in FOUNDRY, offers a promising avenue for enhancing and automating the reengineering of systems into SPLs. The potential benefits and opportunities provided by ST justify the effort to explore research opportunities in ST idea for SPLE area. Here we elaborate on and discuss such opportunities.

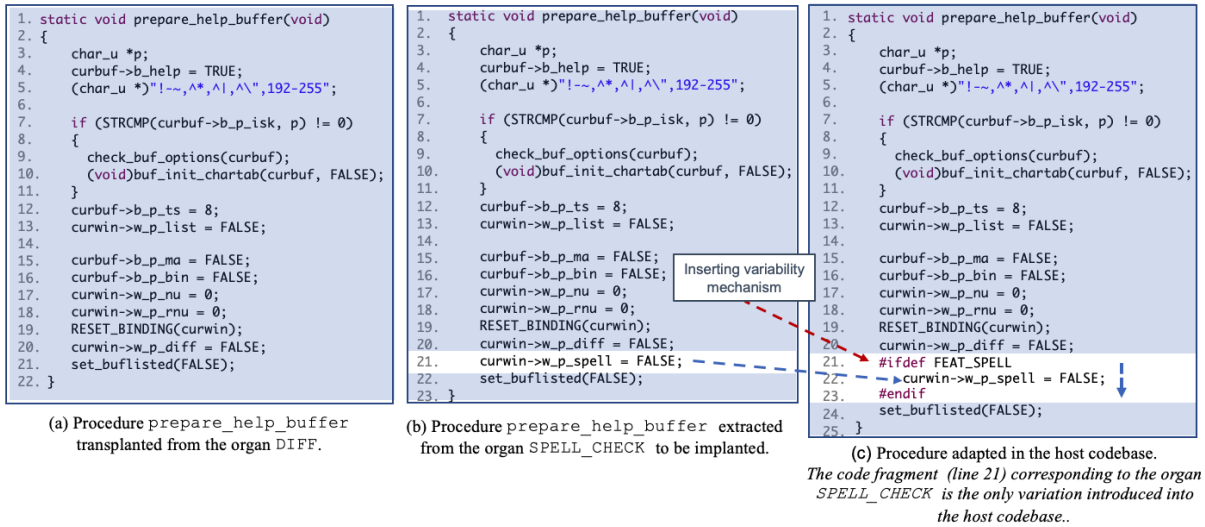


Figure 5.5: Source code merging process.

5.2.1 Automating re-engineering of existing systems into SPL.

The fundamental concept behind the `FOUNDRY` approach is to introduce a novel method for reengineering systems into SPLs by building upon the principles of `ST`. The aim is to improve the traditional reengineering process by leveraging the benefits and insights offered by `ST` area, thereby enabling more efficient and effective SPL adoption.

Our idea of `SPL` based on `ST` offers a promising avenue for automating the process of reengineering existing systems into `SPL`. Companies can use `FOUNDRY` for the initial conversion of an existing codebase into `SPL`, taking advantage of the numerous benefits associated with `SPL` adoption.

Even with such a one-off application of `FOUNDRY`, can itself be used to evolve or maintain a product line based on a conventional variability mechanism. Its automated solution can be configured to encapsulate transplantable organs with feature toggles before it is implanted, which permits enabling and disabling of features, to facilitate its integration into an existing `SPL` that uses them.

The opportunity for research in this area lies in exploring and extending the capabilities of the `FOUNDRY` approach to further enhance the evolution and maintenance of product lines based on conventional variability mechanisms. While `FOUNDRY` initially focuses on the reengineering of existing systems into `SPL`, it can also be utilized as a valuable tool for managing and adapting product lines in an ongoing manner.

5.2.2 Automating clone-and-own technique.

One compelling research avenue lies in utilizing `FOUNDRY` to automate the application of the *clone-and-own* technique [65, 9]. The *clone-and-own* approach entails duplicating existing software artifacts and independently modifying them to accommodate various product variants. However, managing consistency and synchronization between shared features across these variants presents significant challenges.

FOUNDRY offers a solution to such challenge by facilitating the transplantation of fixed or updated versions of shared features. This automation brings consistency among the variants, mitigating the need for manual synchronization and improving overall maintenance efficiency.

Consider a scenario where a bug is identified in a shared feature present in one product variant created using the clone-and-own technique. With FOUNDRY, the fixed version of the feature can be transplanted onto the unpatched copy, synchronizing the changes and eliminating the need for manual intervention.

5.2.3 Automating Reactive Product Line Adoption Process

In some organizations, product lines are already in place, but the adoption process follows a reactive approach, where new features are added only when there is a specific need for them. This reactive strategy often leads to ad-hoc and unstructured product development, resulting in limited scalability and flexibility.

FOUNDRY, with its automated transplantation capabilities, represents an opportunity to enhance and streamline the reactive product line adoption process. By leveraging FOUNDRY, organizations can initially generate product variants based on specific demands or requirements.

As the demand for specific products or features increases, FOUNDRY can be employed to automate the generation of additional product variants from the transplant of organs extracted from specialized products, effectively evolving it into a product line.

The use of FOUNDRY in the reactive product line adoption process offers several benefits. Firstly, it provides a systematic and automated approach to product line expansion, ensuring consistent and controlled feature integration. Secondly, by automating the transplantation of features, FOUNDRY minimizes the risk of introducing errors or inconsistencies during the adoption process. Furthermore, FOUNDRY can be used to guide a modular and incremental growth of a product line. It allows organizations to incrementally introduce new features or product variants in response to market demands.

Further research can explore and refine the automation capabilities of FOUNDRY in the context of reactive product line adoption. This includes investigating techniques for efficient feature identification and prioritization, optimizing the automation process for generating product variants, and developing strategies for effectively managing the evolving product line.

5.2.4 Automating a *Symbiotic SPL*

FOUNDRY introduces a novel approach to SPL called “symbiotic SPL”, enabling a symbiotic relationship between the donor codebase and the ongoing reorganization of the SPL. In this mode, the donor codebase remains oblivious to the parallel SPL reengineering process, allowing for independent improvements to be made in both the donor and host codebases.

A key feature of FOUNDRY is the ability to capture and incorporate improvements from the donor codebase. Periodically, PRODSALPEL refreshes its set of features by re-transplanting them into the transplantation platform and subsequently into the host

products. This symbiotic approach allows for the integration of updates and enhancements into the SPL, without disrupting the ongoing reorganization efforts.

For example, PRODSICALPEL can be utilized to create lightweight and specialized text editors from the *VIM* project. By extracting and transplanting specific features from *VIM*, PRODSICALPEL can generate new text editors tailored to specific user needs or requirements. This symbiotic SPL approach enables the creation of specialized products while leveraging the ongoing development and improvements in the donor codebase.

The symbiotic SPL paradigm can offer several advantages. Firstly, it allows for a more dynamic and responsive development process, as improvements in the donor codebase can be quickly assimilated into the host products. Such strategy can ensure that the SPL remains up-to-date with the latest enhancements, bug fixes, and optimizations.

Secondly, the symbiotic approach enhances the modularity and maintainability of the SPL. By encapsulating features within transplantable over-organs, the donor codebase can evolve independently, enabling the introduction of new features or improvements without affecting the host products. This modularity promotes code reuse and can facilitate maintenance, and simplifies the management of feature variations.

The symbiotic SPL facilitated by FOUNDRY opens up new possibilities for SPLE. It combines the advantages of independent development in the donor codebase with the systematic and automated feature integration provided by SPL.

5.2.5 Supporting Controlled Maintenance and evolution of SPL

Maintaining assets and products within an SPL poses significant challenges, and these challenges are further amplified when incorporating organs as assets. When individual organs are maintained within the platform, there is a risk of introducing errors into the product line or the derived products. This is due to the shared elements, such as variables and functions, among the maintained organ and other organs within the product line. Re-implanting the changed organ can be a potential solution, but a critical problem arises when attempting to match the changed organ with a different version already transplanted in the target product.

We propose two approaches to address this problem and effectively maintain a created product line. The first approach involves re-transplanting the features if the original source codebase undergoes changes. By doing so, the changes in the organ can be synchronized with the product line, ensuring consistency and avoiding potential errors. The second approach entails maintaining the extracted over-organs and re-running the adaptation and implantation stages as needed. This allows for iterative updates and modifications to the product line, as well as, its continuous evolution and alignment with changing requirements.

To facilitate continuous deployment, the implantation process in FOUNDRY can be configured to involve each organ with feature flags. This practice connects new, unreleased code to the production environment while keeping it hidden from users. Once a transplanted organ is deemed ready for production, developers can disable the feature flag, revealing the new organ or its changes to the users.

5.2.6 Providing a Variability Mechanism through Organ Transplantation

FOUNDRY introduces a powerful variability mechanism that leverages the transplantation of organs into product bases. This mechanism offers developers the flexibility to include or exclude specific features, enabling the instantiation of different products at any given moment by simply transplanting the corresponding organs into the target product.

This approach addresses the challenges associated with maintaining and evolving a product line that consists of a vast number of individual products. Instead of managing and maintaining a multitude of product configurations, FOUNDRY streamlines the process by allowing developers to selectively incorporate desired features through organ transplantation. This variability mechanism eliminates the need for extensive feature toggle management and reduces technical debt related to the presence of unremoved feature toggle.

Moreover, FOUNDRY ensures that the source code structure of the product base remains unchanged during the transplantation process. This preservation of the product base's integrity enables smooth integration of the transplanted organs without introducing unnecessary complexity or conflicts.

The provided variability mechanism brings several benefits to the software engineering process. Firstly, it simplifies the management of feature configurations, making it easier to develop, maintain, and evolve the product line. Instead of manually toggling feature flags and dealing with complex branching structures, developers can rely on organ transplantation to achieve the desired product variations.

Secondly, by avoiding the proliferation of feature toggles, FOUNDRY mitigates technical debt associated with the accumulation of unused or obsolete code. The variability mechanism ensures that only relevant features are included in the product, reducing code complexity and improving overall code quality.

Furthermore, the variability mechanism introduced by FOUNDRY fosters modularity and reusability. The transplantation of organs encapsulates the implementation of specific features, promoting code reuse across different products within the product line. This modular approach enhances maintainability, as changes or updates to a feature can be easily propagated by transplanting the updated organ into the relevant products.

5.3 CHAPTER SUMMARY

In this chapter, we presented PRODSICALPEL, an automated solution designed to address the challenges associated with building product lines through ST. We provided an overview of PRODSICALPEL's architecture, highlighting its key features and functionalities. We also discussed the specific challenges faced in the process of reengineering systems into SPLs and how PRODSICALPEL tackles those challenges. Furthermore, we elaborated and discussed some promising opportunities for SPLE provided by our ST approach.

The following chapter delves into a comparative study conducted to evaluate the capabilities of our approach in supporting the SPL reengineering process, as well as identifying any remaining open issues when compared to existing solutions. The primary objective of this study was to provide initial empirical evidence that supports the claim

that adopting ST for SPL reengineering is a promising research direction.

PART IV

EMPIRICAL STUDIES

COMPARATIVE STUDY ON AUTOMATED SPL REENGINEERING PRACTICES VIA ST

Many contributions (including industrial experiences) can be found in the reengineering literature [7]. Nevertheless, there is a lack of automated approaches covering the whole life cycle of reengineering for a product line [7]. Most existing solutions are responsible for the dependency on expert knowledge, manual labour or by using a combination of multiples tools, which is one of the reasons why the reengineering process is still a laborious, time-consuming, and error-prone task that requires a high upfront investment before the first product is produced from an SPL[44, 115, 7].

We argue that ST can be a feasible technique for migrating existing systems to product lines. However, to be able to explore it as a promising new research direction with application in reengineering for SPL, we need to compare it with existing practices regarding its support to the reengineering process and limitations.

In this sense, we conducted a comparative study in the existing literature to evaluate the feasibility of using the approach from the analyse of: (i) how software transplantation supports SPL reengineering phases, and (ii) how it provides a solution for addressing SPL reengineering open issues in comparison with the existing solutions.

The remained of this chapter consists of four sections. **Section 6.1** introduces the exploratory study goal and research questions. **Section 6.2** gives an overview on existing approaches and main open issues. **Section 6.3** discusses the analysis of the study and threats to validity. **Section 6.4** draws concluding remarks and points out future directions.

6.1 THE COMPARATIVE STUDY

This section presents the design, objectives, research questions and selection criteria considered in our comparative study.

Assunção et al. [7] described many open issues on SPL reengineering research including the following: (i) the implementation of automation and tool support, (ii) the use of different sources of information, (iii) need for improvements in the feature management,

(iv) the definition of ways to combine different strategies and methods, (v) lack of sophisticated refactoring, (vi) need for new metrics and (vii) measures and more robust empirical evaluation [7].

In order to demonstrate the potential to use autotransplantation as an automated solution to address the open issues, we compare it with the current reengineering practices for extracting an SPL from existing code bases. Figure 6.1 shows the comparative study design.

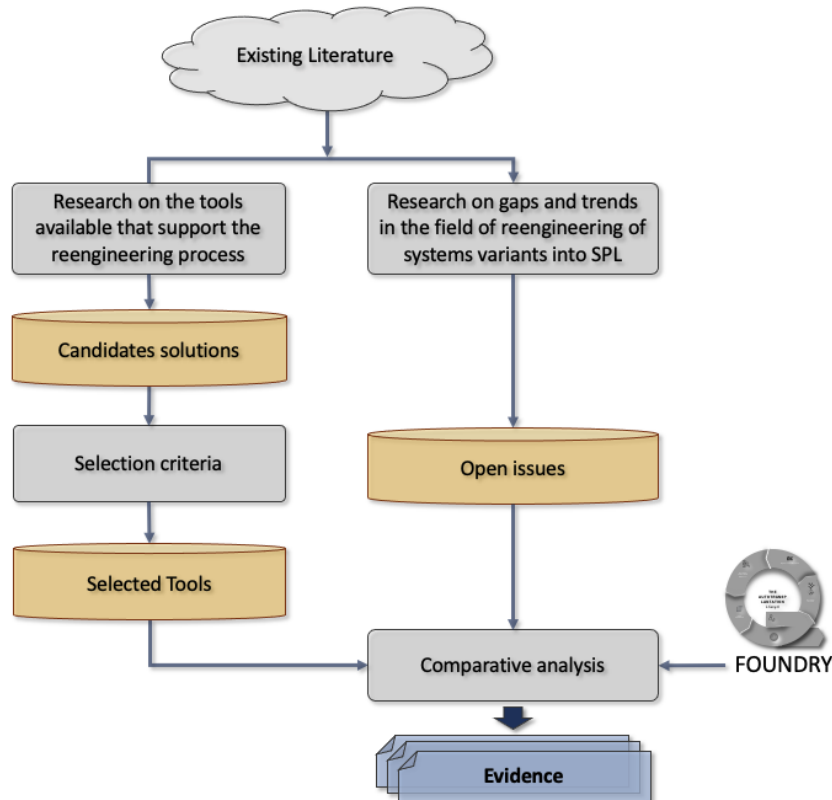


Figure 6.1: Comparative Study Design.

This investigation can support our proposal as it enables us to gain a better understanding and provides evidence of how AutoST can serve as a viable solution for SPL reengineering projects. We acknowledge the significance of conducting new studies to compare our tool with other solutions using real-world systems in the evaluation process. Such studies would allow us to compare the reengineering process and the resulting product line. However, a common challenge lies in the absence of a framework for comparing reengineering approaches [7], especially frameworks that can effectively handle the reengineering of systems implemented in C, as supported by our tool.

6.1.1 Objective and Research Questions

The objective of this study is to analyse and discuss our proposal in comparison with the current practices in the field of reengineering of systems into SPL, thereby demonstrating

the potential of autotransplantation technique for addressing existing open issues. Thus, the following questions were established:

- **RQ1.** *How does multi-organ transplantation (as realised in FOUNDRY) automate existing reengineering practices for extracting an SPL from a codebase?* There already exist a range of tools [7] that support the reengineering of system variants into **SPLs!** (**SPLs!**), generating refactored code as an output. Demonstrating that a new solution can advance the current state of reengineering practices to attain an SPL requires a comparative analysis among the existing solutions. Hence, we have employed the existing reengineering approaches from the literature to address our first research question. We have selected a collection of approaches that also propose automated solutions for the SPL reengineering process and produce refactored source code as the output.
- **RQ2.** *Do the transplantation approach for obtaining SPL address any of the open issues in the field of reengineering of systems variants into SPL? If so, how does it address such challenges?* It is important to understand if AutoST implements any solutions to the open issues identified by [7]. This question aims at analyzing what/how the current limitation of the existing solutions can be addressed by our approach.

6.1.2 Selection Criteria

Our main source of information was the systematic mapping performed by Assunção et al. [7]. According to the authors, from the total of 119 existing studies for guiding the SPL reengineering process, only 19 of them the authors provide automated support to their methods, considering tools that are specific for the reengineering process.

Many proposed tools have focus on more than one phase. Nevertheless, most part of them only covers the *detection* and *analysis* phases (8), *Variability to Aspect tool* [43], *CoDEX Tool* [131], *FeatureMapper* [132, 133], *MapHist Tool* [134], *ExtractorPL* [135], *AUFM Suite* [136], *FMr-T* [64], *ArborCraft* [137]; followed by one (1) that gives support only to the analysis phase, *ETHOM* [138]; and *Model Driven SaaS* [29] tool that provides support only to the *transformation* phase (1). Thus, a total of eight (8) remaining tools cover both three phases, *ThreeVaMar* [62], *Recfeat* [138], *Clone-Different* [139], *SPLevo* [140], *Theme/SPL* [141] *BUT4Reuse* [142], *ECCO Tool* [9], *JfeTkit* [143].

It is important to observe that the transformation phase allows the actual systematic reuse of the artefacts, and source code refactored is the most common outputs [7]. Source code refactored is an output provided to allow a better organization of the features with the SPLE. Thus, in order to select the most relevant set of tools regarding its support of the entire process of reengineering and eliminate studies which do not address the research questions, the following criteria were used to form the final set of tools included:

1. The tool must cover all phases of the reengineering process, i.e., detection, analysis and transformation;
2. The proposed tool must produce source code refactored as output; and

3. Tool considering code as input artefact.

Finally, a total of six remaining tools were selected based on these criteria and compared with our proposal. We analyzed the selected tools by looking at its publication, their documentation such as development documents and user manuals, and available extensions (such as plugins) in those tools which have an extensible architecture.

6.2 TOOL SUPPORT FOR REENGINEERING OF SYSTEMS INTO SPL

The main reason to provide automated support to the reengineering process is to reduce the manual effort [142, 144]. Moreover, an automated process can improve the overall quality of the reengineering process since this process is a labour-intensive task and error-prone [145]. In this sense, authors argue for the necessity of providing tool support, such as [146, 147, 148, 149]. However, in many cases, the reengineering researchers expose only an intention to provide an automated solution to their methods [7]. Further studies should envisage the implementation of tools to automate to support the entire reengineering process.

We present a summary of the tools selected and analysed in this study that were used for our approach evaluation. A brief description of the tools and corresponding work references are presented below:

- **Recfeat** [138]: a prototype tool developed to support the use of the history-sensitive heuristics for the recovery of features in code of degenerate program families. RecFeat tool is used to classify the features' code elements of the selected program families. Once the analysis of the family history is carried out, the feature elements are structured as Java project packages; they are intended to separate those elements in terms of their variability degree;
- **Clone-Different** [139]: a Clone Differentiator tool that automatically characterizes clones returned by a clone detector by differentiating *Program Dependence Graph* of clones. The tool complements clone detection with semantic differencing of reported clones. It is able to provide a precise characterization of semantic differences of clones.
- **SPLevo** [140]: a software development tool that supports the consolidation of customized product copies into a SPL based on program dependencies as represented in *Program dependency graphs*. It reduces the effort of consolidating developers when identifying dependent differences and deriving clusters to consider in their variability design;
- **BUT4Reuse** [142]: (Bottom-Up Technologies for Reuse) a tool-supported bottom-up SPL adoption framework specially designed for genericity and extensibility. This tool provides technologies for leveraging commonality and variability of software artefacts.

- **ECCO Tool [9]**: (Extraction and Composition for Clone-and-Own) automatically locates reusable parts in existing systems and compose a new system from a selection of desired features. It gives support to an approach to enhance clone-and-own that supports the development and maintenance of software product variants. By following this approach, a software engineer selects the desired features, and ECCO finds the proper software artefacts to reuse and then provides guidance during the manual completion by hinting which software artefacts may need adaptation;
- **JfeTkit [143]**: (Java Feature Mining Toolkit) extracts featured code from the software legacy. JFeTkit is a compound system, which uses several existing software analysis libraries, including BCEL (Byte Code Engineering Library), Crystal3 analysis framework and JDT (Java Development Toolkit). JFeTkit collects the information generated using these third-party APIs and annotates software code legacy using a top-down feature mining framework by for SPL proposed in [143].

6.3 RESULTS AND DISCUSSION

Even with numerous researches and advancements, open issues remain in the field of reengineering (with focus on SPL). Assunção et al. [7] identified these research gaps and limitations. From these, they reported the research opportunities and trends uncovered. To answer both of our research questions, we analysed which open issues existing in reengineering practices are addressed by existing tools and compare them with the solution implemented in PRODSICALPEL using ST technique. We summarize the results in Table 6.1.

Table 6.1: Comparison of the PRODSICALPEL with existing reengineering solutions to SPL regarding the strategies used and open issues addressed based on [7].

Tool	Strategies					Input				Open Issues							
	Exp.	Stat.	Dyn.	IR	SB	Test	Req.	Des.	Code.	1	2	3	4	5	6	7	8
Recfeat		✓							✓								✓
Clone-Different					✓		✓	✓	✓		✓	✓		✓		✓	✓
SPLevo		✓	✓						✓				✓				✓
BUT4Reuse	✓	✓					✓	✓	✓		✓		✓	✓	✓		✓
ECCO	✓	✓						✓			✓		✓		✓		✓
JfeTkit		✓							✓							✓	✓
PRODSICALPEL			✓		✓	✓			✓		✓	✓	✓	✓	✓	✓	✓

Open issues:

1. *Automation and tool support*: the first reason to provide tool support for the reengineering process is to reduce the manual effort [142, 144]. Despite the need to automate the entire reengineering process, existing solutions provide support for specific tasks, still requiring manual effort. Additionally, Assunção et al. [7] highlighted that there is still no tool for feature aggregation and abstraction. PRODSICALPEL was implemented to be an automated solution for analysis, detection and transformation

tasks, including with automated support for feature aggregation using GP and clone detection techniques. Our solution automates the process of feature location and dependency handling when the reengineering process requires the transplant of two or more features from the same donor.

2. *Exploiting multiple sources of information for reengineering*: another research gap is exploiting different information sources during the reengineering process. For example, a research opportunity is using test cases, commonly available in most projects, in conjunction with other sources to determine features [61]. Many studies generate feature models as output but, in general, constraints, such as one feature requires or excludes another feature, are not considered [7]. Two of the existing tools use more than one source, *Clone-Different* and *BUT4Reuse*. Our ST solution uses source code as input, but the process feature extraction is also guided by test suite observation.
3. *Feature management*: feature management is an important task in the reengineering process, responsible for providing variability among the features that compose the product variants. Many studies generate feature models as output but, in general, constraints, such as one feature requires or excludes another feature, are not considered [7]. The PRODSALPEL integrates two key techniques, namely *Program slicing* [20] and *Observational slicing in GP* [90], to enable the extraction, reduction, and adaptation of over-organs. Program slicing, a well-established technique in software engineering, is employed to extract the relevant code fragments comprising an over-organ from the donor codebase. This process isolates the specific statements and dependencies necessary for the over-organ's functionality.

In addition to program slicing, PRODSALPEL incorporates observational slicing, a technique based on GP, to further refine and adapt the extracted over-organ. Observational slicing, as introduced in reference [8], leverages GP's capabilities to reduce the size and complexity of the over-organ while preserving its essential functionality. By applying genetic operators and fitness evaluation based on observations of the over-organ's behaviour, PRODSALPEL guides the adaptation process to produce a streamlined and customized version of the over-organ.

4. *Hybrid Approaches*: hybrid approaches can improve the results when compared with the application of only one type of strategy [7]. The combination of techniques is one of the main characteristics of our solution. Regarding the selected solutions, only *SPLevo* consider the combination of dynamic and static analysis strategies. There is still no tool for that consider search-based techniques as a strategy to reengineering process. PRODSALPEL has been implemented to extract and transform features using a form of dynamic observational strategy, close to dynamic analysis, in combination with GP.
5. *Refactoring techniques*: Assunção et al. [7] highlighted the need for new refactoring techniques. The Search-based strategy, for example, has been little explored in the area of SPL [11] and has the potential to exploit as a combination of different

strategies [10]. Regarding new refactoring techniques, our solution explores ST as a new approach to obtain SPL. Regarding the transformation phase, Olszak and Jørgensen point out the labour-intensive task of manually annotating feature entry points [60]. Full automation of the process has been considered unrealistic due to the complexity of the task [150, 151, 152]. Even that existing techniques for locating a feature’s implementation [75, 76, 77], domain experts still need to confirm whether found code fragments belong to the feature and then adapt it to the product line [151].

6. *Need of usage guidelines:* many authors argue the necessity of the creation of guidelines to formalize the tasks of their proposed approaches [7]. In a similar way, Kang et al. point out the need for guidelines for evaluating product line assets [153]. Half of the solutions selected propose some kind of guideline that formalize the tasks predict in their proposed approaches. FOUNDRY provides a detailed guideline for automating the extractive SPL reengineering process. Such guideline can, with suitable tailoring, be applied in a wide range of projects and domains. Additionally, FOUNDRY proposes three validation tasks that together can provide a suitable form to validate the productized products.
7. *New Measures and Metrics:* measures and metrics are important for the reengineering process [7]. Nöbauer et al. [154], for example, exposed the need for a similarity calculation method that allows the identification of commonalities among existing products. By applying FOUNDRY, new measures and metrics can be introduced and integrated into the reengineering process for SPL based on ST.

The possibility of extracting, executing and testing over-organs separately and outside their donor codebase opens up opportunities for extracting new metrics and measurements that provide deeper insights into the behaviour and impact of individual functional features. This capability allows for more granular analysis of the features and their impact on the overall software system. For example, an analysis of individuals of over-organs may be used to calculate similarity scores, identify commonalities among existing products, measure code reuse, assess modularity, analyze feature dependencies, and capture other relevant aspects. These metrics can contribute to a better understanding of the variability, performance, and quality aspects of an SPL.

Additionally, FOUNDRY’s transplantation platform provides a controlled environment where the reengineered codebase and its variations can be analyzed and evaluated. This platform can serve as a data source for applying and validating the new measures and metrics based on organs characteristics, allowing researchers and practitioners to assess the effectiveness of the reengineering process and the impact of different strategies.

8. *More Robust Empirical Evaluation:* Researchers acknowledge the importance of using real case studies. However, the most of authors of tooling support for SPL reengineering expose only an intention to provide empirical evaluation using their approaches in different domains and with complex case studies are [7]. Both FOUNDRY

and PRODSICALPEL have been validated on two case studies using open-source systems (Chapter 7). Additionally, we performed an empirical experiment comparing the performance of our tool in comparison with the performance of SPL experts. Nevertheless, we acknowledge the importance of providing more evidence for them generalisation and investigating its applicability in an industrial context.

In summary, although approaches to conducting the reengineering process with a focus on SPL have been proposed [7], they provide incomplete solutions to transform those single products into an SPL by focusing on part of the process [7]. Many approaches lack the means to consolidate different features present in more than one product, other has not automated support to the phases of the process, or fail by not exploiting multiple sources of code for reengineering [7]. Moreover, the existing solution [9] only cover the reuse of variants from related systems or from the same family what limit the potential of existing codebases reuses for SPL. On the other hand, AutoST for SPL reengineering emerges as an ambitious initiative that could, in the future, facilitate the reengineering of systems even across different languages and platforms.

We believe that with more research and suitable tailoring, AutoST approach can make the reuse of one or more products to generate a new one possible by transplanting features from pre-existing systems. Open source projects, for example, can enable every developer the opportunity to share codes, allowing them to migrate an already existing code from a pre-existing source to their own SPL project. In the same way, software organisations could derive new software products from its existing systems portfolio.

6.4 CHAPTER SUMMARY

This chapter provides a comprehensive comparative study aimed at synthesizing evidence regarding the potential of ST in migrating existing systems to SPLs. By examining existing practices and their limitations, we sought to assess the effectiveness and advantages of ST as an alternative approach.

The comparative study involved an analysis of various factors such as the existing reengineering processes, practices, migration techniques, and the challenges faced in traditional practices. By contrasting these practices with the principles and concepts of ST, we aimed to identify the unique contributions and benefits that ST brings to the field of SPL migration. As outcoming, we were able to highlight the potential of ST as a promising methodology for system migration based on the ST principles.

Next chapter presents a multiple case study that serves as a practical evaluation of the FOUNDRY approach and its supporting tool, PRODSICALPEL. The case study involves the generation of two distinct products from four real-world systems, providing insights into the applicability, effectiveness, and challenges associated with their adoption in real-world context.

THE CASE STUDIES

While various approaches and techniques for reengineering of systems into SPL have been proposed, there is a recognized need for empirical studies that provide concrete evidence of the benefits, challenges, and trade-offs associated with these solutions [7]. Such empirical evaluations can contribute to a deeper understanding of the impact of reengineering strategies on the quality, maintainability, and productivity of software development process.

In this context, this chapter presents a multiple case study [155] conducted to assess the applicability and effort required to generate new product variants from existing systems using FOUNDRY. We validated FOUNDRY and PRODSICALPEL through two case studies involving real-world systems. We employed three donor systems with a focus on assembling two new systems, each specialized with a set of organs transplanted with the support of our tool.

These case studies were influenced by Yin [155] and based on the guidelines defined by Brereton, Runeson and Höst in [156, 157]. These references are important for scientific rigor and possibility for replications. Yin [155] describes case study research as an empirical inquiry that investigates a contemporary phenomenon within its real-life context. *“A case study is a suitable research methodology for software engineering research since it studies contemporary phenomena in its natural context”* [157].

The remaining of this chapter is organized in three sections. **Section 7.1** presents the design of our case studies. **Section 7.2** discusses and results of our empirical study, including the threats to validity. Finally, **Section 7.3** concludes the chapter.

7.1 CASE STUDIES DESIGN

Case study research involves an in-depth investigation of a particular phenomenon within its real-life context, focusing on specific cases or instances. In our study, we aim to evaluate our proposed approach and tool for generating new product variants from real-world systems. By examining multiple product bases and donors, we can assess the feasibility and effectiveness of the transplantation process for SPL reengineering across different

systems and domains. Furthermore, our research questions involve the exploration of specific aspects related to the case, such as effort estimation, characteristics of features, and current limitations. This indicates a focus on examining and understanding a particular case or set of cases in depth, which is characteristic of a case study research approach.

Yin [155] defined four distinctions in designing case studies: (1) *single* or (2) *multiple-case* designs which involve the number of case studies to be carried out and (3) *holistic* or (4) *embedded* which involves the number of units of analysis to be studied within a case study. The choice between holistic and embedded case study depends on the research goals of the study.

Based on the [158], our case study follows a multiple-case design within a holistic design approach. Multiple-case design because the study involves two text editors, *VI* and *VIM*, as the product bases, and multiple donors from different domains, including code analysis software (*GNU Cflow*) and other text editors (*kilo*, *VI*, and *VIM*). By examining multiple cases (product bases and donors), we could compare and evaluate the effectiveness of the transplantation process across different systems and domains. It is a holistic design by considering the entire case as a whole, including all relevant aspects of the transplantation process. They focus on the evaluation of the proposed approach and tool by generating new product variants from real-world systems using ST for SPL reengineering. The features identified for transplantation, the preparatory steps (removing dead-code and reducing hosts to their basic form), the transplantation procedure, and the experimental environment are all part of the holistic design approach.

According to Herriott and Firestone [159] stated in [155], “*The evidence from multiple cases is often considered more compelling, and the overall study is therefore regarded as being more robust*”. A multiple case study approach increases the external validity of the research through the implied “replication” inherent in its design [155].

The replication approach to multiple-case studies is illustrated in Figure 7.1. It indicates that the two initial steps in designing the study consist of theory development and then shows that case selection and the definition of specific measures are important steps in the design and data collection process. Based on the design defined, each case is individually performed. Each individual case study consists of a whole study, in which convergent evidence is sought regarding the facts and conclusions for the case; each case’s conclusions are then considered to be the information needing replication by other individual cases. For each individual case the preparation, data collection, and analysis are carried out. In the final of each study, an individual case report is written. Finally, all individual cases are analyzed and the research may be concluded. This analysis consists in cross-case analysis where patterns are searched and conclusions can be inferred. The loop represented by dotted line corresponds to feedback which may occurs in situations where important discovery occurs during the conduct of one of the individual case studies - for instance, one of the cases did not in fact suit the original design.

Another characteristic of research methodology is related with its flexibility. According to (Robson, 2002), the research process may be characterized as *fixed* or *flexible*. In a fixed design process, all parameters are defined at the beginning of the study. In a flexible design it is possible to modify or customize these parameters. It may occur when the case study is being applied, and new information is provided, or when new data is gathered. For our

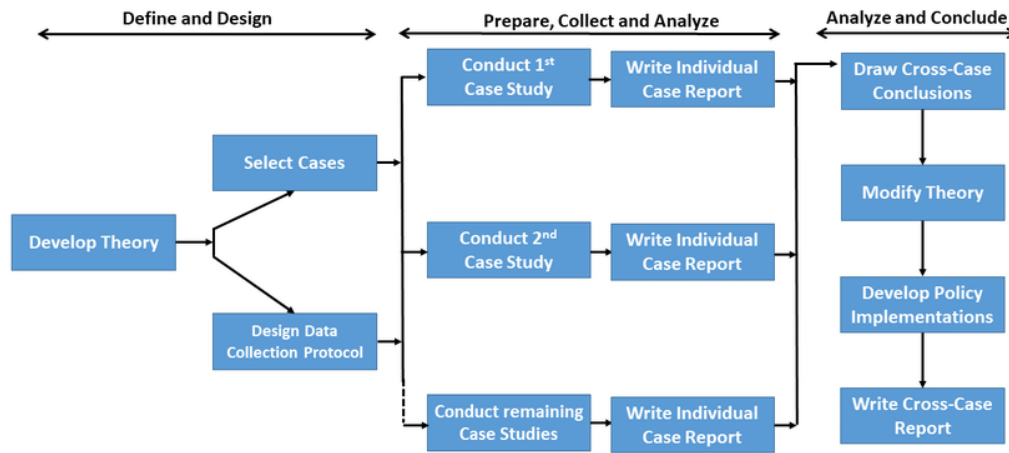


Figure 7.1: Case Study Method

The figure illustrates the case study method as described by Yin in [155], providing a visual representation of the key components and stages involved in conducting a case study research in software engineering.

research, we applied the fixed design option.

During the case study design process, a protocol was created. This protocol was influenced by [156] and complemented with the advises presented by Yin in [155], and Runeson and Höst in [157]. The protocol described the execution of the case study, what methods would be used for data collection and what analysis method would be used. The first version of the protocol was developed and revised with members of the RiSE Labs¹. It is important to highlight that these researchers did not participate in the research. The main improvements were related to define what methods would be used for data collection and analysis.

7.1.1 Objective and Research Questions

The primary objective of this research study is to evaluate the proposed approach and tool for generating new product variants from real-world systems, thereby demonstrating the feasibility of ST for SPL reengineering. To achieve this objective, we have formulated the following research questions:

RQ1. *How much effort is required to generate products from a product line created using PRODSALPEL?* Considering the inherent complexity of the task, it is unrealistic to expect the product derivation process to be instantaneous. However, it should be efficient enough to be seamlessly integrated into the development cycle and exhibit advantages over manual efforts. To answer this question, we have measured the time required for the

¹<http://labs.rise.com.br/>

transplantation process and quantified the number of lines of code (LOCs) transferred.

RQ2. *Which features can PRODSALPEL effectively transplant?* In this study, we investigate the characteristics of features that can be successfully handled by PRODSALPEL, as well as identify any limitations it currently faces. Given the inherent complexity of the transplantation process, it is crucial to establish a clear understanding of the current capabilities and limitations of PRODSALPEL in handling different types of features.

By addressing these research questions, we aim to gain valuable insights into the time, effort and limitations of the proposed approach and tool for generating product variants through ST, ultimately contributing to the advancement of SPL reengineering.

7.1.2 Subject Selection

To define the context for performing the case study research, the first step was to identify the most suitable codebases for composing the product lines. The selection of subject systems for our case studies was based on four criteria aimed at providing a comprehensive evaluation of the proposed approach and tool. These selection criteria were based on the convenient sampling method Wohlin et al. [160].

Diversity and variation: The selection of subjects should aim to encompass a range of diversity and variation to provide a comprehensive understanding of the phenomenon under investigation. This can include diversity in terms of domains, sizes, complexity, functionality, or other relevant characteristics.

Representativeness: The selected subjects should be representative of the broader population or the specific context under study. They should capture the characteristics and variations found in the target population or application domain.

Feasibility and accessibility: Practical considerations, such as availability of data, availability of use and modification, access to organizations or systems, and resources required for data collection, should be taken into account during subject selection.

Established systems: Established systems are typically used in real-world scenarios and have a significant user base. Studying such systems can provide insights into practical challenges and solutions.

Based on these criteria, we present the systems used in Table 7.1. We chose two text editors, *VI* and *VIM*, as product bases. These text editors serve as the foundation upon which the transplantation process is performed. By selecting widely used and established text editors, we increase the relevance of your case study.

To illustrate that donors can come from different application domains, we included *GNU Cflow* one of the donors. *GNU Cflow*, being a call graph extractor from C source code, represents a distinct domain compared to the text editors. This demonstrates that the transplantation process can be applied to diverse types of software systems. In addition to using donors from different domains, we also included *VI* and *VIM* as donors as well as another text editor, *kilo*. This highlights the possibility of using donors from the same system as the product base or from different systems within the same application domain. It showcases the flexibility of the transplantation process and its ability to handle variations within a specific domain.

Table 7.1: Donors and hosts corpus for the evaluation.

Column Features shows the number of features identified.

Subjects	Type	Size (LOC)	#Features
Kilo	Donor	804	17
CFLOW	Donor	4,274	54
VI	Donor	20,292	36
VIM	Donor	839,438	176
VI	Product Base	20,223	36
VIM	Product Base	737,466	117

By selecting different subject systems allowed us to assess the effectiveness, applicability, and versatility of PRODSALPEL in achieving products. We used subjects from different domains and with a wide range of sizes to give evidence that PRODSALPEL can also be used to achieve a product line from a distinct set of usage scenarios.

We identified the following features as possible desired features in a new editor: `output` from *CFLOW*, `enableRawMode` from *kilo*, `vclear` from *VI*, and `spell_check` and `search` from *VIM*.

7.1.3 Procedure and Execution

In the initial phase of the study, PRODSALPEL was employed to automatically eliminate dead-code from both the donors and host codebases. Additionally, the host codebases underwent a reduction process where optional features were removed, resulting in a simplified version. This preparation phase ensured that both the donors and host were ready for the transplantation process.

Subsequently, the researchers introduced organ entry points in the donor systems and each target implantation point in the product base. The researchers also implemented corresponding test suites for each organ. Thus, PRODSALPEL was executed to facilitate the localization and extraction of organs from the donor. Furthermore, it was used to automatically transform each organ, while ensuring compatibility with the specific context of their target sites in the product bases and subsequently implanting them in the beneficiary's environment.

The automated organ transplantation process was repeated 20 times to account for the heuristic nature of the over-organ adaptation process. The runtimes were measured on a system with an Intel Core 3.1 GHz Dual-Core Intel Core i5 processor, 16 GB of memory, running MacOS 10.15.4.

The study measured the average number of lines of code transplanted and the average runtimes for the transplantation process. The measurement of the average is justified by the inherent variability that can arise from the heuristic nature of the over-organ adaptation process performed by GP algorithm. Since the process involves heuristics and adaptations, each organ transplantation may result in a different number of code lines being transplanted and varying runtimes. This variability can stem from factors such as the complexity and compatibility of the organs with the target sites, the specific context

of the transplantation, and the interactions between the transplanted organs and the beneficiary's environment.

By measuring and reporting the average number of lines of code transplanted and the average runtimes, the study captures this inherent variability and provides a more comprehensive understanding of the overall outcomes and performance of the transplantation process. It allows for the identification of tendencies and general characteristics of the process, while also acknowledging that individual transplantation instances may exhibit different results.

7.2 RESULTS AND DISCUSSION

In this section, we present the results of our study and provide a discussion and interpretation of these results. We address each of the research questions outlined in the study and offer insights and implications based on the presented findings.

7.2.1 Results

The case studies provided initial evidence that FOUNDRY implemented by PRODSALPEL can be successful in automatically building product variants by combining features from real-world systems. Table 7.2 displays the average runtime and the number of code lines transplanted for each organ during the transplantation process. The postoperative products, labeled as Product A and Product B, have 28k LOC with 40 features and 745k LOC with 121 features, respectively. The donors contributed three feature variants to the product line, resulting in approximately 7.8k LOC for Product A and 8.1k LOC for Product B. It is worth noting that one feature was removed and re-transplanted in the VI editor.

PRODSALPEL successfully integrated features from donors into two product lines, resulting in the generation of distinct product variants.

To address research question **RQ1**, the study focused on analyzing the average time required by PRODSALPEL to transplant the three features into both the VI and VIM editors. The transplantation process took an average of 4 hours and 31 minutes per 1KLOC for the VI editor, and 4 hours and 40 minutes per 1KLOC for the VIM editor. These findings provide insights into the time efficiency of the transplantation process and highlight the comparable effort required for transplanting features in different product lines.

On average, PRODSALPEL spent 4h31min/1KLOC for transplanting three features into VI, and 4h40min/1KLOC for transplanting the same three features into VIM, demonstrating the effort required for generating products from a product line created using PRODSALPEL.

Table 7.2: Case studies results

Multi-organ transplantation results to generate products A and B. *Columns Trans. Time shows the time (Sys+User) spent on the organ transplant process in which column Sys. correspond to the PRODSICALPEL’s execution time; column User correspond to the user time spent in preoperative stage and augmenting the host’s regression test suite (regression++). Product A uses a reduced VI editor as a product base, while Product B was derived by transplanting features from the donors into the reduced VIM editor.*

Donors	Number of			Trans.time(min)	
	LoC	Functions	Files	Sys.	User
Kilo	~963	35	4	~86	32
CFLOW	~4,822	37	8	~344	123
VI	~1,983	5	15	~1,234	184
Product A	~7,768	77	27	~1,664	339
Kilo	~981	35	4	~94	32
CFLOW	~4,898	37	8	~428	123
VI	~2,234	5	15	~1,294	184
Product B	~8,113	77	27	~1,890	532

In response to research question **RQ2**, an additional transplantation processes were conducted to transplant two more features, namely `spell_check` and `search`, which encompassed a substantial amount of code (104 KLOC and 153 KLOC, respectively). However, the effectiveness of PRODSICALPEL in this regard was influenced by its reliance on Doxygen [120], a source code documentation generator, for generating call graphs. Limitations inherited from the underlying slicing tools, specifically the imprecise generation of call graphs when handling function pointers, presented challenges for PRODSICALPEL in automatically extracting these larger organs from VIM. Although a significant part of over-organ sliced is pruned from over-organ during GP-refinement, it continues retaining unnecessary instructions. How future work, we can explore more precise techniques, such as *Dynamic analysis* [55] and other code manipulating tools, to enhance the efficiency and accuracy of the slicing process.

While manual efforts could potentially overcome these limitations, our tool’s current capabilities prevented fully automated extraction of large organs implemented with functions pointers.

Although PRODSICALPEL is able to successfully transplant smaller organs from donors into the product base, faces limitations when dealing with larger organs due to imprecise call graphs, suggesting the need for further improvements and investigation into more precise slicing techniques.

7.2.2 Discussion

The study's results highlight the contributions made by the donors in terms of feature variants and lines of code added to the product line. The donors, namely *Kilo*, *CFLOW*, and *VI*, provided feature variants that were successfully transplanted into the product base, resulting in the generation of two distinct products, A and B.

Product A, derived from the reduced *VI* editor, incorporated three feature variants from the donors. These feature variants added approximately 7.8k lines of code to the product, enhancing its functionality and expanding its feature set. The transplantation process involved careful extraction, transformation, and implantation of the organs to ensure compatibility with the product base.

Similarly, Product B was derived by transplanting features from the donors into the reduced *VIM* editor. This resulted in the incorporation of three feature variants into the product, contributing around 8.1k lines of code. The transplantation process, similar to Product A, involved the localization, extraction, and transformation of the organs to seamlessly integrate them into the *VIM* editor.

These results allow for an evaluation of the effectiveness of the transplantation process in incorporating feature variants from different donors into the product line. It demonstrates the feasibility of integrating features from diverse sources and showcases the potential for creating product variants with enhanced functionality by leveraging existing systems.

In the context of software engineering context, the ability to transplant features from donors across different domains and integrate them into a product base opens up possibilities for reusing existing functionality and accelerating product development. This approach reduces the need for manual effort in feature implementation, saving time and resources while maintaining the integrity of the resulting products. However, it is important to acknowledge the limitations encountered during the transplantation process, such as the imprecise call graphs when dealing with function pointers, which affected the extraction of larger organs from the *VIM* editor. This highlights the need for future research and improvements in refining the implementation of program slicing technique and overcoming such limitations.

In conclusion, the study's results demonstrate the successful integration of features from donors into the product line, resulting in the generation of distinct product variants. This showcases the potential of ST as an effective approach for product development and highlights avenues for future research to enhance the precision and efficiency of the transplantation process.

7.2.3 Threats to Validity

The relatively small number and diversity of systems used for transplantation pose an external threat to validity. However, we tried to mitigate it by constructing a possible real-world scenario, i.e., transplantation of features that would be useful in production of new text editors.

In terms of internal threats, we are limited by abilities of the tools we use, in particular, Doxygen, for call graph construction, and TXL for program transformation. We also use

testing as a means of validating our approach, which cannot provide a formal proof of its correctness. However, testing is a standard approach in evaluating code in real-world scenarios due to its high scalability.

Additionally, our case studies are limited to a specific implementation technique and specific code-transplantation scenarios; generalization to object-oriented languages and others requires further investigation.

7.3 CHAPTER SUMMARY

We conducted a multiple case study to assess the viability of the proposed approach by generating two product variants through the transplantation of features extracted from three real-world systems. Although preliminary, the findings of our study are promising. Our research provides initial evidence that that `FOUNDRY` as implemented by `PROD-SCALPEL` is a feasible approach for developing product variants with minimal human intervention.

The subsequent chapter concludes the experimental evaluation, which aims to analyze the effectiveness and efficiency of our approach in comparison to the manual process of deriving software products from existing systems, as carried out by experts in SPL.

EXPERIMENTAL EVALUATION

In the previous case study, we showed the viability of our software transplantation approach, implemented in PRODSICALPEL, to generate software product lines from the existing codebase. Although the validation of the approach is based on empirical evidence, it is still important to test its efficiency by comparing it with other tools used to product line migration. Unfortunately, tool support for the reengineering process is limited, in general, they give support for specific activities, such as feature location, refactoring or quality assurance [7, 99]. To the best of our knowledge, there is currently no comparable tool that manages to transplant features from distinct donors systems written in C to generate a product line, remaining to us analysing it concerning to human effort. Thus, we conducted an experiment based on Wohlin et al. [160], that reflects a real-world process of product line migration from existing codebases [35]. The obtained results make us to appreciate as a promising approach.

The remainder of this Chapter is organized into four sections. In **Section 8.1**, we detail the experiment design by stating our research objectives, research questions, metrics, instrumentation, hypotheses and methodology applied. **Section 8.1.8** highlights the data collecting process while **Section 8.2** discusses and results of our empirical study. Additionally, this Chapter includes a subsection for the threats to validity **Section 8.3**. Finally, **Section 8.4** presents the Chapter summary.

8.1 EXPERIMENT DESIGN

The design of our experiment follows the guidelines presented in Wohlin et al. [160]. we detail the experiment design by stating our research objectives, research questions, metrics, instrumentation, hypotheses and methodology applied. The stages of the experimental study are: design, preparation, collection of data, and analysis of data.

8.1.1 Goal

The goal of this experiment is to analyse the effectiveness and efficiency of our approach compared with the manual process of generating a product line from existing systems,

performed by SPL experts. In accordance with the guidelines for reporting software engineering experiments presented in [161], we have framed our research objectives using the Goal Question Metric (GQM) method suggested by Basili [162]. Our goal is to:

Analyse of a software transplantation approach to derive product variants **for the purpose of comparison with respect to** effectiveness and efficiency **from the point of view of** the researcher **in the context of** an SPL project of product line migration from real-world systems.

8.1.2 Research Questions

In order to achieve the stated goal, we defined two quantitative questions. These are related to the data collected during the period that the experiment was executed. The questions are described as follows:

RQ1. *What is the accuracy of the proposed approach for automated migration of target features to a product line?* In this respect, we would like to understand how accurate our approach is when automatically transferring all required code so that the target feature can run in an emergent product line compared to the manual process.

RQ2. *How much feature migration time can be gained using PRODSICALPEL compared to the manual process?* With this question, we evaluate the time spent by SPL experts to *extract*, *adapt* and *merge* features to derive new product variants in comparison with the same process using PRODSICALPEL.

8.1.3 Metrics

With the objective to answer the previous questions, we defined the metrics that must be computed. For each question, it was defined one metric. These are described as follow:

M1. For the first question, the accuracy of our approach is computed by verifying if PRODSICALPEL successfully migrated new functionalities to a product line and it passed in all the regression, augmented regression and acceptance test suites. Together, these our test suites check whether or not the output of the transplanted feature is correct with respect.

M2. For the second question, it is simply tracked down the time that is spent with the activities to transfer the target features. Examples of these activities are *code extraction*, *adaptation*, and *merging*. The time for each of these activities was individually collected.

8.1.4 Instrumentation

According to Wohlin et al. [160], the instruments for an experiment are classified in objects, guidelines, and measurement. The object instruments of the experiment are two donor systems: *NEATVI*¹, vi/ex editor for editing bidirectional UTF-8 text and *Mytar*², an archive manager besides another version of *NEATVI* used as product base are used in this experiment.

¹<https://github.com/aligrudi/neatvi>

²<https://github.com/spektom/mytar>

Manually inspecting code to transfer a feature to a product line is challenging, time-consuming, and potentially tedious [163]. Therefore, we opted to work with small codebases to prevent participants from becoming fatigued. Otherwise, the experiment would necessitate a significantly longer execution time.

Table A.1 gives more details about the object instruments used in this experiment. These objects were available to download together with a script to automatic setup of the environment.

Table 8.1: Experiment instrumentation

Scenario	Donors	LoC	Target features	LoC	Host	LoC
I	NEATVI	5,276	DIR_INIT	239	Product base	5,285
II	Mytar	1,046	WRITE_ARCHIVE	170		

We recruited 20 SPL experts for the experiment that were allocated in two different groups. We chose to allow participants to use their own work environment by avoiding adaptation bias to a strange environment with the use of unknown tools.

Given this experiment involves subjects guidelines were needed to guide the participants in the experiment. It includes a process description and systems documentation. In addition to the guidelines, we provided a soft training on re-engineering to SPL and clone-and-one by assure that all participants have the same idea about the experiment objective.

For the measurement instruments, we used time sheets to track down the effort spent with the necessary activities to transfer features from existing codebase to a product base, as previously mentioned. In addition to the time sheets, we applied two forms used to collect information about the experience of subjects is shown in Table 8.2 and a post-survey used to better understand participants difficulties. Further details on the operationalisation and instrumentation of this construct is presented later in the paper.

8.1.5 Hypotheses

Null hypothesis. Our null hypothesis determines that there is no benefit of using the FOUNDRY. That is, our approach cannot transplant features to generate product variants with better accuracy than manual process or the payoff is not worth. The null hypothesis is specified as follows:

$$H_0: \mu(\text{accuracy with our approach}) \leq \mu(\text{accuracy with manual process})$$

$$\mu(\text{payoff with our approach}) \leq \mu(\text{payoff with with manual process})$$

Alternative hypothesis. The alternative hypothesis of this experiment determines that our approach is a better option than manual approaches. That is, the proposed approach has higher accuracy and payoff. The alternative hypothesis is specified as follows:

$$H_1: \mu(\text{accuracy with our approach}) > \mu(\text{accuracy with manual approaches})$$

$$\mu(\text{payoff with our approach}) > \mu(\text{payoff with manual approaches})$$

8.1.6 Methodology

We answered our research questions by simulating an authentic reengineering process in which two features needed to be transferred to a product line built on top of a product base. The experiment design drew inspiration from documented real product-line migration scenarios [48, 7]. In Scenario I, we assembled a group of 10 SPL experts, each tasked with manually transferring all portions of code that implement the feature

Before the simulation process, we conducted two pilot studies with 6 graduate students. We used the pilot study results to determine the amount of time needed to execute our tasks and the suitable size of features. This allowed us to estimate and plan the number of participants we needed in the main study.

The pilot study also allowed us to assess whether the participants could properly understand the subject systems and the tasks they should perform. We do not consider the results of the pilot in our analysis.

8.1.7 Participants

After the pilot phase, we recruited a total of 20 participants (excluding the pilots): 2 undergraduates (Un), 9 master’s students (M), and 9 Ph.D. candidates (PhD). The majority of participants have over 5 years of experience in software product lines (SPLs) and over 10 years of experience in software development. These participants come from ten distinct universities (U1 to U10), and among them, analysts and developers are associated with four different companies (C1, C2, C3, C4, and C5). To enroll these participants, we reached out via email to professors from two universities who were affiliated with various software reuse research groups, seeking recommendations for both current and former members.

Before the experiment, we asked them to answer an online survey, which we used to collect background data about their experience, mainly in software development and SPL. According to our design, we created balanced groups(A and B) of participants to each product line generation scenario based on their experience. Table 8.2 shows the details of the participants involved in the experiment.

8.1.8 Operation

Before the participants receive their tasks, we introduced the experiment with a *tutorial* about clone-and-own and reengineering of existing systems into SPL. The tutorial took 30 minutes on average.

We provided the participants with the same input as the one required for FOUNDRY, namely: feature entry points in the donor, a set of automated unit testing, the donor’s source code, and a prepared product base with the target insertion point. Additionally, they received a few-sentence description of each feature in the target system and the system’s documentation with donor and host feature models. From those artifacts, they get domain knowledge about the systems.

The direct costs of this experiment are related solely to the time spent by the re-

Table 8.2: Details of participants' expertise (in years) and division into groups.

Group A worked on scenario I, transplanting a feature from different versions of the same donor system as the host. Group B worked on scenario II, transplanting a feature from a donor system different to the host one

Group	Part.	Degree	Inst.	Exp. (years)	
				Dev.	SPL
A	P1	MSc	U7	[1,5)	[5,10)
	P2	Un	C5	[10)	[1,5)
	P3	PhD	U4	[10)	[10)
	P4	MSc	U4	[10)	[1,5)
	P5	MSc	U4	[10)	[5,10)
	P6	MSc	U4	[10)	[5,10)
	P7	PhD	U8	[10)	[10)
	P8	PhD	U9	[10)	[1,5)
	P9	PhD	U10	[10)	[10)
	P10	PhD	U4	[1,5)	[1,5)
B	P11	MSc	C1	[10)	[1,5)
	P12	MSc	C2	[1,5)	[1,5)
	P13	Un	C3	[10)	[1,5)
	P14	PhD	U1	[10)	[10)
	P15	PhD	U2	[10)	[10)
	P16	PhD	U3	[10)	[5,10)
	P17	MSc	U4	[5,10)	[5,10)
	P18	MSc	C4	[5,10)	[5,10)
	P19	PhD	U5	[10)	[10)
	P20	MSc	U6	[5,10)	[1,5)

searcher with setup of the experiment itself. This involved: specifying the respective annotations for the entry point and insertion points of the features, which took approximately 13 minutes of work at the scenario I and 17 minutes at scenario II; creating the test cases, necessary to validate the target features, taking approximately 16 minutes of programming activities; preparing the product base, which took approximately 14 minutes; then, approximately 34 minutes were spent creating all documentation of donor systems including the product base feature model.

To extend the number of product base variants with a new feature transferred from the correspondent donor system, all participants had three activities based on clone&own and the migration of cloned variants to a product line [44, 163]: feature *extraction*, *adaptation* and *merging*, with descriptions and instructions provided for each task.

Initially, participants are required to identify and extract all code associated with the feature of interest to a temporary directory. Each segment of code identified as pertaining to the target feature must be enclosed within `ifdef` directives. For instance,

We have provided a task and time registration worksheet. While participants were running the experiment, we ask them to take notes of which strategies were being used for each stage of the features transfer process and why they are performing each specific task. It allowed us to capture strategies and performance data simultaneously.

We have complemented the above setup with a post-survey. By post-survey, we could better understand participants' difficulties and meaningful differences about the manual and automated process in both scenarios. We have triangulated the data generated with the experiment with the responses we obtain from the pre and post-survey.

To establish the time for feature transplantation using our automated approach concerning manual effort, we ran PRODSICALPEL 20 times, and measure the average time transplanting the same feature used by the participants in each scenario. This average time was compared with the time spent by our participants on the manual re-engineering process.

Based on our pilot study, we set a time limit of 4 hours for each manual and automated process. Another way, the result might be affected negatively by participants' boredom and tiredness.

Our corpus and data collected are available at the project webpage [164].

8.2 RESULTS AND DISCUSSION

We used 22 pre-existing regression test suite designed by the *NEATVI* developers to assess the accuracy of PRODSICALPEL and answer our **RQ1**. However, they were not designed to test *NEATVI* as a product base with new variants and may not be sufficiently rigorous to find regression faults introduced by the re-engineering process. To achieve a better product line coverage, we manually augmented the host's regression test suites with additional tests, our augmented regression suites.

Furthermore, we implemented an acceptance test suite with 3 tests cases for evaluating the transferred feature in the scenario I and II for evaluating the feature in scenario II. Thus, we executed a total of 27 tests in the scenario I and 29 in scenario II, considering the acceptance tests already incorporate into the product base by the first variant inserted in scenario I.

We summarise our results in Table 8.3. We report the status of the product base and variant inserted by the participants, the time spent and the number of passing tests for the regression augmented regression and acceptance test suites. In the first scenario, only one of the participants was not able to finish the process before the timeout. On the other hand, half of the participants were able to finish the process before achieving the timeout in the second scenario and only three of them have been able to insert the target feature without breaking the product base.

For each scenario, we also report the number of PRODSICALPEL runs in which the product derived passed in all test cases in the successful running. For each scenario, we repeat each run 20 times. The success rate was retained for both scenarios I and II, where we lost only one successful run in the timeout and the product line generated passed in all tests in all test suites.

Table 8.3: Scenario I: Donor NEATVI - Host Product Base

Experiment results comparing the time of tool over 20 repetitions with the participants: column product line status shows the generated product line status by participants and tool; column Execution Time shows the time spent on the feature transplant by the participants and the average time of 20 run of PRODSALPEL, we highlight the execution time of the participant that spent less time; columns Test Suites shows the results for each test suite and report statement coverage (%) for the postoperative host and for the organ; columns PASSED report the number of passing tests.

Participants	Product Line Status	Execution Time (minutes)	Test Suites		
			Regression (59%) PASSED	Regression++ (70.1%) PASSED	Acceptance (72.5%) PASSED
P1	OK	82	22/22	30/30	3/3
P2	OK	88	22/22	30/30	3/3
P3	OK	77	22/22	30/30	3/3
P4	OK	68	22/22	30/30	3/3
P5	OK	81	22/22	30/30	3/3
P6	Broken	Timeout	0/22	0/30	0/3
P7	OK	87	22/22	30/30	3/3
P8	OK	83	22/22	30/30	3/3
P9	OK	73	22/22	30/30	3/3
P10	OK	113	22/22	30/30	3/3
prodScalpel	OK in 20/20 runs	20	22/22	30/30	3/3

Table 8.4: Scenario II: Donor Mytar - Host Product Base

Participants	Product Line Status	Execution Time (minutes)	Test Suites		
			Regression (70.1%) PASSED	Regression++ (71.9%) PASSED	Acceptance (73.3%) PASSED
P11	Broken	Timeout	0/22	0/33	0/2
P12	Broken	Timeout	0/22	0/33	0/2
P13	Error	181	0/22	0/33	0/2
P14	Broken	Timeout	0/22	0/33	0/2
P15	Broken	Timeout	0/22	0/33	0/2
P16	Error	114	0/52	0/33	0/2
P17	OK	104	22/22	33/33	2/2
P18	OK	194	22/22	33/33	2/2
P19	OK	131	22/22	33/33	2/2
P20	Broken	Timeout	0/22	0/33	0/2
prodScalpel	OK in 19/20 runs	27	22/22	33/33	2/2

To answer **RQ1**, the results show success of rate was retained for both scenario I and II, where we lost only one successful run in the timeout and all products derived passed in all test in all test suites.

As stated in the definition of the metric **M2** and to answer **RQ2**, we evaluate the pay-off of FOUNDRY. Figure 8.1 graphically shows the time spent on each activity performed in re-engineering to SPL process. In summary, Group A transferred the target feature from *NETVI* to the product base in 1h24 minutes on average. PRODSALPEL turned out

to be quicker, successfully transplanting this feature in all 20 trials, in an average of 20 minutes.

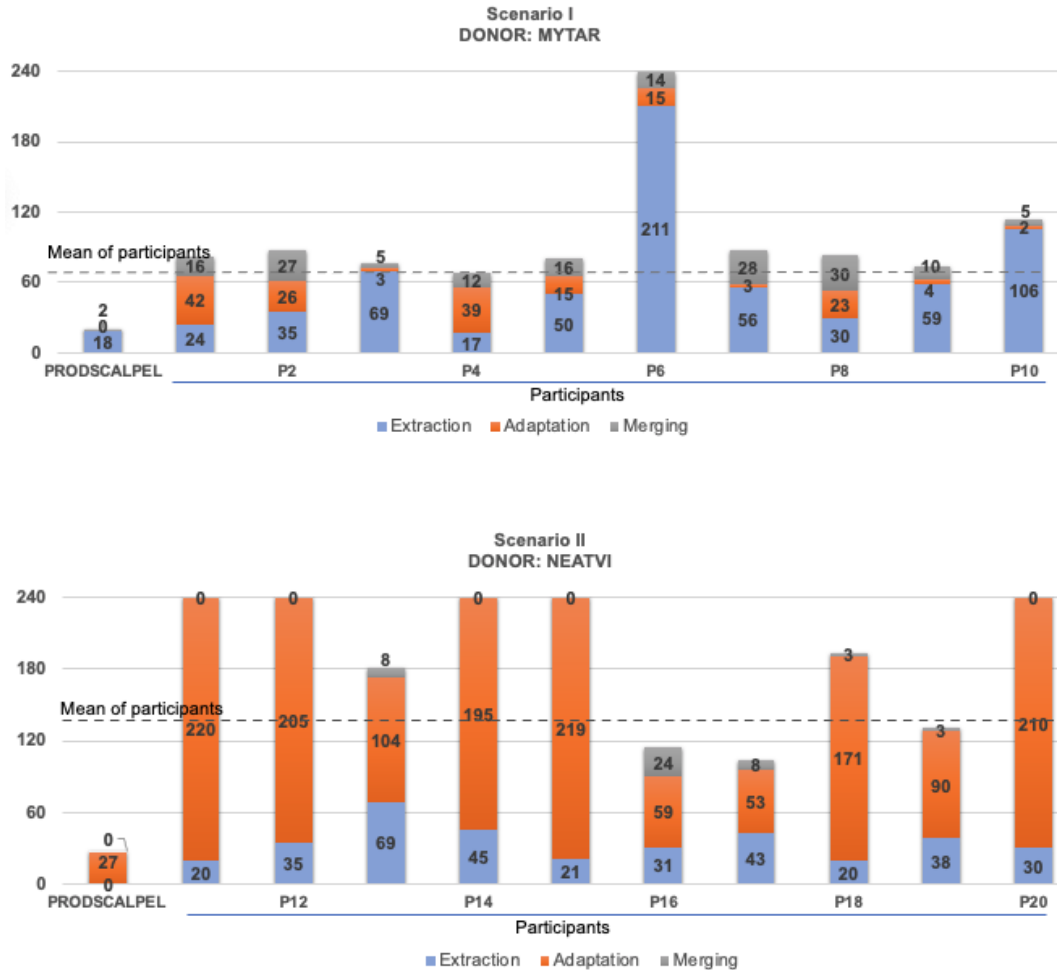


Figure 8.1: Time (in minutes) spent by participants and PRODSICALPEL on performing the three stages of SPL reengineering.

Feature extraction, adaption and merging. The graph highlight the average participants time that successful generated its target product line.

Most of the participants of Group B had not completed the product line generation process from *Mytar* within the 4 hours time limit. Considering the participants that were able to finish the process (i.e., participants *P17*, *P18* and *P19*) and its product line passed in all test suites they spent an average of 2h23 minutes while PRODSICALPEL was able to complete this task in 19 of 20 trials in the timeout, taking 27 minutes on average.

Statistic analysis of performance. The quantitative analysis strategy from the experiment was adapted to this study because there is no comparison between scenarios since used different donors and target features. The first analysis step consists of identifying the statistical distribution for each scenario through the Shapiro-Wilk [165], a method with the best power for a given significance [166]. Its null hypothesis claims the

population is normally distributed. It determines which set of methods must be applied in the hypothesis testing and the strength of association between variables.

Figure 8.2 graphically shows the time results for our two groups in comparison with PRODSICALPEL performance. In the scenario, I, the preliminary information provided by the box plots indicate that all samples are normally distributed ($W = 0.70445$, $p\text{-value} = 3.129e-05$). Thus, a ANOVA [167] and Pairwise Student's t-test analysis were conducted considering the hypothesis of the time values for PRODSICALPEL have statistically higher values if compared to the manual methods ($p\text{-value} < 2e-16$). It led to the rejection of the null hypothesis for all pairs.

In the scenario II, the normality test result showed a normal distribution with a $W = 0.69378$, $p\text{-value} = 1.715e-06$. Thus, we used ANOVA to hypothesis testing and Pairwise t-Student. Based on the ANOVA test and Pairwise t-Student, we rejected the null hypothesis ($p\text{-value} < 2e-16$) that the distribution of the population is homogeneous.

We can concludes that PRODSICALPEL reduces developer effort to transfer features to a product line in both scenarios. For both simulation scenarios, there is a significant effect size between the tasks performed in a manual way and using PRODSICALPEL. The participants had similar performance times in scenario I, with the exception of the participants *P6*. On the other hand, most of the participants of scenario I do not terminate the experiment before the 4-hour timeout. This last one is qualitatively explained by the participants in the post-survey where they exposed how hard is to adapt a feature to run in a strange codebase.

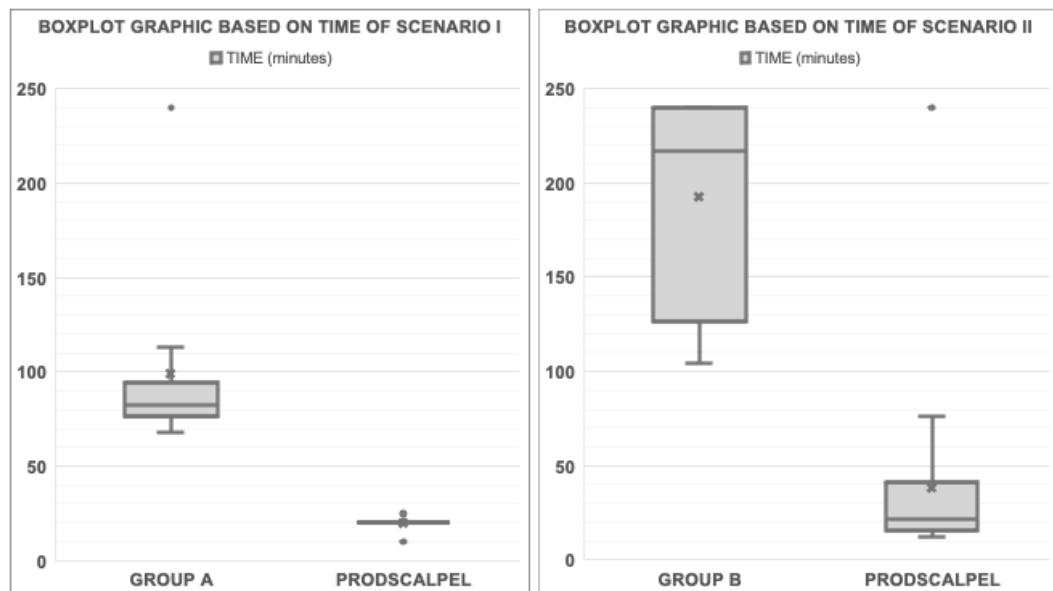


Figure 8.2: Time results grouping automated and manual in both scenarios.
Scenario I: NEATVI - Product Base; Scenario II: Mytar - Product base.

To answer **RQ2**, in both scenarios PRODSICALPEL outperformed participants with respect to the average task time. By considering the sum of time spent with both scenarios, the tool accomplished the product line generation process by inserting two new variants in 4.8 times faster than the mean of participants were able to finish the experiment in the timeout and their resulting product line passed in all test cases.

8.3 THREATS TO VALIDITY

The threats to the validity of this study are mainly related to external and internal validity [160]. Conclusion validity issues involve the low power of the statistical analysis. The number of SPL projects can be a threat to the hypotheses not rejected in this experiment. Aiming to address it, the extension of the experiment to incorporate new donors systems is being explored, and improved results will be reported in future work. In addition, we already provide a set of heterogeneous systems, with distinct characteristics in terms of domains, amount of code lines, and numbers of features. It provides a representation on how the PRODSICALPEL approach behave when applied to different objects as an archive manager providing features to generate a text editors product line.

External Validity. The relatively small number and diversity of systems used for generating a product line pose an external threat to validity. We applied our results to small programs due to the boundaries of an in-lab study; our results may not generalize to larger programs in the wild. We tried to mitigate it by constructing possible real-world scenarios, i.e., reuse of features from unrelated codebase and variations of the same systems, both real-world systems. Additionally, given that our approach was helpful even in small programs, we argue that is likely helpful for larger systems as it is nearly impossible to incorporate new variants to a product line without a large understanding of the donor systems specifications or without specialized tool support [7].

Internal Validity. Due to time expensive nature of a human study, we had few participants. We tried to mitigate this issue by selecting participants with considerable experience in SPL projects. The other threat to the validity is the system size; small features are used in this experiment. We assume that inspecting the code to transfer a feature to a product line is hard, slow, and possibly a tedious work. Thus, we preferred this way to avoid eventual human error as consequence of participant’s tired, in another way, it would require too much time of execution. Even with a limited execution time, we were able to transplant features from donors with significant size (consisting the more than 6k LoC of our donor systems together, as shown in Table A.1. We also use testing as means of validating our approach, which cannot provide a formal proof of its correctness. We used extensive testing to mitigate this risk. Moreover, testing is a standard approach to code evaluation in real-world scenarios due to its high scalability.

8.4 CHAPTER SUMMARY

This chapter presents the results and discussions of an experiment comparing the performance of with that of SPL experts in transplanting features across aN SPL. The

experiment involved two scenarios, and the results were based on the time spent, the number of passing tests, and the product line status. The results showed that outperformed the participants in both scenarios, with only one timeout recorded in 20 trials. In addition, completed the task of transplanting a target feature in all 20 trials, with an average time of 20 minutes, while the participants took an average of 1h24 minutes. The results suggest that is a faster and more reliable option for feature transplantation in SPLs.

Next chapter concludes this thesis, summarizing the main contributions and providing directions to new research.

PART V

CONCLUSIONS

CONCLUDING REMARKS AND FUTURE WORK

In this thesis, we are pursuing a twofold goal. While we extended the ST principles to product variants generation, we also intend to evaluate the flexibility of our transplantation approach as an alternative approach to the optimization of the reengineering process to obtain SPL from existing systems. Thus, we believe we accomplished our goals since we presented the first ST approach and tool for product line and product variants generation (Chapter 4).

We evaluated the potential of FOUNDRY and PRODSICALPEL for automating existing reengineering practices for extracting an SPL from existing systems regarding remaining open issues in this research field (Chapter 6). Then, we validated our approach and tool on two case studies using open-source systems (Chapter 7). We generated two products through the transplantation of features extracted from three real-world systems into two different product bases. Finally, we performed an empirical experiment comparing the performance of our tool in comparison with SPL experts (Chapter 8). The tool accomplished the product line generation process by migrating two features 4.8 times faster than the mean time spent by participants who were able to finish the experiment within the timeout.

Our evaluation studies provide initial evidence to support the claim SPLE using ST, is a feasible and, indeed, promising direction for SPL research and practice. However, more studies are needed to provide more evidence for the generalisability of our approach and to investigate its applicability in an industrial context.

9.1 RESEARCH CONTRIBUTION

The main contributions of this thesis can be summarized as follow:

- **Comparative study on automated SPL reengineering practices via ST.** The comparative study presented in the Chapter 6 aimed at understanding how multi-organ transplantation (as realized in FOUNDRY) automate existing reengineering practices to obtain an SPL. Moreover, how it can be used to address the open issues in the field of reengineering of existing systems into SPL. The result of

this exploratory study gives evidence that ST can be a feasible technique for the reengineering of existing systems to an SPL. Different from existing tools, PROD-SCALPEL can extract, transform and merge features from different systems into a software product with a minimal manual effort of feature identification.

- **An approach for automated SPL reengineering via ST.** Drawing upon the principles of ST, we have devised an approach that encompasses the essential prerequisites for automating the reengineering of systems into **SPLs!**. These prerequisites encompass various facets, including the capacity to extract multiple organs and seamlessly integrate them into a unified product base. Furthermore, our approach harnesses over-organs as valuable SPL assets; once extracted, these over-organs can be refined and tailored to suit diverse host environments. Additionally, our approach demonstrates the ability to identify organ duplication and potential conflicts through the application of clone-ware GI techniques.
- **Realization of the approach in a tool.** The proposed approach depicts the high-level aspects to be considered when automating SPL reengineering via ST assignment. Thus, this work also described how the approach has been implemented, detailing the adopted automation strategy, including all modules implemented to compose our solution.
- **Validation of the approach and tool.** We also conducted two case studies to evaluate the proposed approach where two products were generated from the transplant of features extracted from four real-world systems. Although initial, our results are encouraging. They provide initial evidence to support that automated product line and product variants generation, using transplantation idea, is promising in the direction of product development with no human involvement whatsoever for reinventing functionality that already exists on some other system. Additionally, the proposed approach was validated in an experiment. We could verify its efficiency and efficacy, which were all satisfactory and promising.

9.2 FUTURE WORK

Overall, there is significant potential for further research and development in the area of ST for SPLE. By refining and optimizing the transplantation process, researchers can help to make it a more practical and widely adopted approach for meeting the demands of modern software engineering.

Due to the time constraints to prepare a Ph.D. thesis, this work can be seen as initial research towards the efficient application of ST in the SPLE field. Thus, the following issues should be investigated in future work:

- **Further refinement and optimization of the ST process for SPLE.** In Chapter 7, which presents the case studies, we identified that the process of identifying and extracting semantically required code from multiple donor systems can be a challenging task, particularly if the donor codebase is large and it was implemented

with different architectures, dependencies, and implementation details. This can involve analyzing large amounts of code and identifying all dependencies and interactions across the features in their codebases. Developing more efficient algorithms and methods for call graph generation and identifying and adapting organs can help to streamline this process and make it more effective. This could include research into techniques for automating the extraction and adaptation of code from donor systems, as well as new approaches for efficiently merging and integrating the transplanted features into the target product line. For example, we intend to investigate the use of machine learning or natural language processing techniques to improve the accuracy and efficiency of feature identification and adaptation. They could also explore the use of dynamic analyse techniques and other automated methods to streamline the process of identifying dependencies and interactions among features.

- **The investigation of the use of machine learning techniques for feature identification and adaptation.** Investigation of the use of machine learning techniques for feature identification and adaptation, as well as for predicting potential conflicts and interactions among features. It could involve exploring the use of algorithms such as clustering or classification to automatically identify relevant features in the codebase. Machine learning could also be used to optimize the adaptation of features to the target environment, potentially reducing the need for manual intervention and increasing the efficiency of the ST process.

Furthermore, machine learning could be used to predict potential conflicts and interactions among features during the ST process. For example, a machine learning model could be trained on historical data to identify common conflicts that occur when certain features are combined, or to identify patterns in feature interactions that could lead to issues in the final product.

- **Development of automated testing and validation techniques for ensuring that transplanted features are fully functional and perform as intended.** One of the key challenges in ST is ensuring that the transplanted features are fully functional and perform as intended in the new product. This requires rigorous testing and validation of the features to ensure that they meet the desired requirements and do not introduce any new bugs or errors. Automated testing and validation techniques can greatly aid in this process by providing a systematic and efficient way to test the functionality of the transplanted features. This involves developing automated test suites that cover various aspects of the feature's functionality, such as input/output testing, boundary testing, and stress testing.

Validation techniques can also be developed to ensure that the transplanted features meet the desired quality standards, such as reliability, maintainability, and usability. This can involve developing automated code analysis tools that detect potential issues with the code, as well as developing techniques for measuring the performance and usability of the feature in real-world scenarios.

- **Investigation of the potential use of ST in other areas of software engineering.** There are many potential applications of ST beyond product line reengineering that could be explored in future research. For instance, it could be explored as a technique for modernizing legacy code or for maintaining complex software systems.

In the context of legacy code modernization, ST could be used to extract and transplant specific features from outdated codebases into a new codebase. This could help to reduce the amount of time and effort required for modernization, as it would allow developers to focus on extracting and adapting only the necessary features instead of having to rewrite the entire codebase from scratch.

In terms of software maintenance, ST could potentially be used to transplant features from a well-maintained codebase into a poorly maintained one, thereby improving the overall quality and maintainability of the latter. It could also be used to transplant features from a codebase that is no longer actively developed into one that is still being actively maintained, helping to extend the useful life of the former.

- **Exploration of the use of ST in combination with other software engineering techniques.** In Chapter 6, where we presented a comparative study of ST for SPL reengineering with the existing practice in the literature, we observed a new interesting future work that would apply good practices used in the existing approaches to improve FOUNDRY. By combining the good practices used in such techniques, it may be possible to leverage the strengths of each approach and address their weaknesses, leading to a more robust and comprehensive solution. Additionally, combining ST with other techniques may open up new possibilities and enable the handling of more complex scenarios, such as those involving large-scale software systems with multiple interdependencies.

ST can be combined with other software engineering techniques, such as model-driven engineering (MDE), to enhance the efficiency and effectiveness of SPLE. MDE aims to increase productivity and quality by raising the level of abstraction used in software development, making it easier to reason about complex systems.

One possible approach would be to use MDE to create higher-level models of the donor systems and the desired product line. These models could be used to identify and extract the relevant features more accurately, as well as to detect and resolve potential conflicts among features. The resulting model could then be used as a blueprint for the ST process.

Furthermore, the use of MDE could also improve the testing and validation process by automatically generating test cases from the model, ensuring that the transplanted features behave as intended. This approach can reduce the manual effort required in testing and validation.

Therefore, investigating the combination of ST with MDE can provide new insights into SPLE and can lead to the development of more efficient and effective techniques for software engineering.

- **Evaluation of the scalability and performance of ST in real-world scenarios involving large-scale software systems and product lines.** The results of our evaluations, presented in Chapters 7 and 8, showed that the proposed approach is able to transplant generate product lines via ST techniques. However, we did not evaluate the quality of it in the industry environment.

Evaluating the scalability and performance of ST in real-world scenarios would involve testing the proposed approach on large-scale software systems and product lines to identify any potential scalability limitations or performance issues. This could be done through empirical studies using benchmarks or case studies, where the proposed approach is applied to real-world software systems from software development industry and the resulting performance and scalability metrics are compared to those of existing reengineering approaches.

The evaluation could also involve analyzing the impact of the size and complexity of the software system and the number of features being transplanted on the performance and scalability of the ST approach. Additionally, the evaluation could include testing the approach under different conditions, such as varying levels of feature dependencies and interactions, to assess its effectiveness and efficiency.

- **Investigation of the potential impact of ST on software quality, maintainability, and reliability.** Although our evaluations prove initial evidence of the potential of ST to improve the SPLE field, investigating the potential impact of ST on software quality, maintainability, and reliability is crucial because it can help to understand the benefits and limitations of the approach. Additionally, this investigation can also provide insights into how ST can be further refined and optimized to improve its effectiveness in improving software quality, maintainability, and reliability.

Some potential research directions in this area could include: (i) Developing metrics to measure the quality, maintainability, and reliability of software systems before and after undergoing ST; (ii) Conducting experiments and case studies to evaluate the impact of ST on software quality, maintainability, and reliability, and comparing these results with those of traditional software engineering techniques; (iii) Investigating the potential trade-offs between ST and other software engineering techniques concerning software quality, maintainability, and reliability; (iv) Exploring the use of ST in specific domains or industries, such as safety-critical systems or healthcare; and (v) evaluating the impact on software quality, maintainability, and reliability in these contexts.

9.3 CONCLUDING REMARKS

The automated transplantation aims to identify and extract an organ (interesting behaviour to transplant), all code associated with the feature of interest, and then transform it to be compatible with the namespace and context of its target site in the host. The automatic transplantation of a single organ already faces significant challenges. The code

from different systems is unlikely to compile, execute and pass test cases when relocated into an unrelated foreign system without extensive modification.

The utilization of ST as a means to successfully automate the re-engineering of product lines presents even greater challenges. The multiple features extraction involves the identification of all semantically relevant organ's code from multiple codebases to maintain the organ functional even outside the donor environment. Adapting the extracted features to work in a single and strange host system can also be challenging. The host system may have different architectures or dependencies than the donor systems, which can require significant modifications to the extracted organs. The successful implantation of the extracted organ into an emergent new product is hindered by potential dependencies between it and other transplanted organs.

Face to the additional challenges, we propose an approach and a tool that can be used for optimizing the process of generating both product lines as new products from existing codebases with minimal human involvement.

Both approach and the tool have been validated through case studies with real-world systems and an experiment with SPL experts to compare our approach with manual effort. We showed the applicability of FOUNDRY and significant time effort improvements when using PRODSALPEL.

We argue that the migration to SPL transplantation-based in contrast to the existing SPL practices makes it easier to use in practice. Our approach improves SPL maintainability by physically separating features from the product base. FOUNDRY can be used both for *extractive* or *reactive* product line migration, as well as as a systematic Clone&own strategy to specialize existing products. FOUNDRY avoids code duplication of feature implementations while preserving feature behaviour within a product line of which it is part. It can automatically propagate feature changes. That is, it provides solutions for problems often cited in the reengineering of systems in SPL literature.

Our evaluation studies provide initial evidence to support the claim SPLE using ST, is a feasible and, indeed, promising direction for SPL research and practice. Thus, our work leaves a valuable trail for further research. However, more studies are needed to provide more evidence for the generalisability of our approach and to investigate its applicability in an industrial context.

BIBLIOGRAPHY

- 1 POHL, K.; BÖCKLE, G.; LINDEN, F. J. v. d. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.
- 2 KRUEGER, C. W. Easing the transition to software mass customization. In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK, UK: Springer-Verlag, 2002. (PFE '01), p. 282–293.
- 3 ASSUNÇÃO, W. K. G.; LOPEZ-HERREJON, R. E.; LINSBAUER, L.; VERGILIO, S. R.; EGYED, A. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, v. 22, p. 2972–3016, 2017.
- 4 De Souza, L. O.; O'LEARY, P.; De Almeida, E. S.; De Lemos Meira, S. R. Product derivation in practice. *Information and Software Technology*, Elsevier B.V., v. 58, p. 319–337, 2015.
- 5 LINDEN, F. J. v. d.; SCHMID, K.; ROMMES, E. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.
- 6 BERGER, T.; RUBLACK, R.; NAIR, D.; ATLEE, J. M.; BECKER, M.; CZARNECKI, K.; KaSOWSKI, A. A survey of variability modeling in industrial practice. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. New York, NY, USA: ACM, 2013. (VaMoS '13), p. 7:1–7:8. ISBN 978-1-4503-1541-8. Disponível em: <http://doi.acm.org/10.1145/2430502.2430513>.
- 7 BREIVOLD, H.; LARSSON, S.; LAND, R. Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies. *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, 2008.
- 8 NORTHROP, L. M.; C., C. P. A Framework for Software Product Line Practice version 5.0. technical report. *Software Engineering Institute*, p. 258, 2012. Disponível em: <http://www.sei.cmu.edu/productlines/frame-“”report/index.h>.
- 9 Bockle, G.; Clements, P.; McGregor, J. D.; Muthig, D.; Schmid, K. Calculating roi for software product lines. *IEEE Software*, v. 21, n. 3, p. 23–31, 2004.
- 10 Fischer, S.; Linsbauer, L.; Lopez-Herrejon, R. E.; Egyed, A. The ecco tool: Extraction and composition for clone-and-own. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.: s.n.], 2015. v. 2, p. 665–668.

- 11 HARMAN, M.; JIA, Y.; KRINKE, J.; LANGDON, W. B.; PETKE, J.; ZHANG, Y. Search based software engineering for software product line engineering: A survey and directions for future work. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. New York, NY, USA: [s.n.], 2014. (SPLC '14), p. 5–18.
- 12 LOPEZ-HERREJON, R. E.; LINSBAUER, L.; GALINDO, J. A.; PAREJO, J. A.; BENAVIDES, D.; SEGURA, S.; EGYED, A. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, v. 103, p. 353–369, 2015.
- 13 LOPEZ-HERREJON, R. E.; LINSBAUER, L.; EGYED, A. A systematic mapping study of search-based software engineering for software product lines. *Information and Software Technology*, v. 61, p. 33 – 51, 2015.
- 14 HARMAN, M.; LANGDON, W. B.; WEIMER, W. Genetic Programming for Reverse Engineering. p. 1–10, 2013.
- 15 PETKE, J.; HARMAN, M.; LANGDON, W. B.; WEIMER, W. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In: NICOLAU, M.; KRAWIEC, K.; HEYWOOD, M. I.; CASTELLI, M.; GARCÍA-SÁNCHEZ, P.; MERELO, J. J.; SANTOS, V. M. R.; SIM, K. (Ed.). *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 137–149.
- 16 Petke, J.; Harman, M.; Langdon, W. B.; Weimer, W. Specialising software for different downstream applications using genetic improvement and code transplantation. *IEEE Transactions on Software Engineering*, v. 44, n. 6, p. 574–594, June 2018.
- 17 PETKE, J.; HARALDSSON, S. O.; HARMAN, M.; LANGDON, W. B.; WHITE, D. R.; WOODWARD, J. R. Genetic improvement of software: A comprehensive survey. *IEEE Trans. Evol. Comput.*, v. 22, n. 3, p. 415–432, 2018.
- 18 BARR, E. T.; HARMAN, M.; JIA, Y.; MARGINEAN, A.; PETKE, J. Automated software transplantation. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2015. (ISSTA 2015), p. 257–269.
- 19 SIDIROGLOU-DOUSKOS, S.; LAHTINEN, E.; EDEN, A.; LONG, F.; RINARD, M. Codecarboncopy. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: [s.n.], 2017. (ESEC/FSE 2017), p. 95–105.
- 20 CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001.
- 21 BINKLEY, D.; GOLD, N.; HARMAN, M.; ISLAM, S.; KRINKE, J.; YOO, S. Orbs: Language-independent program slicing. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2014. (FSE 2014), p. 109–120.

- 22 HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. *Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications*. 2009.
- 23 RAHMAN, M. T.; QUEREL, L.-P.; RIGBY, P. C.; ADAMS, B. Feature toggles: Practitioner practices and a case study. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2016. (MSR '16), p. 201–211.
- 24 KÄSTNER, C.; APEL, S.; KUHLEMANN, M. Granularity in software product lines. In: . New York, NY, USA: Association for Computing Machinery, 2008. (ICSE '08), p. 311–320.
- 25 BAYER, J.; GIRARD, J.-F.; WÜRTHNER, M.; DEBAUD, J.-M.; APEL, M. Transitioning legacy assets to a product line architecture. In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 1999. (ESEC/FSE-7), p. 446–463.
- 26 ALMEIDA, E. S. de. Software reuse and product line engineering. In: _____. *Handbook of Software Engineering*. Cham: Springer International Publishing, 2019. p. 321–348.
- 27 FAVRE, L. M. MDA-Based Object-Oriented Reverse Engineering. *Model Driven Architecture for Reverse Engineering Technologies*, p. 199–229, 2011.
- 28 MARTINEZ, J.; ZIADI, T.; BISSYANDÉ, T. F.; KLEIN, J.; TRAON, Y. L. Bottom-up adoption of software product lines: A generic and extensible approach. In: *Proceedings of the 19th International Conference on Software Product Line*. New York, NY, USA: Association for Computing Machinery, 2015. (SPLC '15), p. 101–110.
- 29 ROMERO, D.; URLI, S.; QUINTON, C.; BLAY-FORNARINO, M.; COLLET, P.; DUCHIEN, L.; MOSSER, S. Splemma: a generic framework for controlled-evolution of software product lines. In: *SPLC '13 Workshops*. [S.l.: s.n.], 2013.
- 30 MOHAMED, F.; ABU-MATAR, M.; MIZOUNI, R.; AL-QUTAYRI, M.; MAHMOUD, Z. A. SaaS dynamic evolution based on model-driven software product lines. *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, p. 292–299, 2014.
- 31 INOZEMTSEVA, L.; HOLMES, R. Coverage is not strongly correlated with test suite effectiveness. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 435–445.
- 32 GOPINATH, R.; JENSEN, C.; GROCE, A. Code coverage for suite evaluation by developers. In: . New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 72–82.

- 33 KOCHHAR, P. S.; THUNG, F.; LO, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In: . [S.l.: s.n.], 2015.
- 34 JANOTA, M.; KINIRY, J.; BOTTERWECK, G. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. *Lero Technical Report*, n. March, 2008.
- 35 VALENTE, M. T.; BORGES, V.; PASSOS, L. A semi-automatic approach for extracting software product lines. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 38, p. 737–754, 2011. ISSN 0098-5589.
- 36 KRUEGER, C. W. Easing the transition to software mass customization. In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK, UK: Springer-Verlag, 2002. (PFE '01), p. 282–293.
- 37 PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. Swindon, UK: BCS Learning & Development Ltd., 2008. (EASE'08), p. 68–77.
- 38 KUCZA, T.; NÄTTINEN, M.; PARVIAINEN, P. Improving knowledge management in software reuse process. In: *Proceedings of the Third International Conference on Product Focused Software Process Improvement*. [S.l.: s.n.], 2001. (PROFES '01), p. 141–152.
- 39 JALENDER, B.; GOVARDHAN, A.; PREMCHAND, P. A pragmatic approach to software reuse. *Journal of Theoretical and Applied Information Technology*, v. 14, n. 2, p. 87–96, 2010.
- 40 MCGREGOR, J. D. Initiating software product lines. *IEEE Software*, p. 24–27, 2002.
- 41 SAMETINGER, J. *Software Engineering with Reusable Components*. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN 3-540-62695-6.
- 42 SOORA, S. K. A Framework for Software Reuse and Research Challenges. v. 4, n. 10, p. 441–448, 2014.
- 43 FRAKES, W. B.; KANG, K. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 31, n. 7, p. 529–536, jul. 2005.
- 44 ALVES, V.; MATOS, P.; COLE, L.; VASCONCELOS, A.; BORBA, P.; RAMALHO, G. Extracting and evolving code in product lines with aspect-oriented programming. In: _____. *Transactions on Aspect-Oriented Software Development IV*. Berlin, Heidelberg: Springer-Verlag, 2007. p. 117–142.
- 45 Chikofsky, E. J.; Cross, J. H. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, v. 7, n. 1, p. 13–17, 1990.
- 46 AN integrated environment for reuse reengineering C code. *Journal of Systems and Software*, v. 42, n. 2, p. 153 – 164, 1998.

- 47 LOZANO, A. An overview of techniques for detecting software variability concepts in source code. In: TROYER, O. D.; MEDEIROS, C. B.; BILLEN, R.; HALLOT, P.; SIMITSIS, A.; MINGROOT, H. V. (Ed.). *Advances in Conceptual Modeling. Recent Developments and New Directions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 141–150.
- 48 LAGUNA, M. A.; CRESPO, Y. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, v. 78, n. 8, p. 1010 – 1034, 2013.
- 49 FENSKE, W.; THÜM, T.; SAAKE, G. A taxonomy of software product line reengineering. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: Association for Computing Machinery, 2014. (VaMoS '14).
- 50 ANWIKAR, V.; NAIK, R.; CONTRACTOR, A.; MAKKAPATI, H. Domain-driven technique for functionality identification in source code. *SIGSOFT Softw. Eng. Notes*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 3, p. 1–8, maio 2012.
- 51 KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. [S.l.], 1990.
- 52 TARTLER, R.; LOHMANN, D.; SINCERO, J.; SCHRÖDER-PREIKSCHAT, W. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In: *Proceedings of the Sixth Conference on Computer Systems*. New York, NY, USA: ACM, 2011. (EuroSys '11), p. 47–60.
- 53 WAGNER, C. Model-driven software migration: A methodology - reengineering, recovery and modernization of legacy systems. In: . [S.l.: s.n.], 2014.
- 54 Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsborrow, L. A.; Ward, N. J.; Marsh, D. W. R. Industrial perspective on static analysis. *Software Engineering Journal*, v. 10, n. 2, p. 69–75, 1995.
- 55 Cornelissen, B.; Zaidman, A.; van Deursen, A.; Moonen, L.; Koschke, R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, v. 35, n. 5, p. 684–702, 2009.
- 56 MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. *Introduction to Information Retrieval*. USA: Cambridge University Press, 2008.
- 57 RUBIN, J.; CHECHIK, M. A survey of feature location techniques. In: *Domain Engineering, Product Lines, Languages, and Conceptual Models*. [S.l.: s.n.], 2013.
- 58 Eisenbarth, T.; Koschke, R.; Simon, D. Derivation of feature component maps by means of concept analysis. In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2001. p. 176–179.

- 59 Frenzel, P.; Koschke, R.; Breu, A. P. J.; Angstmann, K. Extending the reflexion method for consolidating software variants into product lines. In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. [S.l.: s.n.], 2007. p. 160–169.
- 60 OLSZAK, A.; JØRGENSEN, B. N. Remodularizing java programs for improved locality of feature implementations in source code. *Sci. Comput. Program.*, v. 77, p. 131–151, 2012.
- 61 KNODEL, J.; FORSTER, T.; GIRARD, J.-F. Comparing design alternatives from field-tested systems to support product line architecture design. In: *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. USA: IEEE Computer Society, 2005. (CSMR '05), p. 344–353. Disponível em: <https://doi.org/10.1109/CSMR.2005.18>.
- 62 RUBIN, J.; CHECHIK, M. From products to product lines using model matching and refactoring. In: *SPLC Workshops*. [S.l.: s.n.], 2010.
- 63 RUBIN, J.; CHECHIK, M. Combining related products into product lines. In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2012. (FASE'12), p. 285–300.
- 64 MAÂZOUN, J.; BOUASSIDA, N.; BEN-ABDALLAH, H. Feature model recovery from product variants based on a cloning technique. In: *SEKE*. [S.l.: s.n.], 2014.
- 65 DUBINSKY, Y.; RUBIN, J.; BERGER, T.; DUSZYNSKI, S.; BECKER, M.; CZARNECKI, K. An exploratory study of cloning in industrial software product lines. In: *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*. USA: IEEE Computer Society, 2013. (CSMR '13), p. 25–34. ISBN 9780769549484. Disponível em: <https://doi.org/10.1109/CSMR.2013.13>.
- 66 KIM, M.; BERGMAN, L.; LAU, T.; NOTKIN, D. An ethnographic study of copy and paste programming practices in oopl. In: *Proceedings of the 2004 International Symposium on Empirical Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. (ISESE '04), p. 83–92.
- 67 HOLMES, R.; WALKER, R. J.; MURPHY, G. C. Strathcona example recommendation tool. *SIGSOFT Softw. Eng. Notes*, Association for Computing Machinery, New York, NY, USA, v. 30, n. 5, p. 237–240, set. 2005.
- 68 JABLONSKI, P.; HOU, D. Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In: *In Proceedings of Eclipse Technology Exchange Workshop at OOPSLA 2007(ETX'07), 5pp*. [S.l.: s.n.], 2007.
- 69 HARMAN, M.; JIA, Y.; LANGDON, W. B. Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 8636 LNCS, p. 247–252, 2014.

- 70 HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012.
- 71 HARMAN, M.; MCMINN, P.; SOUZA, J. T. de; YOO, S. Empirical software engineering and verification. In: MEYER, B.; NORDIO, M. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2012. cap. Search Based Software Engineering: Techniques, Taxonomy, Tutorial, p. 1–59.
- 72 HARMAN, M. Search based software engineering. In: ALEXANDROV, V. N.; ALBADA, G. D. van; SLOOT, P. M. A.; DONGARRA, J. (Ed.). *Computational Science – ICCS 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 740–747.
- 73 CIMITILE, A.; FASOLINO, A. R.; MARESCA, P. Reuse reengineering and validation via concept assignment. In: *Proceedings of the Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1993. (ICSM '93), p. 216–225.
- 74 DEURSEN, A. van; KUIPERS, T. Identifying objects using cluster and concept analysis. In: *Proceedings of the 21st International Conference on Software Engineering*. New York, NY, USA: ACM, 1999. (ICSE '99), p. 246–255.
- 75 HARMAN, M.; GOLD, N.; HIERONS, R.; BINKLEY, D. Code extraction algorithms which unify slicing and concept assignment. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings*. [S.l.: s.n.], 2002. p. 11–20.
- 76 KOMONDOOR, R.; HORWITZ, S. Semantics-preserving procedure extraction. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2000. (POPL '00), p. 155–169. ISBN 1-58113-125-9.
- 77 LANUBILE, F.; VISAGGIO, G. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 23, n. 4, p. 246–259, abr. 1997.
- 78 BECK, J.; EICHMANN, D. Program and interface slicing for reverse engineering. In: *Proceedings of the 15th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993. (ICSE '93), p. 509–518.
- 79 HALL, R. J. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, Springer, v. 2, n. 1, p. 33–53, 1995.
- 80 HARROLD, M. J.; CI, N. Reuse-driven interprocedural slicing. In: *Proceedings of the 20th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 1998. (ICSE '98), p. 74–83.

- 81 HORWITZ, S.; REPS, T.; BINKLEY, D. Interprocedural slicing using dependence graphs. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1988. (PLDI '88), p. 35–46.
- 82 BOOCH, G.; KOZACZYNSKI, W. Guest editors' introduction: Component-based software engineering. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 15, n. 05, p. 34–36, sep 1998.
- 83 WANG, S.; MAO, X.; YU, Y. An initial step towards organ transplantation based on github repository. *IEEE Access*, v. 6, p. 59268–59281, 2018.
- 84 Eisenbarth, T.; Koschke, R.; Simon, D. Locating features in source code. *IEEE Transactions on Software Engineering*, v. 29, n. 3, p. 210–224, 2003.
- 85 WEISER, M. D. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. Tese (Doutorado), USA, 1979.
- 86 HAJNAL, A.; FORGÁCS, I. Understanding program slices. *Acta Cybern.*, Institute of Informatics, University of Szeged, Szeged, HUN, v. 20, n. 4, p. 483–497, dez. 2012.
- 87 De Lucia, A. Program slicing: methods and applications. In: *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2001. p. 142–149.
- 88 HARMAN, M.; ; HARMAN, M.; HIERONS, R. M. *An Overview of Program Slicing*. 2001.
- 89 JIANG, T.; GOLD, N.; HARMAN, M.; LI, Z. Locating dependence structures using search-based slicing. *Information and Software Technology*, v. 50, n. 12, p. 1189 – 1209, 2008.
- 90 BINKLEY, D.; GOLD, N.; HARMAN, M.; ISLAM, S.; KRINKE, J.; YOO, S. Orbs: Language-independent program slicing. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 109–120.
- 91 FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 9, n. 3, p. 319–349, jul. 1987.
- 92 ROZENBERG, G.; BCK, T.; KOK, J. N. *Handbook of Natural Computing*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2011. ISBN 3540929096.
- 93 HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136.

- 94 Langdon, W. B.; Harman, M. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, v. 19, n. 1, p. 118–135, 2015.
- 95 LANGDON, W. B.; POLI, R.; MCPHEE, N. F.; KOZA, J. R. Genetic programming: An introduction and tutorial, with a survey of techniques and applications. In: _____. *Computational Intelligence: A Compendium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 927–1028.
- 96 KOREL, B.; LASKI, J. Dynamic program slicing. *Information Processing Letters*, v. 29, n. 3, p. 155 – 163, 1988.
- 97 Krinke, J. Barrier slicing and chopping. In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2003. p. 81–87.
- 98 PERRY, D. E.; PORTER, A. A.; VOTTA, L. G. Empirical studies of software engineering: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 345–355.
- 99 KRÜGER, J.; KRIETER, S.; SAAKE, G.; LEICH, T. Extracting product lines from variants (explant). In: . New York, NY, USA: Association for Computing Machinery, 2020. (VAMOS '20).
- 100 HLAD, N.; ABDELHAK-DJAMEL, S.; CHRISTOPHE, D. *IsiSPL: Toward an Automated Reactive Approach to build Software Product Lines*. 2021.
- 101 YOSHIMURA, K.; GANESAN, D.; MUTHIG, D. Defining a strategy to introduce a software product line using existing embedded systems. In: *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. New York, NY, USA: Association for Computing Machinery, 2006. (EMSOFT '06), p. 63–72.
- 102 Kästner, C.; Dreiling, A.; Ostermann, K. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, v. 40, n. 1, p. 67–82, 2014.
- 103 LIEBIG, J.; APEL, S.; LENGAUER, C.; KÄSTNER, C.; SCHULZE, M. An analysis of the variability in forty preprocessor-based software product lines. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 105–114.
- 104 KRÜGER, J.; BERGER, T.; LEICH, T.; FEATURES, . Features and How to Find Them: A Survey of Manual Feature Location Feature location; systematic literature review; reverse variability engineering; feature identification; feature mapping. *Software Engineering for Variability Intensive Systems: Foundations and Applications*, 2018.
- 105 MEINICKE, J.; WONG, C.-P.; VASILESCU, B.; KÄSTNER, C. Exploring differences and commonalities between feature flags and configuration options. In: . New York, NY, USA: Association for Computing Machinery, 2020. (ICSE-SEIP '20), p. 233–242.

- 106 APEL, S.; BATORY, D.; KSTNER, C.; SAAKE, G. *Feature-Oriented Software Product Lines: Concepts and Implementation*. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 3642375200, 9783642375200.
- 107 KÄSTNER, C.; TRUJILLO, S.; APEL, S. Visualizing software product line variabilities in source code. In: *In Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*. [S.l.: s.n.], 2008.
- 108 JÉZÉQUEL, J.-M.; KIENZLE, J.; ACHER, M. From feature models to feature toggles in practice. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*. New York, NY, USA: Association for Computing Machinery, 2022. (SPLC '22), p. 234–244.
- 109 BACHMANN, F.; CLEMENTS, P. *Variability in Software Product Lines*. Pittsburgh, PA, 2005.
- 110 LOTUFO, R.; SHE, S.; BERGER, T.; CZARNECKI, K.; WASOWSKI, A. Evolution of the linux kernel variability model. In: . Berlin, Heidelberg: Springer-Verlag, 2010. (SPLC'10), p. 136–150. ISBN 3642155782.
- 111 XU, T.; JIN, L.; FAN, X.; ZHOU, Y.; PASUPATHY, S.; TALWADKER, R. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 307–319.
- 112 BESSEY, A.; BLOCK, K.; CHELF, B.; CHOU, A.; FULTON, B.; HALLEM, S.; HENRI-GROS, C.; KAMSKY, A.; MCPEAK, S.; ENGLER, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 53, n. 2, p. 66–75, feb 2010.
- 113 ILYAS, B.; ELKHALIFA, I. Static code analysis: A systematic literature review and an industrial survey. In: . [S.l.: s.n.], 2016.
- 114 LIU, L.; MAO, X. A Study on Code Transplantation Technique based on Program Slicing. v. 161, n. Tlicsc, p. 294–298, 2018.
- 115 BASTOS, J. F.; NETO, P. A. da M. S.; OLEARY, P.; ALMEIDA, E. S. de; MEIRA, S. R. de L. Software product lines adoption in small organizations. *J. Syst. Softw.*, Elsevier Science Inc., USA, v. 131, n. C, p. 112–128, set. 2017.
- 116 CHATTERJEE, S.; JUVEKAR, S.; SEN, K. Sniff: A search engine for java using free-form queries. In: CHECHIK, M.; WIRSING, M. (Ed.). *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 385–400.
- 117 Savage, T.; Revelle, M.; Poshyvanyk, D. Flat3: feature location and textual tracing tool. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. [S.l.: s.n.], 2010. v. 2, p. 255–258.

- 118 McMillan, C.; Grechanik, M.; Poshyvanyk, D.; Xie, Q.; Fu, C. Portfolio: finding relevant functions and their usage. In: *2011 33rd International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2011. p. 111–120.
- 119 RABKIN, A.; KATZ, R. Static extraction of program configuration options. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2011. (ICSE '11), p. 131–140.
- 120 HEESCH, D. van. *Doxygen: Source code documentation generator tool*. 2018. Disponível em: <http://www.stack.nl/dimitri/doxygen/>.
- 121 WANG, S.; MAO, X.; YU, Y. An initial step towards organ transplantation based on GitHub repository. *IEEE Access*, Institute of Electrical and Electronics Engineers (IEEE), v. 6, p. 59268–59281, 2018.
- 122 RIBEIRO, M.; QUEIROZ, F.; BORBA, P.; TOLêDO, T.; BRABRAND, C.; SOARES, S. On the impact of feature dependencies when maintaining preprocessor-based software product lines. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 47, n. 3, p. 23–32, out. 2011. ISSN 0362-1340.
- 123 CAFEO, B. B.; CIRILO, E.; GARCIA, A.; DANTAS, F.; LEE, J. Feature dependencies as change propagators: An exploratory study of software product lines. *Information and Software Technology*, Elsevier Ltd., v. 69, p. 37–49, 2016.
- 124 ROY, C. K. *Detection and Analysis of Near-miss Software Clones*. Tese (Doutorado), Kingston, Ont., Canada, Canada, 2009. AAINR65337.
- 125 KERNIGHAN, B. W.; PIKE, R. *The UNIX Programming Environment*. [S.l.]: Prentice Hall Professional Technical Reference, 1983.
- 126 CORDY, J. R. The txl source transformation language. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., v. 61, n. 3, p. 190–210, 2006.
- 127 GACEK, C.; ANASTASOPOULES, M. Implementing product line variabilities. In: . New York, NY, USA: Association for Computing Machinery, 2001. (SSR '01, 3), p. 109–117.
- 128 KNUDSEN, J. L. Name collision in multiple classification hierarchies. In: *Proceedings of the European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 1988. (ECOOP '88), p. 93–109. ISBN 3540500537.
- 129 FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.
- 130 KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LONGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: *ECOOP'97 — Object-Oriented Programming*. [S.l.: s.n.], 1997. p. 220–242.

- 131 TRIFU, M. Tool-supported identification of functional concerns in object-oriented code. In: . [S.l.: s.n.], 2010.
- 132 HEIDENREICH, F.; KOPCSEK, J.; WENDE, C. Featuremapper: mapping features to models. In: *ICSE Companion '08*. [S.l.: s.n.], 2008.
- 133 SEIDL, C.; HEIDENREICH, F.; ASSMANN, U. Co-evolution of models and feature mapping in software product lines. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. New York, NY, USA: Association for Computing Machinery, 2012. (SPLC '12), p. 76–85.
- 134 NUNES, C.; GARCIA, A.; LUCENA, C.; LEE, J. Heuristic expansion of feature mappings in evolving program families. *Softw. Pract. Exper.*, John Wiley Sons, Inc., USA, v. 44, n. 11, p. 1315–1349, nov. 2014.
- 135 ZIADI, T.; FRIAS, L.; SILVA, M. A. A. da; ZIANE, M. Feature identification from the source code of product variants. In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. USA: IEEE Computer Society, 2012. (CSMR '12), p. 417–422.
- 136 BAGHERI, E.; ENSAN, F.; GAEVIĆ, D. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering*, v. 19, p. 335–377, 2011.
- 137 WESTON, N.; CHITCHYAN, R.; RASHID, A. A framework for constructing semantically composable feature models from natural language requirements. In: *Proceedings of the 13th International Software Product Line Conference*. USA: Carnegie Mellon University, 2009. (SPLC '09), p. 211–220.
- 138 NUNES, C.; GARCIA, A.; LUCENA, C.; LEE, J. History-sensitive heuristics for recovery of features in code of evolving program families. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. New York, NY, USA: Association for Computing Machinery, 2012. (SPLC '12), p. 136–145.
- 139 Xue, Y. Reengineering legacy software products into software product line based on automatic variability analysis. In: *2011 33rd International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2011. p. 1114–1117.
- 140 KLATT, B.; KROGMANN, K.; SEIDL, C. Program dependency analysis for consolidating customized product copies. *2014 IEEE International Conference on Software Maintenance and Evolution*, p. 496–500, 2014.
- 141 ARAÚJO, J. a.; aO, M. G.; MOREIRA, A.; aO, I. S.; AMARAL, V.; BANIASSAD, E. Advanced modularity for building spl feature models: A model-driven approach. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2013. (SAC '13), p. 1246–1253.

- 142 MARTINEZ, J.; ZIADI, T.; KLEIN, J.; TRAON, Y. L. Identifying and visualising commonality and variability in model variants. In: *ECMFA*. [S.l.: s.n.], 2014.
- 143 TANG, Y.; LEUNG, H. Top-down feature mining framework for software product line. In: *Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 2*. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2015. (ICEIS 2015), p. 71–81.
- 144 Abbasi, E. K.; Acher, M.; Heymans, P.; Cleve, A. Reverse engineering web configurators. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. [S.l.: s.n.], 2014. p. 264–273.
- 145 Stoermer, C.; O'Brien, L. Map - mining architectures for product line evaluations. In: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. [S.l.: s.n.], 2001. p. 35–44.
- 146 SAMPATH, P. An elementary theory of product-line variations. *Form. Asp. Comput.*, Springer-Verlag, Berlin, Heidelberg, v. 26, n. 4, p. 695–727, jul. 2014.
- 147 PASSOS, L.; CZARNECKI, K.; APEL, S.; WaSOWSKI, A.; KäSTNER, C.; GUO, J. Feature-oriented software evolution. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: Association for Computing Machinery, 2013. (VaMoS '13).
- 148 ZHANG, G.; SHEN, L.; PENG, X.; XING, Z.; ZHAO, W. Incremental and iterative reengineering towards software product line: An industrial case study. In: *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*. USA: IEEE Computer Society, 2011. (ICSM '11), p. 418–427.
- 149 ACHER, M.; CLEVE, A.; COLLET, P.; MERLE, P.; DUCHIEN, L.; LAHIRE, P. Extraction and Evolution of Architectural Variability Models in Plugin-based Systems. *Software and Systems Modeling*, Springer Verlag, p. 27 p., jul. 2013. Disponível em: <https://hal.inria.fr/hal-00859472>.
- 150 BIGGERSTAFF, T. J.; MITBANDER, B. G.; WEBSTER, D. E. Program understanding and the concept assignment problem. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 5, p. 72–82, maio 1994.
- 151 KäSTNER, C.; GIARRUSSO, P. G.; RENDEL, T.; ERDWEG, S.; OSTERMANN, K.; BERGER, T. Variability-aware parsing in the presence of lexical macros and conditional compilation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 46, n. 10, p. 805–824, out. 2011. ISSN 0362-1340.
- 152 LOPEZ-HERREJON, R. E.; MONTALVILLO-MENDIZABAL, L.; EGYED, A. From requirements to features: An exploratory study of feature-oriented refactoring. In: . USA: IEEE Computer Society, 2011. (SPLC '11), p. 181–190.

- 153 KANG, K. C.; KIM, M.; LEE, J.; KIM, B. Feature-oriented re-engineering of legacy systems into product line assets: A case study. In: *Proceedings of the 9th International Conference on Software Product Lines*. Berlin, Heidelberg: Springer-Verlag, 2005. (SPLC'05), p. 45–56.
- 154 NÖBAUER, M.; SEYFF, N.; GROHER, I. Similarity analysis within product line scoping: An evaluation of a semi-automatic approach. In: JARKE, M.; MYLOPOULOS, J.; QUIX, C.; ROLLAND, C.; MANOLOPOULOS, Y.; MOURATIDIS, H.; HORKOFF, J. (Ed.). *Advanced Information Systems Engineering*. Cham: Springer International Publishing, 2014. p. 165–179.
- 155 YIN, R. K. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Fourth edition. [S.l.]: Sage Publications, 2008.
- 156 BRERETON, P.; KITCHENHAM, B. A.; BUDGEN, D.; LI, Z. Using a protocol template for case study planning. In: *International Conference on Evaluation & Assessment in Software Engineering*. [S.l.: s.n.], 2008.
- 157 RUNESON, P.; HÖST, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 14, n. 2, p. 131–164, apr 2009.
- 158 HÖST, M.; RUNESON, P. Checklists for software engineering case study research. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, p. 479–481, 2007.
- 159 HERRIOTT, R. E.; FIRESTONE, W. A. Multisite Qualitative Policy Research: Optimizing Description and Generalizability. *Educational Researcher*, v. 12, n. 2, p. 14–19, 1983.
- 160 WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, Incorporated, 2012.
- 161 WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, Incorporated, 2012.
- 162 BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: *Encyclopedia of Software Engineering*. [S.l.]: Wiley, 1994.
- 163 MAHMOOD, W.; STRÜBER, D.; BERGER, T.; LÄMMEL, R.; MUKELABAI, M. Seamless variability management with the virtual platform. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, p. 1658–1670, 2021.
- 164 SOUZA, L.; BARR, E.; PETKE, J.; ALMEIDA, E.; NETO, P. *The project: software product line engineering via automated software transplantation*. 2023. |<https://autotransplantation-spl.github.io/foundry.github.io/>. Accessed: 2023-04-25.

- 165 SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples)†. *Biometrika*, v. 52, n. 3-4, p. 591–611, 12 1965.
- 166 RAZALI, N. M.; WAH, Y. B. Power comparisons of shapiro-wilk , kolmogorov-smirnov , lilliefors and anderson-darling tests. In: . [S.l.: s.n.], 2011.
- 167 GELMAN, A. Analysis of variance: Why it is more important than ever? *Quality Engineering*, v. 51, p. 295–300, 2005.

MULTIPLE CASE STUDIES: DATA AND SUPPORT MATERIAL

This appendix provides instruction of PRODSICALPEL usage, including all dependencies, commands and directory organization. All the case studies artifacts are available in the project webpage¹. You can find links to the artefacts and a script that runs all our case studies. This artifact contains the tool PRODSICALPEL, donors and host codebases. We also provide our regression, augmented regression, and acceptance test suites, where possible. In the other cases these test suites were executed manually, or the original regression test suite was not executing at all the organ.

Note: the scripts contains just one run of each transplant. In the thesis the results are averaged on 20 runs. By using the random seed parameter of PRODSICALPEL, and than run the script for 20 times, the results from the thesis may be approximated.

A.1 PRODSICALPEL USAGE

This section provides detailed information on how to effectively use and apply the PRODSICALPEL tool. It serves as a practical guide for users who want to utilize PRODSICALPEL for their own projects.

Prerequisites. For correctly compile, PRODSICALPEL requires:

- Xcode Command Line Tools²;
- gcc³;
- autoconf⁴;
- cflow⁵;

¹<https://autotransplantation-spl.github.io/foundry.github.io/>

²<https://developer.apple.com/xcode/>

³<https://gcc.gnu.org/>

⁴<https://www.gnu.org/software/autoconf/>

⁵<https://www.gnu.org/software/cflow/>

- doxygen⁶;
- automake⁷;
- libglib2.0⁸;
- make⁹;
- libgrypt20¹⁰;
- check¹¹;
- *Pkg-config*¹²;
- GNU Diff¹³;
- and with at least 16 GB memory.

For correctly execution, PRODSALPEL requires:

- *Autoconf*;
- *libtool*¹⁴;
- *Pkg-config*;
- *Check*;
- cflow;
- doxygen;
- and GNU Diff

Transplantation directory organization. Table 8.2 illustrate the transplantation directory organization. The transplantation directory is structured to facilitate the execution of the ST process. At the root level, there are several key files and directories. The *prodsalpel.exec* file represents the executable file of the PRODSALPEL tool, while the *ErrorFile.out* contains any errors generated during the *TXL* transformation process. The *Transplant_<donor-host names>* directory serves as the current transplantation directory, containing product line-specific files and subdirectories.

⁶<https://www.doxygen.nl>

⁷<https://www.gnu.org/software/automake/>

⁸<https://libgit2.github.com/>

⁹<https://www.gnu.org/software/make/>

¹⁰https://gnupg.org/related_software/libgrypt/

¹¹<https://libcheck.github.io/check/>

¹²<https://www.freedesktop.org/wiki/Software/pkg-config/>

¹³<https://www.gnu.org/s/diffutils/>

¹⁴<https://www.gnu.org/software/libtool/>

Table A.1: TRANSPLANTATION FOLDER STRUCTURE

ROOT		
	-- <i>prodsicalpel.exec</i>	# Exec file
	-- <i>ErrorFile.out</i>	# TXL errors
	-- <i>Transplant_ < donor – hostnames ></i>	#Current experiment directory
.	-- <i>CFLAGS.txt</i>	# Flags to the current experiment
.	-- <i>coreFunctions.in</i>	# Core function input list (Feature entry points)
.	-- <i>Donor</i>	# Donor system directory
.	-- <i>Doxygen_ < donor_name ></i>	# Doxygen directory
.	-- <i>Host</i>	# Host system folder
.	-- <i>Temp</i>	# Temporary folder for the transplant process
.	-- <i>TempDonorFiles</i>	# Temporary folder for Donor system
.	-- <i>TempImplantationDirectory</i>	# Temporary folder for Implantation files
.	-- <i>TempSourceFiles</i>	# Temporary folder for the transplant process
.	-- <i>TestSuites</i>	# Testing directory
.	-- <i>TransplantCode</i>	# Temporary folder for the organ
	-- <i>TXL(FormacOS)</i>	# TXL for macOS directory
	-- <i>TXL_LINUX(ForLINUX)</i>	# TXL for Linux directory
...		# Tool source code

Within the transplantation directory there are various subdirectories and files. The *CFLAGS.txt* file holds the flags specific to the current product line, providing configuration information. The *coreFunctions.in* file contains a list of core function input points, serving as entry points for the features being transplanted.

The *Donor* directory holds the donor’s codebase, while the *Doxygen_<donor_name>* directory is used by PRODSICALPEL to generate the the call graphs. The *Host* directory corresponds to the host system’s codebase (product base). The *Temp* directory serves as a temporary folder for the transplantation process, housing intermediate files and directories. The *TempDonorFiles* directory stores temporary files specific to the donor system, while the *TempImplantationDirectory* directory holds temporary files related to the implantation process. The *TempSourceFiles* directory stores temporary files used during the transplantation process.

The *TestSuites* directory represents the *IceBox* directory. It is dedicated to keep the IceBox tests, containing test suites and associated files. The *TransplantCode* directory , where the transplanted code is stored. Additionally, there are separate directories for the *TXL* tool, with *TXL* for macOS and *TXL_LINUX* for Linux.

The binary with example usage is available in the project webpage¹⁵. It also contains an example run for the *Mytar* Donor - *NEATVI* Host transplant. The PRODSICALPEL binary release was compiled on 64-bit MacOS 10.15.4.

Running prodScalpel. The complete command, as it should be pasted is:

```

1 prodScalpel --seeds_file <transplantation_directory>/seed-1.in \
2   --compiler_options <transplantation_directory>/CFLAGS \
3   --host_target <transplantation_directory>/productBase/insert_point_file.c \
4   --donor_target <donor_directory>/Donor/organ_souce_file.c \
5   --donor_folder <donor_directory>/Donor/ \

```

¹⁵<https://autotransplantation-spl.github.io/foundry.github.io/>

```

6  --workspace <transplantation_directory>/ \
7  --core_function <core_funtion_name> \
8  --host_project <transplantation_directory>/ProductBase

```

You should run this from root folder. The Organ is automatically grafted into the host program, so, for subsequent runs the original version of the host must be restored. If you wish to run PRODSICALPEL on your own transplants, you will need to keep the same folder structure as shown in our examples. The required and optional parameters of PRODSICALPEL are:

`--seeds_file /path/to/file:` *(required) take the seeds from a file. The file must contain 7 lines of 4 numbers each, as in this example.*

`--transplant_log /path/to/folder/:` *(optional) log the results of the transplantation operations, in every generation.*

`--compiler_options /path/to/file:` *(optional) required if the compilation of the code in donor requires additional options or libraries. The format of this file is: CFLAGS = 'libgcrypt-config --libs'. The variable CFLAGS contains all the additional dependencies.*

`--donor_folder/path/to/folder/:` *(required) the folder where is the source code of the donor.*

`--workspace /path/to/folder/:` *(required) the workspace of the transplantation.*

`--txl_tools_path /path/to/folder/:` *(optional) used when the binary files with extension *.x are in a different place than PRODSICALPEL*

`--host_project /path/to/folder:` *(required) the folder where is the source code of the host.*

`--donor_entry_function /path/to/file:` *(required) the function in the donor that correspond to its entry_point (generally the main function).*

`--donor_entry_file: /path/to/file:` *(required) the file in the donor that contains the entry_point (generally `main.c`).*

`--conditional_directives:` *(optional) directive in case when the organ and host must be merged. PRODSICALPEL introduces variability into the organ by inserting this conditional directive around the organ's code, making it variable.*

`--product_base:` *(required) the version of product base after the organ transplantation process.*

Additional parameters:

`--exclude_functions /path/to/file:` *(optional) exclude some functions from the transplantation algorithm.*

`--transplant_statistics /path/to/file:` *(optional) log statistics about the transplantation operation.*

`--urandom_seeds:` PRODSICALPEL will take its seeds from `/dev/urandom`

`--random_seeds:` PRODSICALPEL will take its seeds from `/dev/random`. This may take a while. The default option is `--urandom_seeds`.

For a new organ transplantation change the file `coreFunctions.in`. For example, to transplant the feature `write_archive` from MYTAR to the product base NEATVI the complete command, as it should be pasted in the file is:

```
--coreFunction write_archive
--donorSystem MYTAR
--donorFileTarget Transplant-PRODUCT_BASE/Donor/append.c
--hostFileTarget Transplant-PRODUCT_BASE/ProductBase/NEATVI/ex.c
```

Where:

`--core_function function_name`: *(required) the entry point of the functionality to transplant.*

`--donorSystem`: *(required) the donor system name.*

`--donor_target /path/to/file`: *(required) the file in the donor, with the function annotated for transplantation.*

`--host_target /path/to/file`: *(required) the file in the host that contains the `__ADDGRAFHERE__JUSTHERE` annotation. This annotation is required, and it marks the place where the organ will be added.*

EXPERIMENT: DATA AND SUPPORT MATERIAL

This appendix lists the documentation, data, script, and support material used in the experimental study, earlier addressed in Chapter 8.

This appendix is organized as follows: Section B.1 presents the online survey used to get the profile of each participant; B.2 presents the online survey to get the participants feedback about the experiment execution process; Sections B.3 and B.4 show the experiment tasks respectively, i.e., the instructions followed by the participants during the experiment execution; In Section B.5 we present the results of post-execution survey; and Section B.6 presents the time spent by each participant to execute the tasks.

B.1 ONLINE PRE-SURVEY (BACKGROUND FORM)

The online survey used to get the profile of each participant.

PARTICIPANTS BACKGROUND FORM

Objective: The following questionnaire was constructed to collect pre-execution informations about each experiment participant.

Context: This questionnaire is part of an experiment to analyse the effectiveness and efficiency of our Software Transplantation approach compared with the manual process of generating a product line from existing systems, performed by SPL experts. Software transplantation is "the adaptation of one system's behaviour or structure to incorporate a subset of the behaviour or structure of another"

Instructions and Target Population: This questionnaire should be answered by software engineers which have any experience with Software Product lines. The questionnaire is composed of 9 questions.

Confidentiality: The information obtained with this questionnaire will be only used for academic purposes, and no identification of the respondents is required. This questionnaire is part of a Ph.D. thesis to develop new solution approaches to Software Product Line Engineering via Software Transplantation, supervised by Prof. Ph.D. Eduardo Santana de Almeida from the Computer Science Department in Federal University of Bahia (UFBA, Brazil), Ph.D. Earl T. Barr from the Computer Science Department in University College London., Ph.D. Justyna Petke from the Computer Science Department in University College London.

Leandro Oliveira de Souza
Ph.D. student in Computer Science - UFBA
Professor at Federal Institute of Bahia

Thank you for participating in this study. All information you provide in completing this questionnaire will remain anonymous. We do not let your personal information about analytics.

***Mandatory**

****Questions applied only in the company**

* Indica uma pergunta obrigatória

Personal information

This information will help us analyze the results of the experiment according to the characterization of the group.

1. **Email address*:** *

2. **Full name*:** *

3. **Company and/or University**:** *

4. **What is your level of education? ***

Marcar apenas uma oval.

- High School
- Undergraduate
- Non-degree Graduate Course
- M.Sc.
- Ph.D.
- Pos-Ph.D.

5. **Position in the Company**:**

Marque todas que se aplicam.

- Software Engineer
- Developer
- Analyst
- Researcher
- Outro: _____

Experience with software development

6. How many years of programming experience do you have using any programming language?

Marcar apenas uma oval.

- < 1 year
- >= 1 year and < 5 years
- >=5 years and < 10 years
- >= 10 years

Experience in the area of Software Product Lines

We would like to know about your experience with Highly Configurable Systems (HCS) and Software Product Lines (SPL) (project and industry).

7. **Regarding your Software Product Line (SPL) background knowledge: (Choose only one)**

Marcar apenas uma oval.

- I have been involved in software development teams applying HCS / SPL approaches.
- I am a researcher working on topics related to HCS / SPL development.
- I know what HCS / SPL is, but I have never participated in a software project applying HCS / SPL development.

8. **How many years of experience do you have in SPL (consider industry and academic)**

Marcar apenas uma oval.

- < 1 year
- >= 1 year and < 5 years
- >=5 years and < 10 years
- >= 10 years

9. **Have you applied the SPL approach to software development? (choose only one)**

Marcar apenas uma oval.

- Yes, but only in the research domain
- Yes, but only in the industry domain
- Yes, both research and industry domain
- no

Este conteúdo não foi criado nem aprovado pelo Google.

Google Formulários

B.2 ONLINE POST-SURVEY (FEEDBACK FORM)

The online survey to get the participants feedback about the experiment execution process

FEEDBACK FORM (POST-EXECUTION)

Thank you for participating in this study. All information you provide in completing this questionnaire will remain anonymous. We do not let your personal information about analytics.

Feel free to answer the questions in English or Portuguese.

***Mandatory**

1. **Full name*:**

TRAINING

2. **How effective do you think the training was?**

Marcar apenas uma oval.

- It was effective, helped to understand the tasks, steps and artifacts of the process
- It was effective, helped to understand the tasks, steps and artifacts of the process, but the time needed to be longer
- It would be more effective if there were more examples
- Very intuitive activity, but a good experience is needed to apply it according to the rules and estimated time
- A model should be shown, following step by step all the possible details that may occur during the process

3. Comments:

4. Were you in doubt about any concept or activity used in the manual process? If yes, how did you handle it?

Marcar apenas uma oval.

- Yes. Asked for an explanation to the instructor
- Yes. Reviewed the training material.
- No.

5. In addition to the knowledge acquired in training, you needed other information to perform the processes:

Marcar apenas uma oval.

- Yes
- No

6. If yes, which ones?

ABOUT THE FEATURE IDENTIFICATION AND EXTRACTION

- 7. **Did you experience difficulties in carrying out feature IDENTIFICATION/EXTRACTION steps during the feature transferring process? Which one(s)?**

- 8. **On a Scale Of 1 To 5, how difficult was to complete this stage to you? Please, do not answer it if you had no time to start it.**

Marcar apenas uma oval.

- 1. Very easy
- 2. Easy
- 3. Neutral
- 4. Difficult
- 5. Very difficult

ABOUT THE FEATURE ADAPTATION STAGE

- 9. **Did you experience difficulties in carrying out feature ADAPTATION steps during the feature transferring process? Which one(s)?**

10. **On A Scale Of 1 To 5, how difficult was to complete this stage to you? Please, do no answer it if you had no time to start it.**

Marcar apenas uma oval.

- 1. Very easy
- 2. Easy
- 3. Neutral
- 4. Difficult
- 5. Very difficult

ABOUT THE FEATURE ADAPTATION STAGE

11. **Did you experience difficulties in carrying out feature INSERTION steps during the feature transferring process? Which one(s)?**

12. **On A Scale Of 1 To 5, how difficult was to complete this stage to you? Please, do no answer it if you had no time to start it.**

Marcar apenas uma oval.

- 1. Very easy
- 2. Easy
- 3. Neutral
- 4. Difficult
- 5. Very difficult

ABOUT THE FEATURE TRANSFERRING PROCESS TO REALIZE AN PRODUCT LINE

13. **Did the use of any tool increase the efficiency of the feature transferring process?**

14. **In general, did you have any difficulties in executing the process? Which one(s)?**

15. **Was you able to finish all step in the time limited of 4 hours? If not, why?**

16. **Have you ever used or knows any tools that could be use to support any stage of th process?**

MANUAL APPROACH

17. **I believe the manual approach is robust enough for SPL generation**

Marcar apenas uma oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree

18. **The manual process is complex when you do not know how the donor system was implemented.**

Marcar apenas uma oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree

Send form

Thank you for your feedback.

19. **Do you have any additional suggestion, comment, etc? Please share with us.**

Este conteúdo não foi criado nem aprovado pelo Google.

Google Formulários

B.3 EXPERIMENT TASKS: SCENARIO I

Experiment tasks executed in scenario I, i.e., the instructions followed by the participants during the experiment execution.

EXPERIMENT SCRIPT FOR THE SCENARIO I (GROUP A)

1. SUMMARY

The purpose of the experiment is to analyze the transplant process (transfer) feature manual between two systems, the donor and the receiver (host), both written in C. For this process, we will compute the necessary activities, time and effort spent by a set of SPL specialists compared to our automated approach proposal. The process consists of four steps: extract, identify, adapt, and insert.

The participants will have access to a set of unit testing, a feature's entry point in the donor (a function) and an insertion point in the host system provided by the researchers. Participants may use the tools (s) that they want to try to perform the process in the shortest time possible. We request to the participants to compute all activities, step, and time spent as well as the tool of the tool used.

It is important to highlight that the experiment will not need to be performed without any breaking, since they compute time and activities correctly with each return to the execution process.

2. TRAINING

We have prepared some videos describing each section of this script. You can assist at any time while performing the experiment. We only ask you not to register this time in the time and effort spreadsheet.

3. EXPERIMENT EXECUTION ACTIVITIES

1. Download the experiment files, available at: [experiment directory](#)
2. Access experiment form containing the access link to your time and effort registration spreadsheet: [registration sheet](#).
3. Install dependencies.
4. Execute the Feature transferring process.
5. Execute the unit test.
6. Perform system testing in the post-transference receiver system.
7. Send files to researchers via the experiment form.
8. Signal the end of the experiment to researchers.

4. EXPERIMENT DIRECTORY

Execution Directory: As part of the experiment preparation process, the participant should download the files contained in the folder corresponding to the group in which he was allocated. All files needed for the experiment can be downloaded by the link: [transplantation directory](#)

Effort Register Worksheet: For time and effort registration, we provide a spreadsheet for each participant. You can it by the link: [transplantation directory](#)

4.1. DIRECTORIAL STRUCTURE

GROUP A
| _ Dependencies_run -> Dependency Installation Files

- |_ Documentation -> Contains the Donor and host system documentation if the participant wants to better understand the systems.
- |_ Owner -> Contains the source code of the donor system.
- |_ Host_original -> Contains the source code of the receiving system (host)
- |_ Host_to_transplant -> Contains the host source code to receive the feature.
- |_ Test_Suite -> Temporary directory for feature and unit tests
- |_ Feature -> Temporary directory of the feature source code insertion

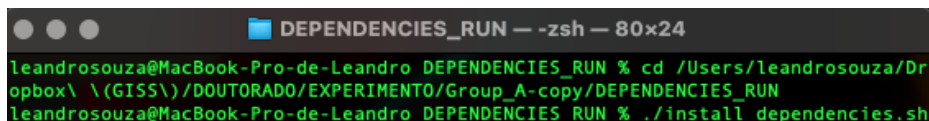
5. DEPENDENCIES INSTALATION

Please install some dependencies required to execute unit tests. **Do not worry about installing these premises, at the end you can run the script that will be automatically generated after the facilities in the directory will be installed Installation of dependencies (dependencies_run)**

We have prepared a shell script to install the dependencies automatically from the directory: `Dependencies_install`

5.1. INSTALLATION OF DEPENDENCIES (FOR MAC OS)

From the directory `Dependencies_Run` execute the following command: `./install_dependencies.sh`



```

leandrosouza@MacBook-Pro-de-Leandro DEPENDENCIES_RUN % cd /Users/leandrosouza/Dr
opbox\ \(\GISS\)/DOCTORADO/EXPERIMENTO/Group_A-copy/DEPENDENCIES_RUN
leandrosouza@MacBook-Pro-de-Leandro DEPENDENCIES_RUN % ./install_dependencies.sh

```

Terminal screen with CD commands e `./install_dependencies.sh`

This command will install the following dependencies:

- [Autoconf](#)
- [libtool](#)
- [Pkg-config](#)
- [Check](#)
- [Glib](#)

6. DESCRIPTION OF SYSTEMS AND FEATURE

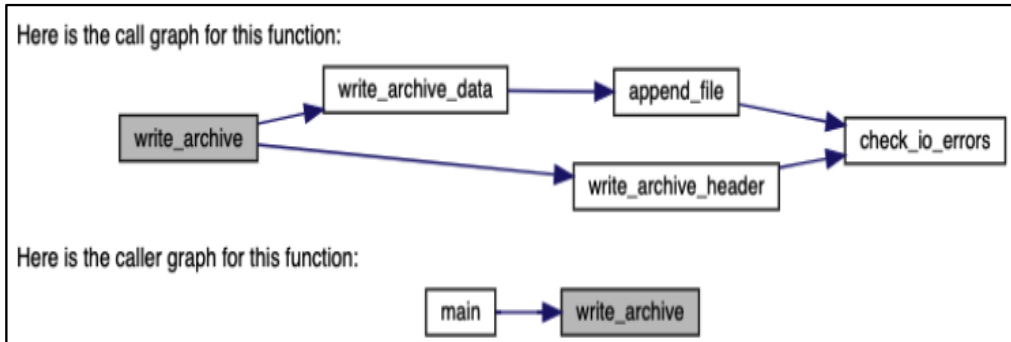
We have divided the participants into two groups (A and B). This script describes the process to be performed by group A. The participant of group A have to try to transfer the feature `WRITE_ARCHIVE` from the *Mytar* system, a file manager in *NeatVI*, an editor of text. When executed, the feature `WRITE_ARCHIVE` copies all contents of a file and writes it in other with a different format. To do this, the feature must receive as input at least the name of input file and the target output file.

6.1. AVAILABLE ARTIFACTS

As they are distinct systems it will be necessary to adapt the code that implements the entry point of the feature (ie the `write_Archive` function) for the insertion point in the *Neatvi* system, based on the following information:

- Feature inserting point: The `WRITE_ARCHIVE` function of the `append.c` file.

- Host system insertion point: point defined by notation with `__ADDGRAFTHERE__JUSTHERE` in the `ex.c` file
- Call_graph: The Feature call graph available.
- Test_suite: A set of unit tests used to test the feature. It can be used to assist in process of the feature adapting and executing before insertion. Such test files are available in the experiment directory.



These artifacts are described more detailed in the following sections.

7. UNIT TEST EXECUTION

When identifying, extract the code elements and perform the necessary adaptation you can perform the unit tests available in the `test_suite.c` file. For this, you must execute the script in `run_test.sh`

If Feature is working properly, you will receive the message:

```

UNIT_TEST -- zsh -- 80x24
leandrosouza@MacBook-Pro-de-Leandro UNIT_TEST % ./Individual.x
Archive TEXT.TAR: writing file FILE.in
1 files in archive
Offset: 0      Length: 184      Name: FILE.in

100%: Checks: 1, Failures: 0, Errors: 0
leandrosouza@MacBook-Pro-de-Leandro UNIT_TEST %
  
```

Terminal with unit testing informing: 100% percentage of acceptance; Checks: 1. Number of tests performed; Failures: 0 number of failures and errors: 0, number of errors.

8. TRANSPLANT VALIDATION

After transferring the feature to the host system and it is passing on the unit tests, you can perform the final transplant validation test. We have provided a post-operative test suite. The post-operative tests correspond to a set of regression, augmented regression and acceptance tests to exercise the feature transplanted and check if the transplant has not broken the host system.

Executing the post-operative tests:

1. Open the terminal.
2. Access the directory **HOST_TO_TRANSPLANT/NEATVI_1.0/TRANSPLANTATION_TEST_CASES** where you insert the feature source code.
3. Execute the commands: `./test_product_line.sh`

9. SENDING ALL ARTIIFACTS

After the execution of the experiment is completed, make sure that it computed all the time elapsed at each stage of the process. Sign up to the researchers to end the activity and [time registration worksheet](#).

Rename Group B Directory Complete your name and compact the folder. Then upload the folder with your changes from the [form](#).

If you have a shipping problem, download the artifacts from [transplantation artifacts](#) and report this to the researchers

We greatly appreciate the time and availability to perform this vital experiment for the progress of our research.

B.4 EXPERIMENT TASKS: SCENARIO II

Experiment tasks executed in scenario II, i.e., the instructions followed by the participants during the experiment execution.

EXPERIMENT SCRIPT FOR THE SCENARIO II (GROUP B)

1. SUMMARY

The purpose of the experiment is to analyze the manual feature transfer process between two systems, the donor and the receiver (host), both written in C. For this process, we will record the necessary activities, time, and effort spent by a set of SPL specialists compared to our proposed automated approach. The process consists of four steps: extraction, identification, adaptation, and insertion.

The participants will have access to a set of unit tests, a feature's entry point in the donor (a function), and an insertion point in the host system provided by the researchers. Participants may use the tools they prefer to try to perform the process in the shortest time possible. We request that participants record all activities, steps, and time spent, as well as the tool they used.

It is important to note that the experiment will not require participants to perform the process without any breaks, as long as they record time and activities correctly each time they return to the execution process.

2. TRAINING

We have prepared some videos describing each section of this script. You can assist at any time while performing the experiment. We only ask you not to register this time in the time and effort spreadsheet.

3. EXPERIMENT EXECUTION ACTIVITIES

1. Download the experiment files, available at: [experiment directory](#);
2. Access experiment form containing the access link to your time and effort registration spreadsheet: [registration sheet](#);
3. Install dependencies;
4. Execute the Feature transferring process;
5. Execute the unit test;
6. Perform system testing in the post-transference receiver system;
7. Send files to researchers via the experiment form;
8. Signal the end of the experiment to researchers.

4. EXPERIMENT DIRECTORY

Execution Directory: As part of the experiment preparation process, the participant should download the files contained in the folder corresponding to the group in which he was allocated. All files needed for the experiment can be downloaded by the link: [transplantation directory](#)

Effort Register Worksheet: For time and effort registration, we provide a spreadsheet for each participant. You can it by the link: [effort form](#)

4.1. DIRECTORIAL STRUCTURE

GROUP B

- |_ Dependencies_run -> Dependency Installation Files
- |_ Documentation -> Contains the Donor and host system documentation if the participant wants to better understand the systems.
- |_ Owner -> Contains the source code of the donor system.
- |_ Host_original -> Contains the source source code of the receiving system (host)
- |_ Host_to_transplant -> Contains the host source code to receive the feature
- |_ Test_Suite -> Temporary directory for feature and unit tests
- |_ Feature -> Temporary directory of the feature source code insertion

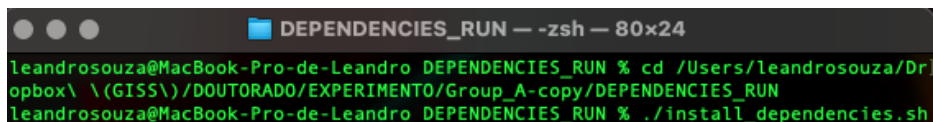
5. DEPENDENCIES INSTALATION

Please install some dependencies required to execute unit tests. **Do not worry about installing these premises, at the end you can run the script that will be automatically generated after the facilities in the directory will be installed Installation of dependencies (dependencies_run)**

We have prepared a shell script to install the dependencies automatically from the directory: `Dependencies_Run` execut

5.1. INSTALLATION OF DEPENDENCIES (FOR MAC OS)

From the directory `Dependencies_Run` execut the following command: `./install_dependencies.sh`



```

DEPENDENCIES_RUN -- zsh -- 80x24
leandrosouza@MacBook-Pro-de-Leandro DEPENDENCIES_RUN % cd /Users/leandrosouza/Dr
opbox\ \ (GISS)\ /DOCTORADO/EXPERIMENTO/Group_A-copy/DEPENDENCIES_RUN
leandrosouza@MacBook-Pro-de-Leandro DEPENDENCIES_RUN % ./install_dependencies.sh

```

Terminal screen with CD commands e `./install_dependencies.sh`

This command will install the following dependencies:

- [Autoconf](#)
- [libtool](#)
- [Pkg-config](#)
- [Check](#)
- [Glib](#)

6. DESCRIPTION OF SYSTEMS AND FEATURE

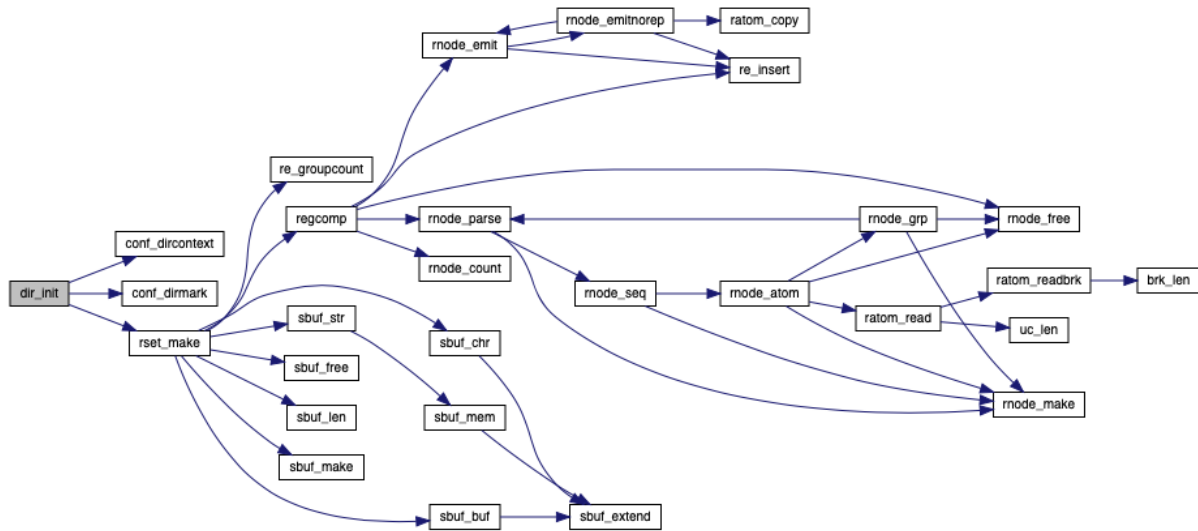
We divide the participants of our experiment into two groups (A and B). This script describes the process to be performed by group B participants. As a member of Group B, you will try to transfer the feature `dir_init` between two versions of the *Neatvi* text editor, *Neatvi_1.0* version (receiver system) and *Neatvi_2.0* (fetal donor). The feature `dir_init`, present only in the *Neatvi_2.0* version, allows the opening of a file already created. For this, the function performs changes in global variables:

```

struct rset *dir_rslr; /* pattern of marks for left-to-right strings */
struct rset *dir_rsrli; /* pattern of marks for right-to-left strings */
struct rset *dir_rscctx; /* direction context patterns */

```

, present in the *dir.c*. Without such variables will be null but do not interfere with the functioning of the system. To function correctly it will be necessary to extract all functions called from the *dir_init ()* function implemented in the *dir.c* file, following its call chart.



6.1. AVAILABLE ARTIFACTS

As they are distinct systems it will be necessary to adapt the code that implements the entry point of the feature (ie the *dir_init()* function) for the insertion point in the *NEATVI* system, based on the following information:

- **Feature input point:** place where a call to the *dir_init ()* function will be inserted, the entry point of the feature implemented in the *dir.c* file.
- **Host system insertion point:** point defined by notation with `__ADDGRAFTHERE_JUSTHERE` in file *vi.c*
- **Call_GRAPH:** The Feature call chart available both in your documentation in `documentation/owner_documentation/index` and can be viewed through the `documentation/call_graph.png` image.
- **Test Suite:** A set of unit tests that can be used to assist in adapting and executing the feature before its insertion. Such test files are available in the transfer directory and have already been implemented by the researchers, and participants are only the task of executing them from the commands described in Section 6.

These artifacts are described more detailed in the following sections.

7. FEATURE TRANSPLANTATION PROCESS

In this section, we describe the systems and features involved in the transfer process and the step-by-step process for executing the feature transfer. We provide each participant with training [mini-videos](#) of the process that can be viewed as many times as necessary. If there are still any questions after reading the step-by-step instructions and viewing the training, please contact the researchers.

The purpose of this experiment is to analyse the manual feature transfer process between two versions of the same system written in C. You may use the IDE you feel comfortable with or just a simple text editor. You should record the activities and time spent on the individual time collection spreadsheet provided. Consider any action taken during each stage of the process as an activity, for example, IDENTIFICATION STAGE: searching for global variables used by extracted functions. Do not count the time for training and reading these scripts. Below we highlight the stages of the process that will be counted.

7.1. PROCESS STEP

Once the premises are installed, you can start execution of the experiment. Have the activity and time -spent [registration worksheet](#) available.

We divide the experiment into 4 steps: extraction, identification, adaptation and insertion of feature. Next we describe each step.

We divide the experiment into 3 steps: extraction, adaptation and insertion of feature. Next we describe each step.

1. Extraction: corresponds to the extraction process of the entire portion of code that implements the feature, necessary for its execution (functions) from the function that determines the entry point of the feature and its call chart. All code implements the feature should be copied to the host_to_transplant directory, environment with the host system. If the function already exists in the receiving system, you should involve it with the `F_DIR_INIT` directive.

All copied elements must be delimited with FLAG `F_DIR_INIT`. As it is the transfer of features between two versions of the same system it will be common to find functions already implemented in the receiving version. In this case, you should delimit the function with FLAG `F_DIR_INIT_TOO`. Both flags are already defined in file `vi.h` (lines 4 and 5).

To facilitate this process, we provide the call graph. The full version of it is available in the Documentation directory.

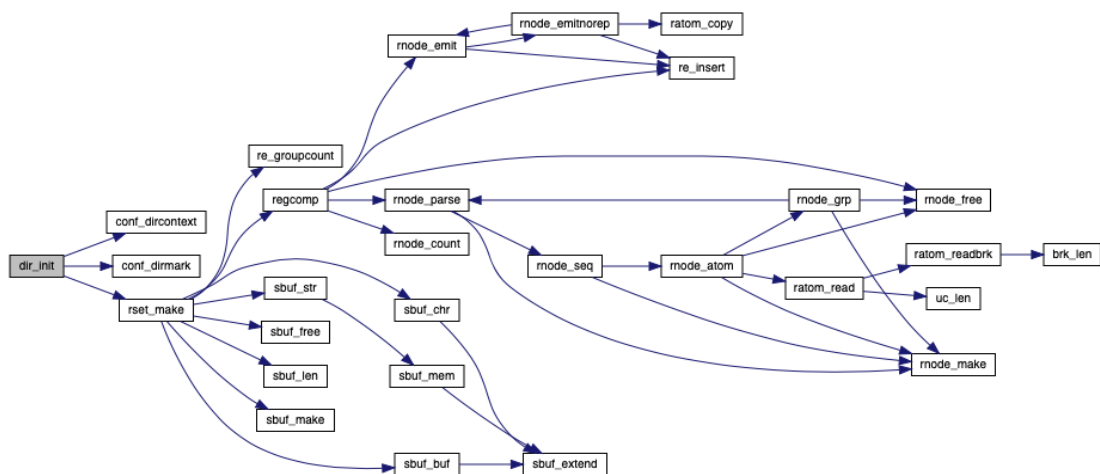


Grafico de chamada da feature `dir_init`

- 2. Adaptation:** You should, in addition to extracting the functions called by Feature, analyze whether any portion of receiving system code needs to be adjusted to load the parameters correctly from the feature point of the feature in the host environment. The inserting point is defined with the notation: `__ADDGRAFHERE__JUSTHERE`, which can be found in the *main* function of file *vi.c*.

3.1 Test Execution: Once all these steps are done, the host system will be ready to perform the unit tests already implemented and made available in the **Test_Suite** Directory. These tests have already been implemented, requiring the participant only to execute the commands described in *Section 6* of this document. It is possible that in the first execution the tests do not go through the lack of an incorrect source code or adaptation. In this case, you should analyze if the adaptation performed is correct and feature has all the code for its execution, fixing any errors and running again the unit test execution commands until the test passes without errors, as can be seen in the figure below. See how to perform the unit tests in the next section.

```
100%: Checks: 1, Failures: 0, Errors: 0
leandrosouza@MacBook-Pro-de-Leandro UNIT_TEST %
```

OBS: The proper feature compilation process will generate an individual file.x. Only after the generation of this file and subsequent execution of failure -free tests, as per section 6, we can say that the feature has been correctly extracted from the donor and its files can be inserted into the host system directory.

- 3. Mergin:** Once the code is extracted and the unit tests are performed you can enter the feature insertion into the host environment. This process consists of inserting a call to the insertion point replacing the notation `__ADDGRAFHERE__JUSTHERE` by the chamda to the *dir_init()* function. You must access the *vi.c* file, within the host system directory, and identify the notation `__ADDGRAFHERE__JUSTHERE`. This annotation signals the insertion point of a call to the feature. Instead of this notation you should insert a call to the *dir_init* function, adding the parameters, if necessary, among the local variables declared in the main function.

Once all these steps are completed and feature is passing on the unit tests, you will be able to execute the host execution command and use feature in the host. The details of this process are given in *Section 7* of this document.

8. UNIT TEST EXECUTION

When identifying, extracting the code elements, and performing the necessary adaptation you can perform the unit tests made available within the *Test_Suite* directory. For this, you must open the terminal, access the *Test_Suite* Directory and execute the command: `./run_test.sh` that will compile the copied feature and execute it. He may inform some Warning that you should disregard but should not generate errors. If you are informed of the compilation process, you must correct the error in the transferred code.

```
TEMP -- zsh -- 80x24
\ (GISS)\DOCTORADO\EXPERIMENTO\Group_A-copy\TEMP
leandrosouza@MacBook-Pro-de-Leandro TEMP % ./run_test.sh
```

Terminal screen with the execution of comand `./run_test.sh`.

If compilation occurs successfully and no error is informed, the above command should generate an individual file.x in the test directory. Once this file is generated it can be executed with the command: `./Individual.x` that will test the feature. If Feature is working properly, you will receive the message:

```
UNIT_TEST -- zsh -- 80x24
leandrosouza@MacBook-Pro-de-Leandro UNIT_TEST % ./Individual.x
Archive TEXT.TAR: writing file FILE.in
1 files in archive
Offset: 0      Length: 184      Name: FILE.in

100%: Checks: 1, Failures: 0, Errors: 0
leandrosouza@MacBook-Pro-de-Leandro UNIT_TEST %
```

Terminal with unit testing informing: 100% percentage of acceptance; Checks: 1. Number of tests performed; Failures: 0 number of failures and errors: 0, number of errors.

9. TRANSPLANT VALIDATION

After transferring the feature to the host system and it is passing on the unit tests, you can perform the final transplant validation test. We have provided a post-operative test suite. The post-operative tests correspond to a set of regression, augmented regression and acceptance tests to exercise the feature transplanted and check if the transplant has no broken the host system.

Executing the post-operative tests:

1. Open the terminal;
2. Access the directory `HOST_TO_TRANSPLANT/NEATVI_1.0/TRANSPLANTATION_TEST_CASES` where you insert the feature source code.
3. Execute the commands: `./test_product_line.sh`

10. SENDING ALL ARTIIFACTS

After the execution of the experiment is completed, make sure that it computed all the time elapsed at each stage of the process. Sign up to the researchers to end the activity and [time registration worksheet](#).

Rename Group B Directory Complete your name and compact the folder. Then upload the folder with your changes from the [form](#).

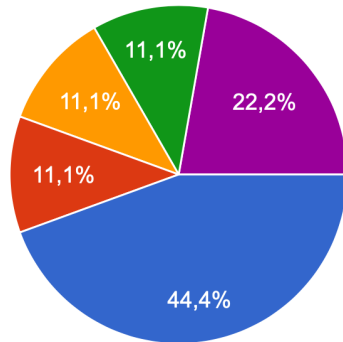
If you have a shipping problem, download the artifacts from [transplantation artifacts](#) and report this to the researchers

We greatly appreciate the time and availability to perform this vital experiment for the progress of our research.

B.5 EVALUATION: RESULTS OF POST-EXECUTION SURVEY

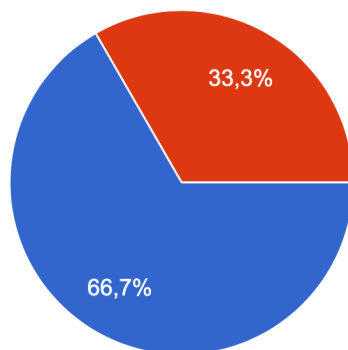
RESULT OF POST-EXECUTION SURVEY

How effective do you think the training was?



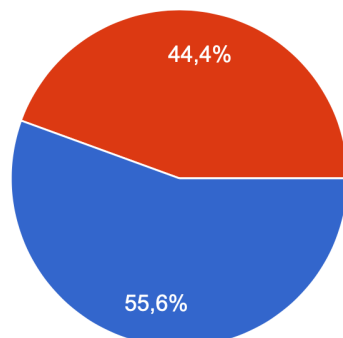
- It was effective, helped to understand the tasks, steps and artifacts of the pr...
- It was effective, helped to understand the tasks, steps and artifacts of the pr...
- It would be more effective if there were more examples
- Very intuitive activity, but a good experience is needed to apply it acco...
- A model should be shown, following step by step all the possible details th...

Were you in doubt about any concept or activity used in the manual process? If yes, how did you handle it?



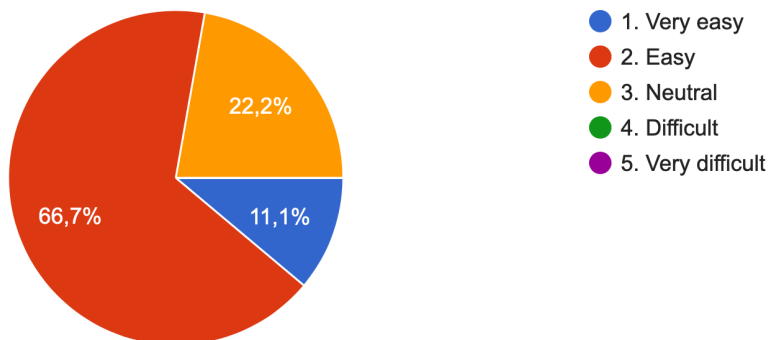
- Yes. Asked for an explanation to the instructor
- Yes. Reviewed the training material.
- No.

In addition to the knowledge acquired in training, you needed other information to perform the processes:

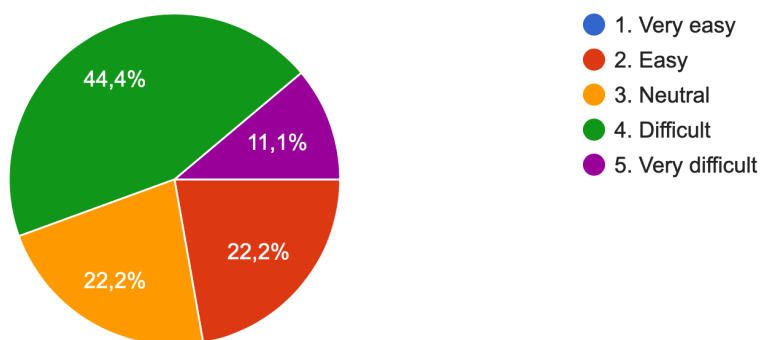


- Yes
- No

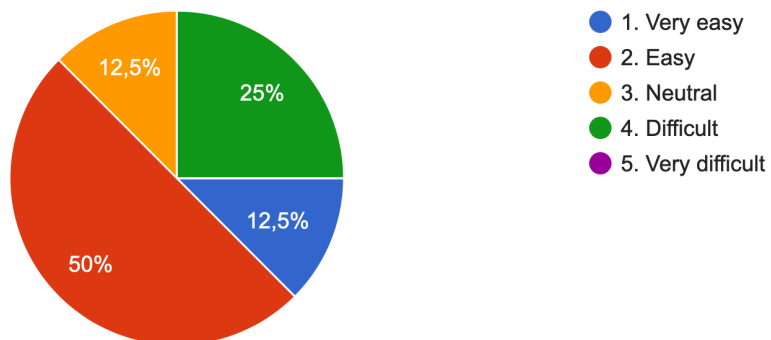
On a Scale Of 1 To 5, how difficult was to complete this stage to you? Please, do not answer it if you had no time to start it.



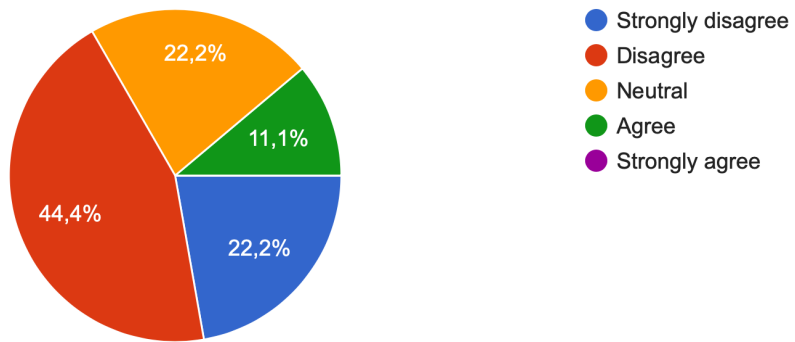
On A Scale Of 1 To 5, how difficult was to complete this stage to you? Please, do not answer it if you had no time to start it.



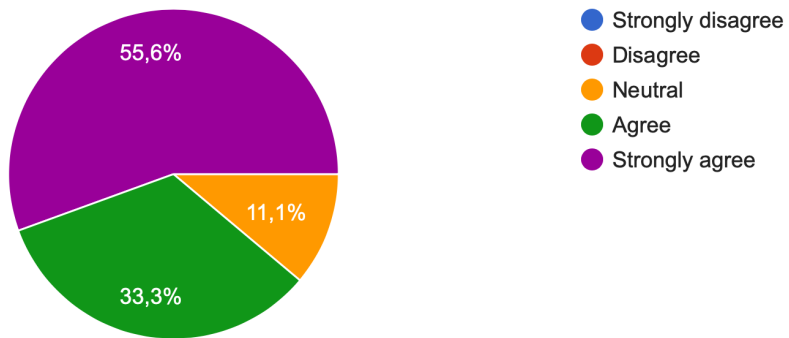
On A Scale Of 1 To 5, how difficult was to complete this stage to you? Please, do not answer it if you had no time to start it.



I believe the manual approach is robust enough for SPL generation



The manual process is complex when you do not know how the donor system was implemented.



B.6 EVALUATION: THE TIME MEASURED FOR THE PARTICIPANTS AND TOOL

The time spent by each participant to execute the tasks.

Table B.1: Scenario I.

The time measured for the participants that transferred a feature from NEATVI to NEATVI as a product base.

FASES/PARTICIPANTS	PRODSCALPEL	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	MIN.	MAX.	AVG.	AV. WITH PRODSCALPEL
Extraction	18	24	35	69	17	50	211	56	30	59	106	17	69	49.6	67.5
Adaptation	0	42	26	3	39	15	15	3	23	4	2	2	42	17.4	17.2
Merging	2	16	27	5	12	16	14	28	30	10	5	5	30	16.6	16.5
Total	20	82	88	77	68	81	240	87	83	73	113			83.6	101.2

Table B.2: Scenario II.

The time measured for the participants that transferred a feature from MYTAR to NEATVI as a product base.

FASES/PARTICIPANTS	PRODSCALPEL	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	MIN.	MAX.	AVG.	AV. WITH PRODSCALPEL
Extraction	0	20	35	69	45	21	31	43	20	38	30	21	69	33.67	32.00
Adaptation	27	220	205	104	195	219	59	53	171	90	210	53	205	104.67	141.18
Merging	0	0	0	8	0	0	24	8	3	3	0	8	24	4.67	4.18
Total	27	240	240	181	240	240	114	104	194	131	240			143.00	177.36

Table B.3: Total time (in minutes) spent by on performing the three stages of SPL reengineering: feature extraction, adaption and merging.

The highlights line show the execution that achieved the timeout without no result.

EXECUTION	SCENARIO I			SCENARIO II		
	START	END	TIME	START	END	TIME
1	17:37:30	18:22:04	00:44:34	20:06:07	20:27:08	00:21:01
2	18:22:04	18:43:20	00:21:16	20:06:07	20:48:46	00:21:20
3	18:43:30	00:00:00	04:00:00	20:27:26	21:09:53	00:20:48
4	22:27:09	22:43:38	00:16:29	20:49:05	21:31:04	00:10:54
5	22:43:38	23:08:18	00:24:40	21:20:10	21:52:00	00:20:38
6	23:24:13	23:49:09	00:15:40	21:31:22	22:13:16	00:20:58
7	23:49:18	00:09:14	00:24:56	21:52:18	22:34:29	00:20:54
8	00:09:22	00:44:01	00:19:56	22:13:35	22:55:44	00:20:56
9	08:13:35	09:23:14	00:34:39	22:34:48	23:16:56	00:20:53
10	00:44:08	01:02:01	01:09:39	22:56:03	23:38:19	00:21:06
11	01:02:09	01:15:32	00:17:53	23:17:13	23:59:34	00:20:57
12	01:15:41	01:28:38	00:13:23	23:38:37	00:20:47	00:20:53
13	01:28:46	01:50:56	00:12:57	23:59:54	00:41:54	00:20:49
14	01:51:04	02:16:23	00:22:10	00:21:05	01:03:02	00:20:50
15	02:16:32	03:32:46	00:25:19	00:42:12	01:24:11	00:20:50
16	02:32:53	03:21:21	01:16:14	01:03:21	01:45:17	00:20:48
17	03:21:30	03:35:17	00:48:28	01:24:29	02:06:25	00:20:49
18	03:35:26	03:56:14	00:13:47	01:45:36	02:27:33	00:20:50
19	03:56:21	04:10:25	00:20:48	02:06:43	02:48:40	00:20:47
20	04:10:32	04:29:38	00:14:04	02:27:53	03:09:57	00:20:58
AVERAGE TIME			00:28:15			00:20:24