Software Product Line Engineering via Software Transplantation

LEANDRO OLIVERIA DE SOUZA, Federal Institute of Bahia, Brazil
EDUARDO SANTANA DE ALMEIDA, Institute of Computing (IC), Federal University of Bahia, Brazil
PAULO ANSELMO DA MOTA SILVEIRA NETO, Federal Rural University of Pernambuco, Brazil
EARL T. BARR, University College London, UK
JUSTYNA PETKE, University College London, UK

Software Product Lines (SPLs) improve time-to-market, enhance software quality, and reduce maintenance costs. Current SPL re-engineering practices are largely manual and require domain knowledge. Thus, adopting and, to a lesser extent, maintaining SPLs are expensive tasks, preventing many companies from enjoying their benefits. To address these challenges, we introduce Foundry, an approach utilizing software transplantation to reduce the manual effort of SPL adoption and maintenance. Foundry enables integrating features across different codebases, even codebases that are unaware that they are contributing features to a software product line. Each product produced by Foundry is pure code, without variability annotation, unlike feature flags, which eases variability management and reduces code bloat.

We realise Foundry in prodScalpel, a tool that transplants multiple organs (*i.e.*, a set of interesting features) from donor systems into an emergent product line for codebases written in C. Given tests and lightweight annotations identifying features and implantation points, prodScalpel automates feature extraction and integration. To evaluate its effectiveness, our evaluation compares feature transplantation using prodScalpel to the current state of practice: on our dataset, prodScalpel's use speeds up feature migration by an average of 4.8 times when compared to current practice.

Additional Key Words and Phrases: Software Product Lines, Software Transplantation, Genetic Improvement

ACM Reference Format:
Leandro Oliveria de Souza, Eduardo Santana de Almeida, Paulo Anselmo da Mota Silveira Neto, Earl T. Barr, and Justyna Petke. 2024. Software Product Line Engineering via Software Transplantation . 1, 1 (August 2024), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission [2]. The Software Product Line Engineering (SPLE) discipline provides a systematic methodology for producing and maintaining SPL from shared development assets [18, 35, 46]. For companies producing related products, adopting their codebase into SPL improves productivity

Authors' addresses: Leandro Oliveria de Souza, leandro.souza@ifba.edu.br, Federal Institute of Bahia, Irecê, Bahia, Brazil; Eduardo Santana de Almeida, eduardo.almeida@ufba.br, Institute of Computing (IC), Federal University of Bahia, Salvador, Bahia, Brazil; Paulo Anselmo da Mota Silveira Neto, paulo.motant@ufrpe.br, Federal Rural University of Pernambuco, Recife, Pernambuco, Brazil; Earl T. Barr, E.Barr@ucl.ac.uk, University College London, London, UK; Justyna Petke, j.petke@ucl.ac.uk, University College London, London, UK.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

and quality, speeds time to market, and reduces cost [5, 35], as it facilitates the reuse of development artifacts, such as code and design.

Despite its benefits, adopting SPL requires considerable upfront investment before its benefits can be realised [6, 11, 16]. The cost of migrating existing products to SPL is lower than adopting SPL from scratch, making *extractive* [27] adoption more common, especially in companies with many software system variants in production [7]. Two factors drive this preference: 1) it is often hard to know upfront that SPL 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 [11, 42].

To reengineer existing products, companies must solve four problems: They must analyse their products to 1) identify and 2) extract the features these products share, and 3) learn their interdependencies. Finally, they must 4) define a variability mechanism for combining these features, subject to their interdependency constraints [2]. Currently, reengineering to adopt SPL remains largely manual [2] and costly [10]. Indeed, because of its cost, software companies delay, or even refrain from, adopting SPL [18]. Automating these tasks remains an open challenge [2].

In 2013, Harman et al. [21] introduced software transplantation as a new research direction and laid out its implications for SPL reengineering. Harman *et al.* defined software transplantation as "*the adaptation of one system's behaviour or structure to incorporate a subset of the behaviour or structure of another*" [21]. In terms of automated software transplantation, Petke *et al.* [44, 45] were the pioneers in transplanting code snippets from different versions of a system to enhance its performance using genetic improvement [43]. A year later, Barr et al. [4] 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 Stelios Sidiroglou-Douskos et al. [54] , 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.

Inspired by this line of work, we introduce Foundry (Section 3), the first software transplantation approach for SPLE. Foundry is independent of the programming language, and supports SPL's *domain engineering* and *application engineering* [12] processes at the code level. It supports both reengineering and product assembly from previously extracted assets. Foundry successfully applies software transplantation to further automate both of these tasks.

Foundry does not eliminate the manual labour of feature identification, but reduces it to the task of annotating the entry points (*i.e.* 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; to do so, it uses slicing to overapproximate feature dependencies. It leverages transplantation to automate the variability mechanism and, simultaneously, tackle slice-imprecision. Key to Foundry is mapping software transplantation's "over-organs", conservative program slices [4], to product line assets, *i.e.* features. A product line via software transplantation is composed of multiple over-organs and a "product base", a host that contains all features that are shared across all products within the product line, so constructing a product entails transplanting a set of organs into a product base.

Foundry's use of slicing means that it does not need specially prepared donors. The donor programs can even be unaware that they are participating in an SPLE task. Indeed, Foundry permits a wholly new form of SPL, *symbiotic SPL*. In this form of SPL, a donor is oblivious to a parallel, ongoing SPL use of its codebase. It is Foundry's ability to transplant a feature given only a single annotation that makes this possible. Symbiotic SPL enables more than just one-off feature reuse: To capture feature improvements, a symbiotic SPL could periodically refresh features by re-transplanting them form an SPL-oblivious donor into its product base.

FOUNDRY may extract organs that 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. Therefore, FOUNDRY extends previous work on software transplantation [4] to handle multiorgan transplantation. FOUNDRY uses clone-aware genetic improvement [43] to combat bloat arising from slicing's overapproxition, specialise an over-organ to its implantation point, and to detect and remove cross organ redundancies (Section 3.4).

We realise FOUNDRY in PRODSCALPEL, a tool that transplants multiple organs (*i.e.*, a set of interesting features) from donor systems into an emergent product for codebases written in C. It can be used to make possible the extraction of features from donor systems, adapting them to integrate with the host product base, and ultimately, automating the generation of products. PRODSCALPEL also supports the use of existing variability mechanisms [19] based on *feature toggle* [48] or *preprocessor directives* [26]. 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.

To evaluate PRODSCALPEL, we conducted a controlled experiment (Section 4) and two case studies (Section 5). First, we conducted an experiment in which we asked twenty SPL experts to move a feature into a product line. We gave them the same inputs as those PRODSCALPEL requires, *i.e.*, annotated code and test cases. In all cases, using PRODSCALPEL sped up feature migration when compared with the time our study's participants took. On average, PRODSCALPEL took 18% of the time to transplant features from single systems to a product base than the participants who completed the task within the timeout. Next, we generated products by transplanting features from three real-world systems — Kilo[1], VI[2] and CFLOW[3] — into two product bases generated from VI and VIM[4], used as hosts for the target transplantations. This shows that our approach can transplant multiple features from unrelated codebases.

Our results (Section 4.5) and (Section 5.2) show that software transplantation speeds SPL reengineering, by combining features extracted from existing, possibly unrelated, systems.

The main contributions of this paper are:

(1) FOUNDRY (Section 3), a novel SPL reengineering approach that leverages software transplantation to extract and reuse features from existing codebases either to reengineer a code base into a product line or to assemble a product, even when those features and their codebases were not built for, or even aware of, the product line.
(2) FOUNDRY's realisation for C in PRODSCALPEL (Section 3.4), a tool that transplants multiple, multi-file organs in sequence and uses clone detection to prevent implanting redundant features.
(3) A rigorous evaluation of PRODSCALPEL that demonstrates FOUNDRY's promise. We show that PRODSCALPEL migrates features on average 4.8 faster than SPL experts performing the same task (Section 4). We also use PRODSCALPEL to generate two new products, composed of features transplanted from three different real-world codebases (Section 5).

All the source code and data needed to reproduce this work are available at the project webpage [55].

2 OVERVIEW

Many text editors have extended or re-written the original VI text editor. These programs share features, but, because they are separately developed, their implementations are often hard to reuse across the VI family of editors. Transforming VI, into a Software Product Line (SPL) would allow reusing its shared features across different text editors and accelerate

[1] https://github.com/antirez/kilo
[2] http://ex-vi.sourceforge.net/
[3] https://www.gnu.org/software/cflow/
[4] https://www.vim.org/

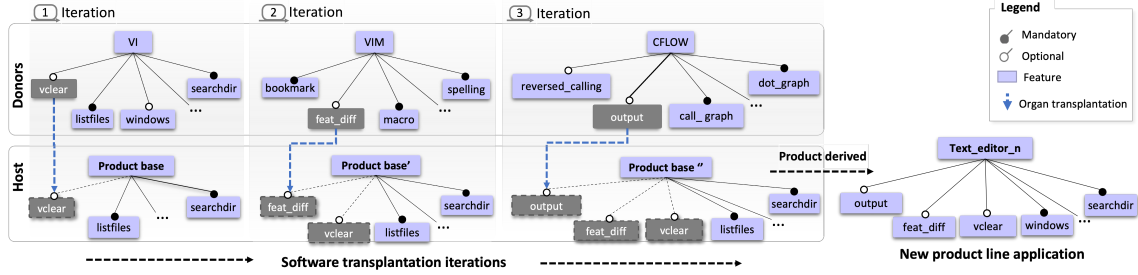

Fig. 1. Product derivation process using the Foundry approach. *prodScalpel transplants three features, in sequence, into the VI's product base to derive a new text editor.*

its development pace [46]. In short, the VI project is a natural candidate for software product line engineering (SPLE), if only the cost of re-engineering VI to adopt SPL were not prohibitive.

2.1 Current Product Assembly Practice

Suppose the VI community decided to build a product line for text editors. VI contains code from which one can extract an SPL product base, the shared substrate of a product line that hosts product-specific features. Although the exact steps of an SPL reengineering process vary in the literature [2], we present the main tasks on the VI example.

First, the VI development team would have to identify all the existing features, and map these to their implementation. This requires identifying mandatory features that occur in all products, which will form a *product base* and optional features that could be added to any product, as desired. In our VI example, suppose `listfiles` and `searchdir` compose a product base, while `vclear` and `windows` are optional features (see top-left tree in Figure 1). Although there have been attempts to automate the tasks of product base and feature identification [33, 51], they generally lack in accuracy and require substantial effort to adapt. Thus, the current state-of-the-art is still a largely manual task [29] that requires substantial domain knowledge.

Next, the VI team would have to analyse their portfolio to propose a variability model, which defines valid combinations of features. This is most commonly done using a tree-like structure called a feature model. This step requires expert domain knowledge. In our example the resultant feature model is presented in the top-left tree in Figure 1. Depending on the size of the model, it could be created directly or with support from tools such as ArborCraft [59] that produce a candidate model from natural language requirements.

Finally, the VI team would have to transform the artefacts to obtain the SPL. This requires creating an implementation that allows for insertion of optional features in a seamless manner, *e.g.*, using IFDEF flags. In our example, we would have to re-write the code for our product base, *i.e.*, features `listfiles` and `searchdir`, and insert an IFDEF flag with the appropriately amended code for `vclear` and `windows`. Given a code-feature mapping identified in the first phase, extraction requires simply copying the code. The adaptation task is non-trivial as the code needs to be executable outside of its original environment. Moreover, certain functions might be common among multiple features. There is little work in automating the adaptation of extracted code [2]. The most recent example reported by Assuncao *et al.* [2] is the ECCO tool [17, 18]. Given the database of features for a product family, ECCO only identifies which software artifacts may need to be migrated and adapted, leaving it to a developer to confirm the suggestion and manually perform the actual migration and adaptation.

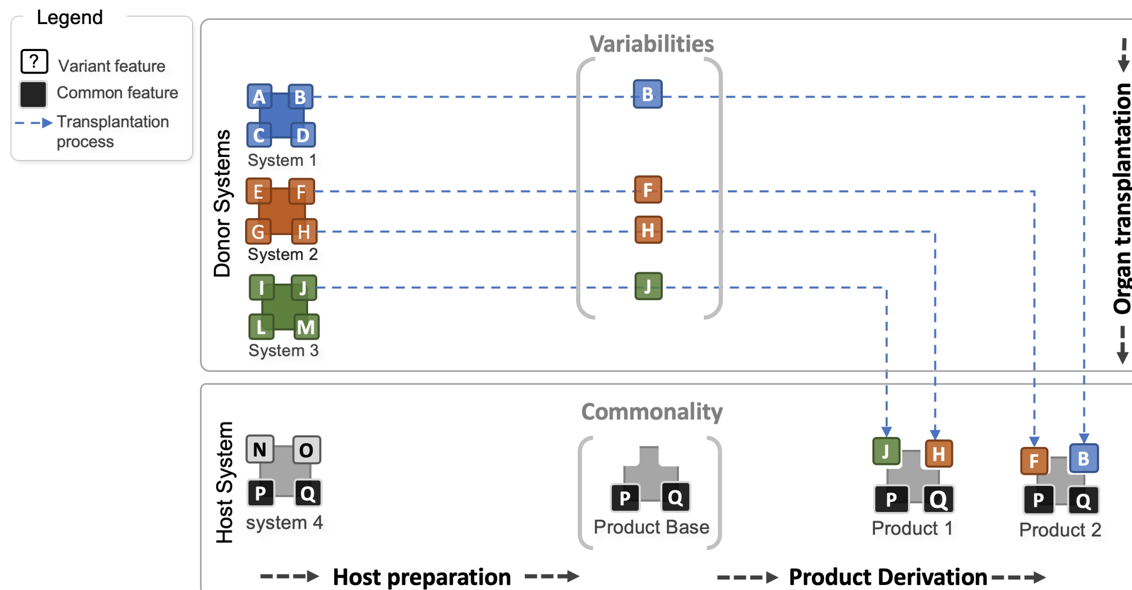

Fig. 2. An overview of how new products are derived from a product line based on ST.*Four over-organs (A, D, G, L) are extracted from three donor systems and kept in the transplantation platform, with product base consisting of 2 features shared across all products (P and Q).*

2.2 Assembling Products via Software Transplantation

We introduce FOUNDRY as a new approach to SPLE. Figure 2 illustrates the concept under discussion. In the first part of the figure, we identify three donor systems from which features are transplanted into a codebase. The representation of features B, F, H, and J serves to demonstrate the inherent variability in the product line. These features are extracted into the product line and subsequently implanted into specific products according to their requirements.

In the second part of the figure, we have a product base generated from a host system. The features N and O in the host system symbolize elements that need to be removed during the transplantation process due to their irrelevance to the target product line or the resulting products. Features P and Q exemplify the commonality within the product line, as they will be kept in the product base and shared among multiple products. As a result, we have a product line comprising features B, F, H, J, P, and Q, enabling the generation of diverse products through the software transplantation process.

FOUNDRY thus has two modes of operation: it extracts features for later transplantations, keeping them in an ice-box environment, which we denote as transplantation platform; second, it directly produces products by transplanting and adapting features from the transplantation platform into the target product base.

In contrast to the most common SPLE practices, FOUNDRY presents a new approach to variability management. By leveraging software transplantation, FOUNDRY enables the integration of features across different environments. This not only reduces the complexity and effort required in variability management but also minimizes the code bloat caused by the use of, *e.g.*, feature flags.

By easing SPL adoption, we hope Foundry will be transformative. We believe that a sufficient quantitative difference can become qualitative[5]. In this spirit, we have designed Foundry to make extracting and transplanting features so easy that its adoption will allow an SPL product base to extract and incorporate features from projects not explicitly engineered for SPL, like open source projects, that are oblivious to the use of their features in an SPL.

To provide more detail, we come back to our VI example. To extract the product base, using Foundry-based SPL reengineering, the VI team would first need to annotate some of VI's functions as features, then repeatedly apply prodScalpel to remove each feature. Foundry assumes that features correspond to functions, so annotating a feature consists solely of adding `__FOUNDRY Feature <NAME>` to a function's header comment. `<NAME>` is replaced with the feature's name. The unannotated code, which remains after applying prodScalpel to an annotated donor, then forms the starting product base for text editor products. This product base then serves as the host for transplanted features. Note that the previously largely manual task of identifying all code for a given feature boils down to feature location annotation - prodScalpel automates feature extraction.

After creating an editor product base from VI, suppose the development team targets building editor products with different combinations of three features: (1) `vclear` from VI; (2) `feat_diff` from VIM text editor; and (3) `output`, a feature in Cflow, a C call graph extractor. The Cflow example was chosen to exemplify the case where the donor comes from a different software family than the product base. Foundry provides this feature because it reduces the cost of identifying a feature to annotating a function.

The extraction of a target feature, even after it has been identified, can require substantial effort. A feature can, for example, involve a considerable amount of code at different levels of granularity, and require more than just moving functions and individual statements, up to including entire files and libraries [58]. For instance, VIM's `feat_diff` feature has more than 5k LOCs scattered across 33 of VIM's 166 source files. Given annotated donors and an annotated host together with a set of test cases (adapted from the donor to reach the implantation point in host) Foundry completely automates extraction and adaption via transplantation.

In the product base formed from stripping features from VI, the engineers must annotate the implantation points for each target feature, using the annotation `__FOUNDRY Implantation <Name>` where name can be used to differentiate different features, so a host/product based can be prepped via annotation once to accept multiple organs and produce multiple different products. Usually, these points correspond to calls to the function that implemented the feature (*i.e.* the organ) in VI. The engineers then run prodScalpel on these inputs with

```
prodScalpel -d donor_list -h host
```

where `donor_list` is a list of annotated donors and `host` is a product base. This command automatically extracts all of each feature's source code and its dependencies from each annotated donor, then transplants the features into the specified host/product base.

SPL aims at easing software maintenance, evolution, and customisation. As our evaluation shows, Foundry significantly speeds extracting and moving features — speeding SPL reengineering and product assembly — with the aim of unlocking the benefits of SPL for more development shops.

[5]We have adapted this observation from the saying "Quantity has a quality all its own.", whose attribution, at the time of this writing, is discussed in this blog.

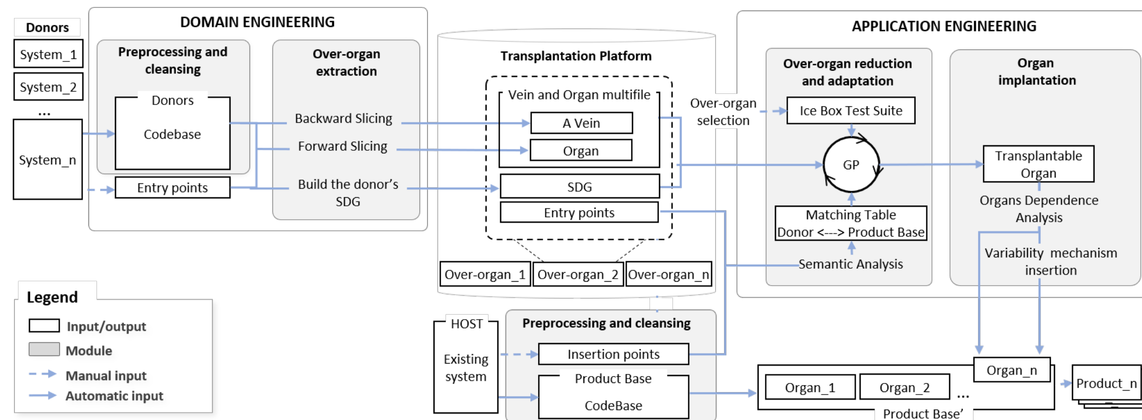

Fig. 3. Overall architecture of prodScalpel. *An SDG is a system dependency graph used in the genetic programming phase.*

3 FOUNDRY

Foundry is a methodology for extracting, adapting, and combining features from a set of code base sources to either reengineer a code base into a product line or to construct a product for a Foundry-based SPL. Foundry is independent of the programming language, and supports SPL's *domain engineering* and *application engineering* [12] processes at the code level.

Figure 3 shows Foundry's key processes and artefacts. Top left, Foundry's domain engineering for preps both donors and the host via preprocessing, slices them to extract over-organs (overapproximate features) and establish a product base, then stores over-organs to create a transplantation platform (Section 3.2). Foundry's application engineering stage uses genetic programming to reduce and adapt over-organs to their implantation point. In this setting, implantation must employ a clone detector to avoid adding clones (Section 3.3). We close this section by discussing challenges overcome to build prodScalpel, tooling that implements Foundry for C (Section 3.4).

3.1 Terminology

Software transplantation, as formulated by Barr *et al.* [4], specifies a methodology for automating copying code from a donor code base to a target host code base. An *organ* is the unique code needed to implement some functionality of interest. In SPL terms, an organ is a feature.

Software transplantation approximates an organ by starting from an annotated in the donor. From the annotation, software transplantation forward-slices to over-approximate the organ forming an *over-organ* and backward-slices to over-approximate a *vein*, donor code needed to initialise the over-organ's execution environment. Over-organs are executable. If an over-organ is extracted for either to be transplanted later or into multiple hosts, it is stored in an *ice box*, along with the over-organ's test suite, and its vein.

In SPL, a *product base* implements all functionality shared across a product line. In Foundry, the product base is the transplantation target: it hosts all the organs transplanted into it to form a product. Foundry establishes a product base by repeatedly extracted product-specific features from a suitable donor, a program that implements many of an target product line's shared functionalities. Foundry supports product variability via transplantation. Foundry treats a

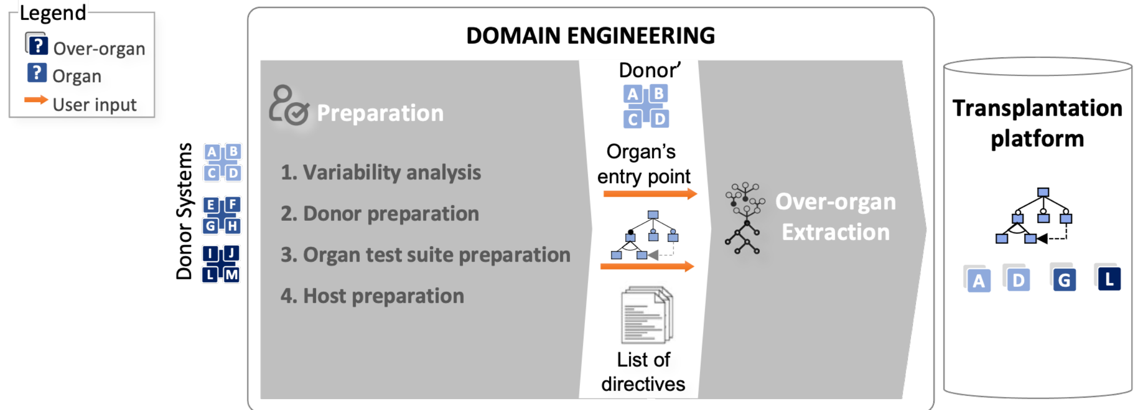

Fig. 4. Domain engineering process supported by FOUNDRY.

product base and a set of ice boxes, which combine over-organ, an organ-specific test suite and a vein, as product line assets.

3.2 Domain Engineering

In SPL, *domain engineering* establishes a reusable platform of core assets, shared by all products built on top of a product base [12]. It also specifies the permitted variations of these assets, *aka* products. Figure 4 illustrates how Foundry supports domain engineering in two phases: prepping the donors and the host, then organ extraction.

Prepping a system for Foundry starts with variability analysis using a feature model [25], augmented with icons depicting Foundry's organs (features). Each organ is annotated with its corresponding entry point in the donor codebase. Identifying organs is manual: An SPL engineer must identify an organ's entry point, the root-level function definition that implements it. Here, Foundry's main advantage is that it does not require an engineer to fully identify the organ and its dependencies, as is the case in most related work. This reduces the domain expertise required to adopt Foundry-based SPL.

After variability analysis, three tasks remain. The first task is to prepare the donor by applying a program transformation to make the organs amenable to software transplantation's genetic programming (GP) phase. Specifically, this phase uses search to reduce and adapt the organ when implanting that organ into a host. When Foundry is instantiated for C, for instance, this transformation handles IFDEFs, whose ability to conditionally combine several different programs into a single source file complicate search. Section 3.4 details how prodScalpel, Foundry's realisation for C, handles IFDEFs. Dead code elimination is another example. While software transplantation's GP effectively implements a form of observational slicing to automatically eliminate dead code, removing dead code upfront can greatly increase the speed and reduce the cost of this search.

The second task requires an SPL engineer to identify or supply test suites, called *ice-box tests* [4]. These test suites guide GP during over-organ adaptation to implant an organ into the product base that is fully executable. Ice-box tests can be extracted from donor tests that traverse the organ or produced using automated test generation [47].

The final task, host preparation, starts with identifying a suitable candidate host. To do so, an SPL engineer selects a set of products from the same domain and creates a feature model for each product. During this analysis, the engineer

identifies common features across these products. The goal is to find a system that possesses the most common features from this group, since it can serve as a strong foundation for assembling products in the product line [8, 13, 53]. For instance, a text editing program might be good product base/host for new products that enhance editing with features like find, format, spellcheck, or a more exotic feature like translation.

As this discussion makes clear, while FOUNDRY aims to reduce the cost of reengineering and product assembly, prepping a code base for FOUNDRY still requires substantial manual effort. Two of these four tasks — variability analysis and identifying and annotating the host — are entirely manual. Nonetheless, these two tasks are one-off and will be amortised across multiple transplantations and/or product assemblies. In contrast, the remaining two tasks — applying a program transformation to the donor and generating ice-box tests — can, in some cases, be entirely automated.

Over-organ Extraction. Following Barr *et al.*'s μTrans, FOUNDRY takes an entry point in the donor annotated by an SPL engineer and conservatively slices it to automatically extract a feature into an over-organ. Given an organ entry point, FOUNDRY follows μTrans and produces an over-organ from a forward slice and a vein from a *backward* slice, both starting from the organ's entry point. These slices are conservative, so they over-approximate the code that an organ and its vein actually require. Like μTrans, FOUNDRY relies on GP to remove any bloat that slicing may have introduced.

Adapting over-organ extraction to SPL, as FOUNDRY does, poses a new challenge: preserving an organ's structure, not just its semantics. Failing to preserve structure impedes propagating changes made to the organ (feature) in the donor [58], like bug fixes, enhancements, *etc.*, to the products that host that organ. FOUNDRY's setting exacerbates this problem in two ways: first, multifile organs are common in SPL and, second, preserving structure underpins FOUNDRY's ability to include features from donors outside of the FOUNDRY SPL that is using them. Thus, FOUNDRY requires interprocedural slicing.

After extraction, FOUNDRY stores the source code of each over-organ in an transplantation platform. Collectively, these over-organs, their ice-box tests, and product base comprise a FOUNDRY product line.

3.3 Application Engineering

In SPL, features are assembled into a product during the *application engineering* phase. FOUNDRY transplants organs into a product base, or host, to assemble features. FOUNDRY offers two ways to create products. When transplanting an over-organ into a product base, FOUNDRY can acquire a target over-organ either from a pre-existing *transplantation platform*, *i.e.* a repository of transplantation assets, or it can directly extract and transplant an organ from an SPL-oblivious donor. This first way is standard practice in SPL. The second way is unique to FOUNDRY. Those who adopt FOUNDRY can mix and match these two ways to create products that combine native and foreign features.

As illustrated in Figure 5, FOUNDRY assembles and validates a product via software transplantation in four stages: (i) over-organ selection, (ii) over-organ reduction and adaptation, (iii) organ implantation and (iv) postoperative stage.

In the first step, an SPL engineer refers to the feature model built during variability analysis to select over-organs for a target product. Following Barr *et al.*, over-organ reduction and adaptation specialise the organ to the host environment, *i.e.* the product base. An SPL engineer must specialise the product base with product-specific annotations that specify where organs should be implanted.

Almost all products combine multiple features. Thus, any transplantation-based SPL approach, like FOUNDRY, must support transplanting multiple features into a host in any order[6]. Multiple features can, of course, interact and introduce dependencies, complicating reuse [49]. FOUNDRY assumes that features interact either through their

[6]We note that Barr *et al.* did not address issues arising from a sequence of transplantations in their prototype.

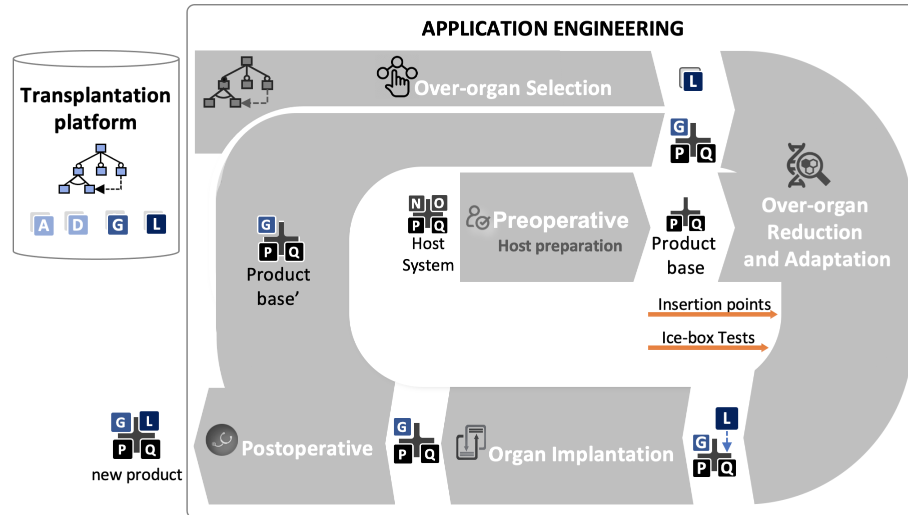

Fig. 5. Application engineering process supported by FOUNDRY. *A new product is derived after two software transplantations: organs G and L.*

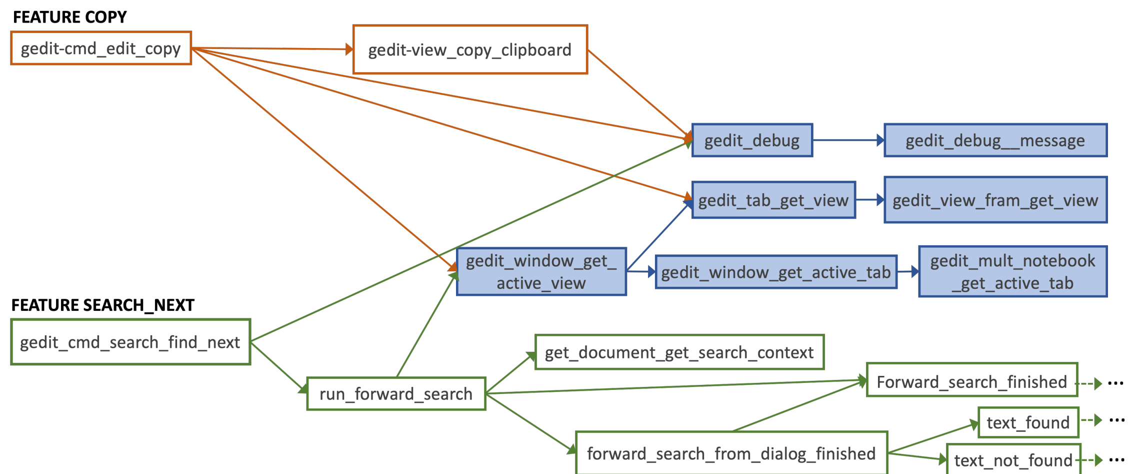

Fig. 6. A call graph extracted from GEdit text editor. *An example of connection points among call graphs from organs* `copy` *and* `search_next`. *Highlighted with blue boxes are functions belonging to both organs.*

interface, via function calls, since Foundry maps features to functions, or through shared data structures. When using software transplantation, shared data structures appear in the vein of an over-organ, the code that initialises a feature's environment prior to a feature's first invocation.

Figure 6 illustrates the problem. It depicts two call graphs in GEdit that share several functions. If we consider the code that these calls graphs traverse to be parts of two unrelated organs, naïve transplantation would duplicate the shared functions, potentially violating correctness.

To solve this problem, FOUNDRY must recognise and preserve these dependencies. Slicing captures all the invocation interactions in each feature; shared data structures, for instance, cannot simply be naïvely copied over to the host because doing so creates duplicates and violates the sharing requirement. To solve this problem, FOUNDRY incorporates clone detection into software transplantation.

3.4 Implementation

PRODSCALPEL, our realisation of FOUNDRY for C, extends μTrans [4] to handle multi-file organs, tame the C preprocessor (Section 3.4), and construct a wrapper to initialise any newly imported data structures and to map host variables to organ variables, then inject it into the host. PRODSCALPEL also transplants multiple organs, in succession, and implants them into a single host (Section 3.4). This feature is essential for supporting FOUNDRY, since creating a product involves transplanting many features. Like μTrans, PRODSCALPEL is written in C and TXL [14].

Taming the C Preprocessor. The C preprocessor is an (in)famous source of challenges for program understanding, analysis, and transformation [40, 56]. It is also a mechanism for supporting SPL, see Section 7.3. As a result, IFDEFs can generate dead code never used in any FOUNDRY product [56]. Further, raw C code that contains preprocessor directives, especially IFDEFs, does not compile until the preprocessor has run and removed them. FOUNDRY's GP phase repeatedly executes tests to debloat over-organs and adapt them to their host, so its search runs faster on cooked C source, *i.e.*, C source after preprocessing. PRODSCALPEL speeds search by producing cooked, directive-free, C source for each feature, which, for FOUNDRY, starts from an annotated function definition. When operating in this mode, PRODSCALPEL takes a list of IFDEF flags to remove, along with their contents.

Mutually Recursive Functions. Barr *et al.*'s μTrans relies on inlining to eliminate recursion in organs. Unfortunately, mutually recursive functions are not uncommon in C codebases. Indeed, we encountered them in the VI and VIM codebases that we use to evaluate PRODSCALPEL. Removing recursion via inlining diverges on mutually recursive functions. To combat this problem, PRODSCALPEL pushes each function that it inlines into a stack that it uses to detect recursive calls, even indirect ones. When PRODSCALPEL encounters a recursive function, it does not inline it. Instead, PRODSCALPEL writes the function to a file. Then, it redirects the recursive call in the vein to the function defined in the file.

Implanting Multiple Organs. As first formulated, software translation considered only transplanting a single organ into a host. An over-organ consists of a vein, all the code that builds the data structures on which a feature depends, and its organ, the code unique to the feature itself. Backward slicing from a feature-annotated function extracts a feature's vein; forward slicing creates its organ. Transplanting multiple over-organs in succession creates a problem: the veins of the over-organs might share code implementation data structures or other functionality. When this happens, naïvely implanting these over-organs will introduce clones in the host. These clones may do worse than introduce bloat: for instance, the organs may communicate through the shared functionality/data structure, so cloning them could compromise correctness.

To solve this problem, PRODSCALPEL must avoid implanting code clones. Figure 7 illustrates our solution. To sum up, PRODSCALPEL incorporates a clone detector, based on NiCad [50]. When implanting a vein, PRODSCALPEL checks for clones in the vein and the host. If it detects a clone, it rewrites the organ it is implanting to use the host's instance of the clone.

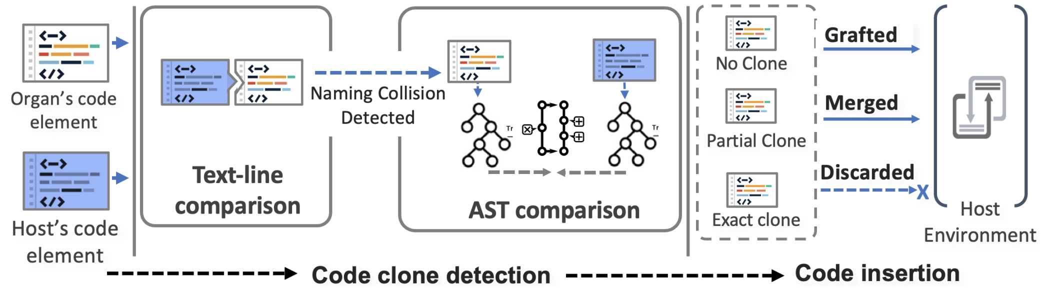

Fig. 7. Code clone detector. *It performs two comparison steps: text-line and ASTs comparisons to identify code clones and make decision if a code element must be grafted or not into the target product base environment.*

4 EVALUATION

To evaluate Foundry, we first compare our approach with current state-of-the-art. Unfortunately, tool support for the reengineering process is limited, providing support for specific activities, such as feature location, refactoring or quality assurance [2, 30]. To the best of our knowledge, there is currently no comparable tool that manages to extract, adapt and insert a feature from one codebase to another in an automated manner. Therefore, we compare our approach with the manual process and ask the following research questions:

> **RQ1.** *How often does prodScalpel migrate a feature without breaking a test suite when compared to the standard, largely manual, process?*

Having established whether or not prodScalpel preserves semantics modulo a test suite when moving a feature, we want to know whether prodScalpel improves upon the current process, thus we ask:

> **RQ2.** *How much time does it take for prodScalpel to migrate a feature compared to the standard process?*

With this question, we evaluate the time spent by SPL experts to *extract*, *adapt* and *merge* features to derive new product variants and compare these with the time taken for the same tasks when using prodScalpel.

To answer our research questions, we conducted an experiment that reflects a real-world process of feature migration from existing codebases [28], described next. Our corpus and data collected are available at the project's webpage [55].

4.1 Methodology

We use two donor systems in our experiment: *NEATVI*[7], text editor extended from VI for editing bidirectional UTF-8 text and *Mytar*[8], an archive manager. We use NEATVI as the host, from which we extract two product bases. Table 1 gives more details about the systems used in this experiment. Having in mind manually inspecting a codebase to transfer a feature to a product line is hard, slow, and tedious [38], we chose to select small systems to avoid participants getting tired. These codebases were available to download together with a script to automate setup of the environment.

We recruited 20 SPL experts for the experiment that were divided into 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. Guidelines provided to the participants of *Group A* and *Group B* are available at [55]. It consisted of a process description and systems documentation. Additionally, we provided training on reengineering software systems

[7]https://github.com/aligrudi/neatvi
[8]https://github.com/spektom/mytar

Table 1. Details of donors and product base systems used in our study. NEATVI^{-} represents the original NEATVI system with the `dir_init` feature removed.

Scenario	Donors	LoC	Target features	LoC	Host	LoC
I	NEATVI^{-}	6,536	`dir_init`	239	NEATVI	6,645
II	Mytar	1,046	`write_archive`	170		

to SPL, in particular the extraction, adaptation, and implantation tasks, to ensure that all participants understood the experiment's objectives.

We used timesheets to record the effort spent on the necessary activities required to transfer features from an existing codebase to a product base, as previously mentioned. In addition to the timesheets, we used two forms to collect information about participants' experience. This data is shown in Table 2. We also conducted a post-survey to better understand participants' problems encountered.

We simulate a real reengineering process where two features must be transferred to a product base. The experimental design was inspired by documented real product-line migration scenarios [2, 33].

In Scenario I, we gathered a group of 10 SPL experts (called Group A) where each one of them had to re-transplant all portions of code that implement the feature `dir_init` to the product base. We removed this feature from the original version of *NEATVI* to generate the product base used in this scenario. In Scenario II, another group of 10 SPL experts (called Group B) tried to insert the feature `write_archive` from *MYTAR* into the original version of NEATVI used as the product base. Here, we chose not to provide the post-operative product base used in the Scenario I. Instead, we provide the original version of NEATVI, already with the feature `dir_init`, to prevent possible code errors introduced in Scenario I. As we analyse both scenarios separately, we believe that this strategy has minimized possible human bias.

The idea is to use each scenario to represent real scenarios of migrating features to a product base, where system variants came from both similar (Scenario I) and distinct codebases (an archive manager providing features to a text editor - Scenario II) as Figure 8 shows.

In both scenarios participants are given the same inputs as prodScalpel. Thus the independent variable in our experiment is whether the feature migration process is automated or not.

In this experiment, the dependent variables are: success of the product generation process and the payoff for using our approach. That is, to analyze the success of our approach (RQ1), we evaluate how often the transplanted features pass all the provided test cases. To analyze the performance of the approach (RQ2), we measured the time spent by participants to extract, adapt and merge one feature into a product base in comparison with prodScalpel's time to complete the same tasks.

4.2 Participants

We recruited 20 participants: 2 undergrads (Un), 9 masters (M), 7 PhD. (PhD), and 2 Post-Ph.D. (Post-PhD). Most of them have more than 5 years of SPL experience and 10 years of software development. The participants are from ten different universities (from U1to U10) and the analysts/developers work in four different companies (C1, C2, C3, C4 and C5). To recruit them, we sent emails to professors from two universities, from different software reuse research groups, to suggest current and ex-members.

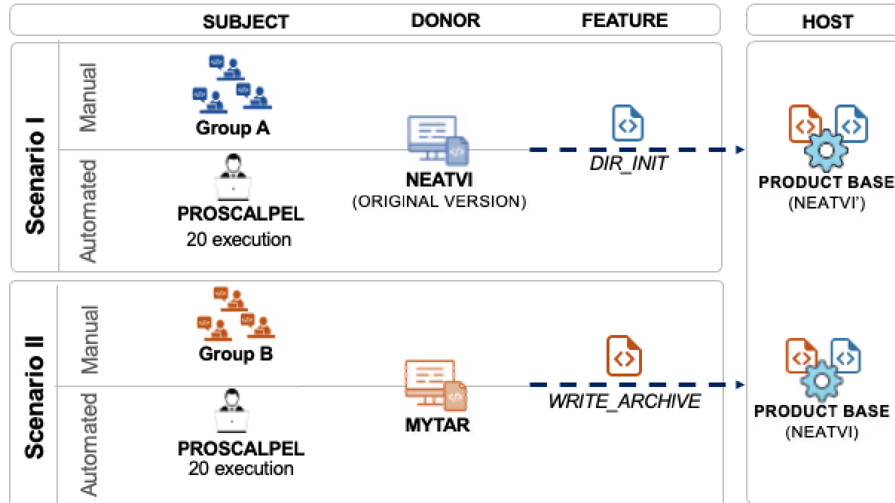

Fig. 8. Experimental design: one factor with two treatments applied in two feature migration scenarios.

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[9]. We created balanced groups (A and B) of participants for each ifeature migration scenario, based on their experience. Table 2 shows the details of the participants involved in the experiment.

4.3 Operation

First, we conducted a pilot study with 6 graduate students, for the two aforementioned scenarios. None of them participated in the main study to avoid learning bias. The pilot study allowed us to assess whether the participants could properly understand the subject systems and the tasks they should perform. We used the results to determine the final setup, *e.g.*, the amount of time needed to execute our tasks, and the instructions provided.

Before the participants receive their tasks, we introduced the experiment with a *tutorial* on reengineering of existing systems into SPL, with special focus on feature migration. 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, the donor's source code, and a prepared product base with the target insertion point. We also supplied the participants with ice-box tests that could be used to guide the search for organ code modifications required to check if it continues to be executable when deployed in the host. 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.

The direct costs of this experiment are related solely to the time spent by the researcher with the 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 for Scenario I and 17 minutes for Scenario II; creating the test cases, necessary to validate the target features, taking approximately 16 minutes of programming activities;

[9] https://rb.gy/ant4g8

Table 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)

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 emergent product with a new feature transferred from the correspondent donor system, all participants were required to perform three activities [27, 38]: feature *extraction*, *adaptation* and *merging*, with descriptions and instructions provided for each task.

Initially, the participants had to identify and extract all code associated with the feature of interest to a temporary directory. Then, each participant had to adapt the extracted feature so it executes correctly in the product base environment, passing all unit tests. In practice, each participant had to change the feature source code to be compatible with the name space and context of its target insertion point in the product base. Finally, they had to insert the feature's code into the product base, and validate its correctness via regression testing.

4.4 Data Collection & Analysis

We have provided a task and time registration worksheet. While participants were conducting the assigned tasks, we asked them to take notes of which strategies were being used for each stage of the feature 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. This way we can better understand participants' problems and differences between the manual and automated process in both scenarios. We have triangulated the data generated from the experiment with the responses we obtain from the pre- and post-surveys.

To establish the time for feature transplantation using our automated approach, we ran prodScalpel 20 times, and measure the average time spent on feature migration in each scenario. This average time was compared with the time spent by our participants on the same task.

Based on our pilot study, we set a time limit of 4 hours for each manual and automated process. This is to allow for enough time to transfer required amount of LoC while avoiding participants getting bored or tired.

We used 22 pre-existing regression tests designed by the *NEATVI* developers to assess the success of prodScalpel and manual transplantations. However, these 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 migration process. To achieve better code coverage, we manually augmented the host's regression test suites with additional tests, our augmented regression suites. Furthermore, we implemented an acceptance test suite for evaluating the transferred feature in both Scenario I and II. We have a total of 30 such tests in Scenario I and 33 in Scenario II. Our test suites provided statement coverage of 72.5% to the post-operative product line in Scenario I and 73.3% to the post-operative product line in Scenario II.

4.5 Results and Discussion

We summarise our results in Table 3. We report the status of the product base and feature inserted by the participants, the time spent and the number of passing tests for the regression augmented regression and acceptance test suites. The time measured for the participants is available at [55]. 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 report the number of prodScalpel runs in which the product derived passed all test cases. For each scenario, we repeat each run 20 times. The success rate was retained for both Scenarios I and II, where only one run timed out and the product generated passed all tests from all test suites.

The results show success rate was retained in both Scenario I and II, where we lost only one successful run in the timeout and all products derived passed all tests from all test suites. With regards to the manual process, nine participants have successfully transplanted the target feature in Scenario I. In Scenario II, seven of ten participants broke the product base when trying to transplant the target feature, while half of the participants were not able to transplant the feature within the timeout.

RQ1 Answer: Our results show that use of prodScalpel helps complete feature migration within the time limit more frequently than using the standard process, with only one run timing out. Eight of twenty participants (considering both scenarios) were not able to transplant the target features without breaking tests.

To answer **RQ2**, we evaluate the payoff of prodScalpel. Figure 9 graphically shows the time spent on each activity performed in the reengineering to SPL process. Overall, Group A transferred the target feature from *NETVI* to the product base in 1h24 minutes on average. Use of prodScalpel turned out to be quicker, successfully transplanting this feature in all 20 trials, taking an average of 20 minutes.

Table 3. **Scenario I:** Donor: NEATVI - Product Base: NEATVI without the desired feature. *Experiment results comparing the time of* prodScalpel *over 20 repetitions with the participants: column product status shows the generated product status by participants and the tool; column* Execution Time *shows the time spent on the feature transplantation by the participants and the average time of 20 runs of* prodScalpel. *We highlight the execution time of the participant that most quickly completed the task. Columns in* Test Suites *show results for each test suite and report statement coverage (%) for the postoperative host and for the organ. Columns marked with* PASSED *report the number of passing tests.*

Participants	Product Status	Execution Time (minutes)	Test Suites		
			Regre. (59%) PASSED	Regre.++ (70.1%) PASSED	Accept. (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 4. **Scenario II:** Donor: Mytar - Product Base: NEATVI. All other columns are the same as in the previous table.

Participants	Product Status	Execution Time (minutes)	Test Suites		
			Regre. (70.1%) PASSED	Regre.++ (71.9%) PASSED	Accept. (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**

Most of the participants from Group B had not completed the feature migration 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*) successfully (all tests passed) they spent an average of 2h23 minutes while prodScalpel was able to complete this task in 19 of 20 trials in the timeout, taking 27 minutes on average.

By considering the time spent in both scenarios, the tool accomplished the product generation process 4.8 times faster than the mean time taken by participants who were able to finish the experiment within the timeout.

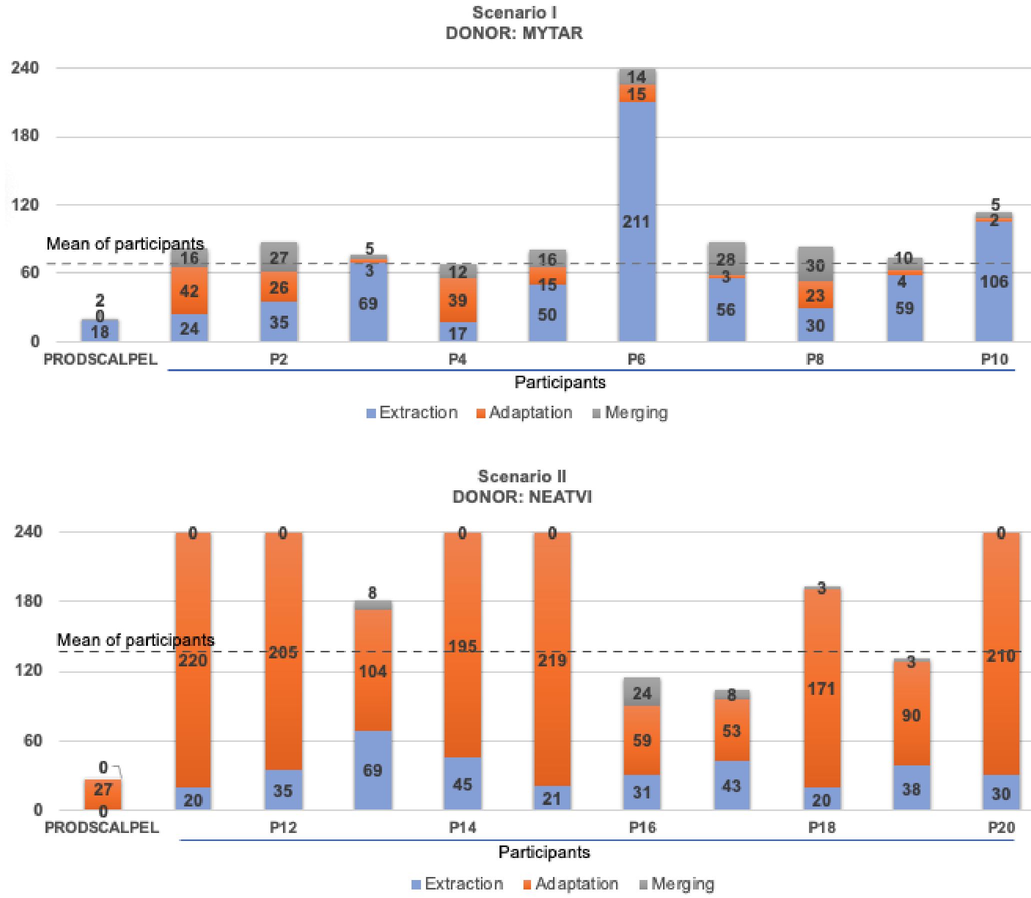

Fig. 9. Time (in minutes) spent by participants and PRODSCALPEL on performing the three stages of SPL reengineering: feature *extraction*, *adaption* and *merging*. The graph highlights the average time spent by participants who successfully generated new products.

To establish statistical significance of our results we first establish statistical distribution of runtimes for each scenario through the Shapiro-Wilk [52] test. Next, we use ANOVA [20] and Pairwise Student's t-test analysis to test the hypothesis that PRODSCALPEL's runtimes are statistically lower when compared with the times required by the participants to complete the same task (p-value < 2e-16). We rejected the null hypothesis for all pairs. Figure 10 graphically shows the time results for our two groups in comparison with PRODSCALPEL's performance. In Scenario I, the preliminary information provided by the box plots indicates that all samples are normally distributed (W = 0.70445, p-value = 3.129e-05).

In Scenario II, the normality test result showed a normal distribution with a W = 0.69378, p-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-value < 2e-16) that the distribution of the population is homogeneous.

We can conclude that PRODSCALPEL reduces developer effort to transfer features to a product base in both scenarios. For both simulation scenarios, there is a significant effect size between the tasks performed in a manual way and using

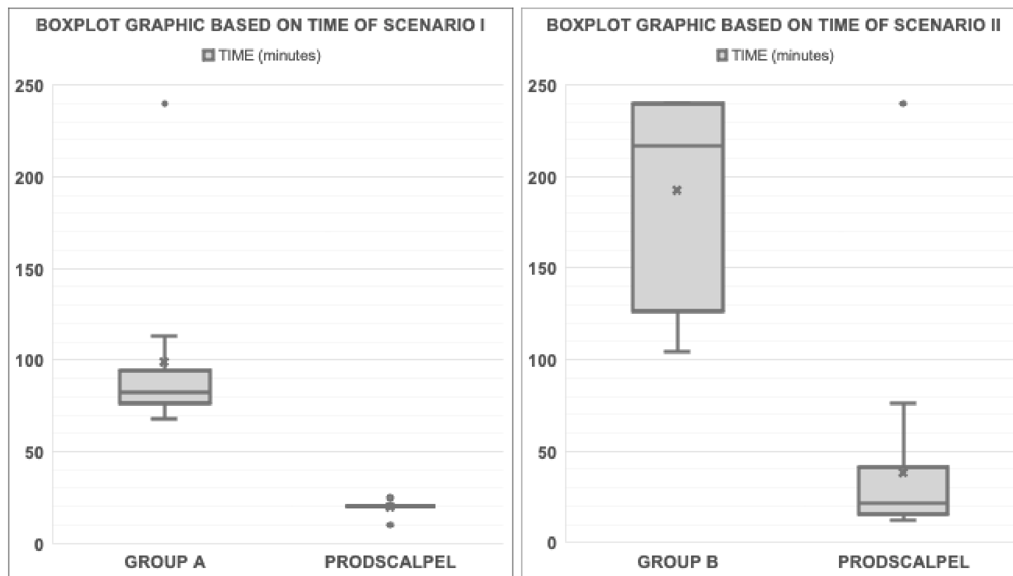

Fig. 10. Time results grouping automated and manual in both scenarios. Scenario I: *NEATVI* - Product Base; Scenario II: *Mytar* - Product base.

prodScalpel. Our participants had similar performance times in Scenario I, with the exception of participant *P6*. On the other hand, most of the participants of Scenario II 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 mentioned it is hard to adapt a feature from one codebase to run in a different codebase.

RQ2 Answer: The results show that use of prodScalpel for transplantation of two features is 4.8 times faster, on average, than the time taken by participants who finished the same tasks within a timeout.

5 CASE STUDIES

In the previous section, we showed feasability of Foundry and how it improves upon state-of-the-art. Here, we present two case studies to showcase other features, *e.g.* multi-feature transplantation, and limitations of prodScalpel. Our corpus and data collected are available at the project's webpage [55].

5.1 Methodology

We chose two text editors, *VI* and *VIM*, as product bases. For donors we chose subjects from two different domains: code analysis and text editors. We chose *GNU Cflow*, a call graph extractor from C source code, as a donor. This is to show that donors can come from different application domains. We reuse *VI* and *VIM* as donors, as well as another text editor, *kilo*. This is to show that donors can come from the same system as the product base, but also different systems within the same application domain. 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. Table 5 presents more details on the subject systems.

Table 5. Donors and hosts corpus for the evaluation: column Features shows the number of features identified.

Subjects	Type	Size (LOC)	#Features	#Functions
Kilo	Donor	804	17	42
CFLOW	Donor	4,274	54	227
VI	Donor	20,292	36	582
VIM	Donor	839,438	176	7,077
VI	Host	20,223	35	563
VIM	Host	737,466	117	9,167

Initially, we automatically remove dead-code from both donor and host codebases by using prodScalpel. We also reduced each host to its basic form removing all optional features. Thus, both donors and hosts are prepared for the transplantation process. Given an organs' entry point for each organ in the donor systems provided by us, their target implantation points in the product base, and a set of test suites for each organ, prodScalpel was used to localise and extract a set of organs from the donor, transform each organ to be compatible with the context of their target sites in the product base and implant the organs in the beneficiary's environment. Each automated organ transplantation process was repeated 20 times, due to the heuristic nature of the over-organ adaptation process. We computed the average number of lines of code transplanted and the average runtimes for the transplantation process.

The runtimes for each transplantation were measured on an Intel Core 3.1 GHz Dual-Core Intel Core i5, with 16 GB memory running MacOS 10.15.4.

5.2 Results and Discussion

In the transplantation of features from two different versions of the same system — one with the desired feature as the donor and another without it as the host — prodScalpel replicated the original code, incorporating the transplanted feature. The introduction of the organ-host wrapper, a necessary addition, did not compromise the overall integrity or functionality of the codebase. To thoroughly validate this integration, acceptance tests were conducted on the resulting code. These tests affirmed the successful assimilation of the transplanted organ into the original codebase.

Table 6 shows average runtimes of transplanting each organ into the product base as well as the total number of code lines transplanted to derive each product. At the end of the transplantation process, the postoperative Product A (with VI as the product base) has a total of 28k LoC and 40 features while Product B (with VIM as the product base) has a total of 745k LOC and 121 features. Together, donors provided three features to each of the emergent products, approximately 7.8k LOC to Product A and 8.1k to Product B, including a feature removed and re-transplanted in *VI*.

On average, prodScalpel spent 4h31min/1KLoC for transplanting three features into VI, and 4h40min/1KLOC for transplanting the same three features into VIM.

We also tried to transplant two more features: `spell_check` and `search`, both containing larger amounts of code, about 104 KLOCs and 153 KLOCs, respectively. These experiments proved unsuccessful.

prodScalpel uses Doxygen [57], a source code documentation generator, to generate the call and caller graphs. It thus inherits the limitation of generating an imprecise call graph when dealing with function pointers. This leads to unneccessary instructions to be copied, while others missing. Although possible with an additional manual effort, such limitations made our tool unable to automatically extract the larger organs from VIM. We can, in the future, turn to *Dynamic analysis* [15] and other code manipulating tools to optimise the efficiency of the slicing process, to efficiently

Table 6. Case studies results

Donors	Number of			Trans.time(min)	
	LoC	Functions	Files	Sys.	User
Kilo	~963	35	2	~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	2	~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

identify those statements in the program slice which have influence on the target organ. Investigating more precise techniques is future work.

In summary, prodScalpel was able to productise three features from three different systems. These case studies show initial evidence that software transplantation can be successfully used for building product variants automatically by combining features transplanted from real-world systems. However, attempted transplants of larger features proved unsuccessful. prodScalpel inherits limitations of the underlying slicing tools, thus its generalisability can be further improved with progress in that domain. New studies may give more evidence that our approach is extensible and flexible in other domains, opening perspectives for future work.

6 THREATS TO VALIDITY

Internal Validity. Due to time expensive nature of a human study, we had only 20 participants. We tried to mitigate this issue by selecting participants with experience in SPL and software development, both from academia and the industry (Table 2).

We use testing to validate Foundry, which cannot provide a formal proof of correctness. We used extensive testing to mitigate this risk, achieving over 70% statement coverage for each product generated (Section 4). We expect the resultant code to undergo standard code review process, as is standard in real-world settings.

We are limited by abilities of the tools we use. In particular, Doxygen, for call graph construction, and TXL for program transformation. Nevertheless, our studies show that prodScalpel can still be a useful tool in a real-world scenario to create products in an automated way (Section 4.5 and Section 5.2). Any improvements to internal tooling will only increase applicability of our approach.

External Validity. The relatively small number and diversity of systems used (Table 1 and Table 5)to derive new products poses a threat to generalisability of our study to larger software. We tried to mitigate this threat by constructing possible real-world scenarios. All subject chosen are popular software in everyday use. Gven that our approach was helpful even for small programs, we argue that it is likely helpful for larger systems as it automates feature extraction, adaptation and implantation — tasks that are currently largely performed manually [29].

Code inspection to transfer a feature to a product base is hard, which has been confirmed in our post-study survey (Section 4.5). We conducted a pilot study to mitigate threats arising from issues related to fatigue, task difficulty, and time required (Section 4.2). As a result, most participants were able to complete the given tasks within the time limit (Table 3).

In our experiment, one of the authors used prodScalpel for feature migration. Since they are familiar with the codebase and the tool, it might have taken them less time to setup the transplantations. Nevertheless, participants in our user study and the tool were given the same inputs. The work required by one of the authors boiled down to issuing commands to prodScalpel. The substantial time-efficiency gains observed (Section 4.5) support our claim that using prodScalpel outperforms current practice.

7 RELATED WORK

In this section, we present an overview of existing research related to our work. We position our work within the existing body of knowledge in areas of reengineering of systems into SPL, clone-and-own, variability in SPL and software transplantation.

7.1 Reengineering of Software Systems into SPL

Diverse academic proposals and industrial experience reports addressing reengineering of legacy systems into SPL are present in the literature [2]. However, this number decreases considerably when we are interested in proposals that automate the lifecycle of the reengineering process [30].

Martinez et al. [39] 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 [22] is a reactive approach [27] 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 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 the use of such directives to handle variability is not a mandatory condition to use it. The developer is free to automatically introduce conditional directives in the evolved product line. That is they can also generate a product without surrounding the feature's code with additional directives, simply by transplanting organs from the transplantation platform.

7.2 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 [31, 61].

Fischer et al. [18], 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.

7.3 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 works have already identified the frequent use of the *variability mechanisms*, like preprocessor directives [34] and feature flags [29, 41], 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 software product lines require explicit annotations often scattered across multiple code units (e.g., preprocessor annotations such as `#IFDEFs`) [1]. These annotations establish a mapping of portions of code to features defined in a variability model. This mapping serves as input to the 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 [29]. Liegib et al. [34] presents 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. Christian Kästner et al. [32] also discuss the concept of variability based on preprocessor directives in software product lines 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.* [24] 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 software product line 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 [3, 34]. Preprocessor directives and feature flags can cause code bloat and reduce maintainability. Rahman *et al.* [48] 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.* [41] also discovered that despite developers' initial intention to remove them, feature toggles tend to remain in the codebase unless their removal was compelled by policy or technical considerations.

When building an SPL the number of feature options can grow quickly [37, 60]. Thus, the use of feature flags and/or preprocessor directives can lead to a large codebase in an emergent product line [41]. Foundry can be an interesting alternative to those traditional variability techniques. Existing approaches rely on adding permanent annotations whose number increases over time. Foundry, in contrast, requires only annotating a feature's entry point and its insertion point in the product base. By obviating feature flags, Foundry's transplantation-based variability mechanism has the potential to reduce code bloat and increase readability. Furthermore, Foundry keeps all reusable features of a product line (its organs) functional and physically separate, thereby permitting an SPL engineer to defer integrating them into a product until they are actually needed to assemble a new product.

7.4 Software Transplantation

As far as the literature on automated software transplantation is concerned, Petke et al. [44, 45] were the first to transplant snipets of code from various versions of the same system to improve its performance, using genetic improvement [43]. One year later, Barr *et al.* formalised software transplantion and successfully transplanted a feature from one program into another [4]. Barr *et al.* did not discuss organ interdependencies in their work. Indeed, a limitation of their prototype — μTrans — is that it only handles one-off transplantations.

Stelios Sidiroglou-Douskos et al. [54] 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 [9] to identify the feature codes. 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 [9, 23].

More recently, Liu et al. [36] 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 reengineering of systems into SPL. Furthermore, they still not provide any tool that support their method.

In contrast to this previous work, we are the first to use automated software transplantation to automate SPL engineering tasks. Our approach, Foundry, and its realisation for C in prodScalpel 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 software transplantation literature, such as code redundancy, organ dependence and feature interactions.

8 CONCLUSION

In this work, we propose an approach, Foundry, and a tool, prodScalpel, that use software transplantation to speed conversion to and maintenance of SPL. We have validated both Foundry and prodScalpel through case studies. We generated two products by transplanting features extracted from three real-world systems into two different product bases. Moreover, we performed an experiment with SPL experts to compare our approach with manual effort. We showed a significant improvement in task completion time when using prodScalpel. The prodScalpel accomplished 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.

We argue that transplantation-based will be easier to adopt and maintain than a configuration-based software product line. Foundry improves SPL maintainability by physically separating features from the product base. Foundry can be used for Clone & Own or for either *extractive* or *reactive* product line migration. Foundry avoids code duplication when transplanting features, while preserving their behaviour. It can automatically propagate feature changes. That is, it addresses many problems often cited in the SPL reengineering literature.

Our evaluation provides initial evidence to support our central claim that SPLE based on software transplantation is a promising direction for SPL research and practice. However, more study is needed to provide more evidence for generalisability of our approach, and to investigate its applicability in an industrial context.

9 ACKNOWLEDGMENTS

This work was supported by the UKRI EPSRC grant no. EP/P023991/1; INES (www.ines.org.br), CNPq grant 465614/2014-0, CAPES grant 88887.136410/2017-00, PDSE/CAPES program, and FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14; and FAPESB INCITE PIE0002/2022; and Federal Institute of Bahia.

10 COPYRIGHT

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[2] Wesley K. G. Assunção, Roberto Erick Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22 (2017), 2972–3016.

[3] Felix Bachmann and Paul Clements. 2005. *Variability in Software Product Lines.* Technical Report CMU/SEI-2005-TR-012. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[4] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 257–269.

[5] Jonatas Ferreira Bastos, Paulo Anselmo da Mota Silveira Neto, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2015. Software Product Lines Adoption: An Industrial Case Study (Keynote). In *Proceedings of the Third International Workshop on Conducting Empirical Studies in Industry* (Florence, Italy) *(CESI '15)*. IEEE Press, Piscataway, NJ, USA, 35–42.

[6] Jonatas Ferreira Bastos, Paulo Anselmo da Mota Silveira Neto, Pdraig OLeary, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2017. Software Product Lines Adoption in Small Organizations. *J. Syst. Softw.* 131, C (Sept. 2017), 112–128.

[7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej kasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems* (Pisa, Italy) *(VaMoS '13)*. ACM, New York, NY, USA, Article 7, 8 pages.

[8] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1858996.1859010

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (feb 2010), 66–75.

[10] G. Bockle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid. 2004. Calculating ROI for software product lines. *IEEE Software* 21, 3 (2004), 23–31.

[11] H.P. Breivold, S. Larsson, and R. Land. 2008. Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies. *2008 34th Euromicro Conference Software Engineering and Advanced Applications* (2008).

[12] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, USA.

[13] Paul Clements and Linda M. Northrop. 2002. *Software product lines - practices and patterns.* Addison-Wesley.

[14] James R. Cordy. 2006. The TXL Source Transformation Language. *Sci. Comput. Program.* 61, 3 (2006), 190–210.

[15] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[16] Adekola Olubukola Daniel. 2020. ECONOMIC IMPACT OF SOFTWARE PRODUCT LINE ENGINEERING METHOD– A SURVEY. *International Journal of Advanced Research in Computer Science* (2020). https://api.semanticscholar.org/CorpusID:212553721

[17] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution.* 391–400.

[18] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 665–668.

[19] Critina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. *SIGSOFT Softw. Eng. Notes* 26, 3 (may 2001), 109–117.

[20] Andrew Gelman. 2005. Analysis of variance: Why it is more important than ever? *Quality Engineering* 51 (2005), 295–300.

[21] Mark Harman, William B Langdon, and Westley Weimer. 2013. Genetic Programming for Reverse Engineering. (2013), 1–10.

[22] Nicolas Hlad, Seriai Abdelhak-Djamel, and Dony Christophe. 2021. IsiSPL: Toward an Automated Reactive Approach to build Software Product Lines. arXiv:2107.00552 [cs.SE]

[23] Bilal Ilyas and Islam Elkhalifa. 2016. Static Code Analysis: A Systematic Literature Review and an Industrial Survey.

[24] Jean-Marc Jézéquel, Jörg Kienzle, and Mathieu Acher. 2022. From Feature Models to Feature Toggles in Practice. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 234–244.

[25] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

[26] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 311–320.

[27] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE '01)*. Springer-Verlag, London, UK, UK, 282–293.

[28] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE '01)*. Springer-Verlag, London, UK, UK, 282–293.

[29] Jacob Krüger, Thorsten Berger, Thomas Leich, and " Features. 2018. 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).

[30] Jacob Krüger, Sebastian Krieter, Gunter Saake, and Thomas Leich. 2020. EXtracting Product Lines from VAriaNTs (EXPLANT) *(VAMOS '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 2 pages.

[31] C. Kästner, A. Dreiling, and K. Ostermann. 2014. Variability Mining: Consistent Semi-automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering* 40, 1 (2014), 67–82.

[32] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code. In *In Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE*.

[33] Miguel A. Laguna and Yania Crespo. 2013. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming* 78, 8 (2013), 1010 – 1034.

[34] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 105–114.

[35] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[36] Lupeng Liu and Xiaoguang Mao. 2018. A Study on Code Transplantation Technique based on Program Slicing. 161, Tlicsc (2018), 294–298.

[37] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Evolution of the Linux Kernel Variability Model *(SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–150.

[38] Wardah Mahmood, Daniel Strüber, Thorsten Berger, Ralf Lämmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management with the Virtual Platform. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 1658–1670.

[39] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach. In *Proceedings of the 19th International Conference on Software Product Line* (Nashville, Tennessee) *(SPLC '15)*. Association for Computing Machinery, New York, NY, USA, 101–110.

[40] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[41] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 233–242.

[42] Linda M. Northrop and Clements Paul C. 2012. A Framework for Software Product Line Practice version 5.0. technical report. *Software Engineering Institute* (2012), 258. http://www.sei.cmu.edu/productlines/frame{_}report/index.html

[43] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (2018), 415–432.

[44] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Genetic Programming*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo García-Sánchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–149.

[45] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. 2018. Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation. *IEEE Transactions on Software Engineering* 44, 6 (June 2018), 574–594.

[46] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[47] Tomas Potuzak and Richard Lipka. 2023. Current Trends in Automated Test Case Generation. In *2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*. 627–636. https://doi.org/10.15439/2023F9829

[48] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 201–211.

[49] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. 2011. On the Impact of Feature Dependencies when Maintaining Preprocessor-based Software Product Lines. *SIGPLAN Not.* 47, 3 (Oct. 2011), 23–32.

[50] Chanchal K. Roy. 2009. *Detection and Analysis of Near-miss Software Clones.* Ph. D. Dissertation. Kingston, Ont., Canada, Canada. AAINR65337.

[51] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*.

[52] S. S. SHAPIRO and M. B. WILK. 1965. An analysis of variance test for normality (complete samples)†. *Biometrika* 52, 3-4 (12 1965), 591–611.

[53] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 461–470. https://doi.org/10.1145/1985793.1985856

[54] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. New York, NY, USA, 95–105.

[55] Leandro Souza, Earl Barr, Justyna Petke, Eduardo Almeida, and Paulo Neto. 2023. The project: software product line engineering via automated software transplantation. https://autotransplantation-spl.github.io/foundry.github.io/. Accessed: 2024-04-01.

[56] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. ACM, New York, NY, USA, 47–60.

[57] D. van Heesch. 2018. Doxygen: Source code documentation generator tool. http://www.stack.nl/dimitri/doxygen/

[58] Shangwen Wang, Xiaoguang Mao, and Yue Yu. 2018. An Initial Step Towards Organ Transplantation Based on GitHub Repository. *IEEE Access* 6 (2018), 59268–59281.

[59] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. 2009. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In *Proceedings of the 13th International Software Product Line Conference* (San Francisco, California, USA) *(SPLC '09)*. Carnegie Mellon University, USA, 211–220.

[60] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 307–319.

[61] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. 2006. 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* (Seoul, Korea) *(EMSOFT '06)*. Association for Computing Machinery, New York, NY, USA, 63–72.