Software Product Line Engineering via Software Transplantation

LEANDRO O. SOUZA, Federal Institute of Bahia, Brazil
EARL T. BARR, University College London, UK
JUSTYNA PETKE, University College London, UK
EDUARDO S. ALMEIDA, Federal University of Bahia, Brazil
PAULO ANSELMO M. S. NETO, Federal Rural University of Pernambuco, Brazil

For companies producing related products, a Software Product Line (SPL) is a software reuse method that improves time-to-market and software quality, achieving substantial cost reductions. These benefits do not come for free. It often takes years to re-architect and re-engineer a codebase to support SPL and, once adopted, it must be maintained. Current SPL practice relies on a collection of tools, tailored for different re-engineering phases, whose output developers must coordinate and integrate. We present Foundry, a general automated approach for leveraging software transplantation to speed conversion to and maintenance of SPL. Foundry facilitates feature extraction and migration. It can efficiently, repeatedly, transplant a sequence of features, implemented in multiple files. We used Foundry to create two valid product lines that integrate features from three real-world systems in an automated way. 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.

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

ACM Reference Format:
Leandro O. Souza, Earl T. Barr, Justyna Petke, Eduardo S. Almeida, and Paulo Anselmo M. S. Neto. 2022. Software Product Line Engineering via Software Transplantation . *J. ACM* 37, 4, Article 111 (August 2022), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software Product Line (SPL) is a systematic methodology for producing related software products from shared development assets [16, 34, 44]. SPL organises features as core assets that are shared across all products within the product line. Then, it defines a *dependency relation* over those features and a *variability mechanism* that creates products from subsets of the features, subject to the dependency relation. By centralising features into a product base, SPL prevents cross-product feature drift, allowing all products to benefit from a feature's improvement, if they use it. For companies producing related products, adopting SPL improves productivity and quality, speeds time to market, and reduces cost, because it facilitates the reuse of development artifacts, such as code and design [6, 34].

Despite its benefits, adopting SPL requires considerable upfront investment before its benefits can be realised. The cost of migrating existing products to SPL is lower than adopting SPL from scratch, making *extractive* [26] adoption more common, especially in companies with many software system variants in production [7]. Two factors drive this

Authors' addresses: Leandro O. Souza, leandro.souza@ifba.edu.br, Federal Institute of Bahia, Irecê, Bahia, Brazil; Earl T. Barr, University College London, London, UK; Justyna Petke, j.petke@ucl.ac.uk, University College London, London, UK; Eduardo S. Almeida, Federal University of Bahia, Salvador, Bahia, Brazil; Paulo Anselmo M. S. Neto, Federal Rural University of Pernambuco, Recife, Pernambuco, Brazil.

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 [10, 40].

To re-engineer 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 inter-dependencies. Finally, they must 4) define a variability mechanism for combining these features, subject to their inter-dependency constraints [2]. Currently, re-engineering to adopt SPL remains largely manual [2] and costly [9]. Indeed, because of its cost, software companies delay, or even refrain from, adopting SPL [16]. Automating these tasks remains an open challenge [2].

In 2013, Harman et al. [19] introduced software transplantation (ST) as a new research direction and laid out its implications for SPL re-engineering. 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*" [19]. In terms of automated software transplantation, Petke et al. [42, 43] were the pioneers in transplanting code snippets from different versions of a system to enhance its performance using genetic improvement [41]. 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. [49] , 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 3.2), the first software transplantation approach for SPL re-engineering. Foundry is independent of the programming language, and supports SPL's *domain engineering* and *application engineering* [12] processes at the code level. It tackles each SPL re-engineering 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 (*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, or features. Its use of slicing 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 [41] to specialise an over-organ to its implantation point and to detect and remove cross organ redundancies (Section 3.2).

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. prodScalpel also supports the use of existing variability mechanisms [17] based on *feature toggle* [45] or *preprocessor directives* [24]. 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.

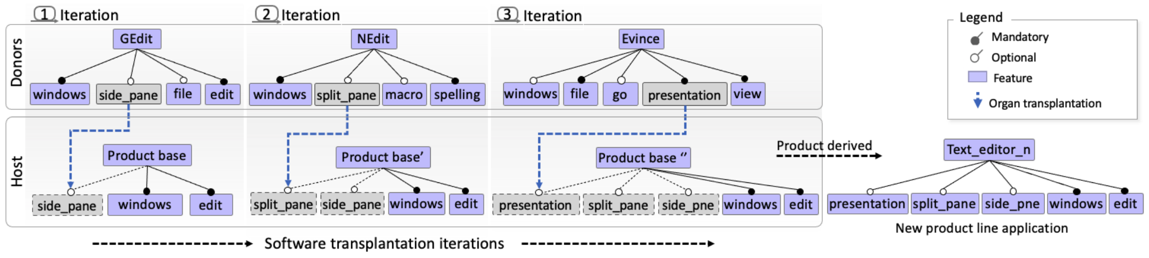


Fig. 1. Product derivation process using the Foundry approach. *prodScalpel transplants three features, in sequence, into the GEdit's product base to derive a new text editor after three iterations of organ transplantation.*

To evaluate prodScalpel, we conducted two case studies (Section 5.3) and a controlled experiment (Section 7). We first generate 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. Next, 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. In all cases, prodScalpel outperformed our experiment's participants in the time taken to transplant the feature. On average, prodScalpel took 18% of the time to transplant features from single systems to a product line than the participants who completed the task within the timeout.

Our results (Section 5.3 and Section 6.11) show that software transplantation speeds SPL re-engineering, by combining features extracted from existing, possibly unrelated, systems.

The main contributions of this paper are:

(1) Foundry, a novel SPL re-engineering approach that leverages software transplantation to extract and reuse features from existing codebases to construct a product line, 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, a tool that transplants multi-file organs and uses clone detection to prevent implanting redundant features;
(3) A rigorous evaluation of prodScalpel that demonstrates Foundry's promise. We use prodScalpel to generate two product lines and two new products, composed of features transplanted from three different real-world codebases. We also show that prodScalpel migrates features on average 4.8 faster than SPL experts performing the same task.

All the source code and data needed to reproduce this work are available at the project webpage [50].

2 MOTIVATING EXAMPLE

The open-source GNOME project[5] encompasses a large portfolio of individual programs 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 customisation and a common platform, principles of Software Product Line Engineering

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

(SPLE) [44], would allows to GNOME team reuses a common base of technology and, at the same time, to bring out products tailored individual customers. Without a common platform and a software development process base on mass customisation, it may be more difficult to the GNOME project provides 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 have prevented it from adopting SPL. FOUNDRY is transformative because it can be used to reduce this cost.

We show how the GNOME team could use PRODSCALPEL to quickly generate a product line. 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*[6], with additional features. Since they have decided to augment Gedit, GNOME team would select it as the product base, the shared substrate of a product line that, for FOUNDRY, serves 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*[7], a multi-purpose text editor that is not part of the GNOME portfolio, and *Evince*[8], a document viewer for multiple document formats that is part of GNOME, not not an editor.

PRODSCALPEL iteratively and incrementally re-engineers the codebase for SPL. In the donor, engineers need to demarcate all feature entry points to transplant; a single annotation is sufficient for PRODSCALPEL to extract a feature.

To prepare the host, GNOME engineers use PRODSCALPEL to extract a product base from an existing system by removing all features not shared across all products within the built product line

Then, the engineers must annotate the host to indicate the implantation point for each target feature, or "organ" using transplantation nomenclature. GNOME engineers then run PRODSCALPEL on these inputs, once per feature, with

`./prodScalpel —seeds_file`: *The file which contains the seeds for Genetic Programming(GP) algorithm.*
`—donor_folder`: *The path to the donor source code.*
`—host_target`: *The file in host that contains the insertion point of the transplant.*
`—donor_target`: *The file in the donor that contains the core function.*
`—workspace`: *The path to the workspace of the transplant.*
`—core_function_target`: *The file which contains all feature entry points.*
`—host_project`: *The path to the product base source code.*

This command automatically extracts all of the specified feature's source code and its dependencies, or "over-organ" using transplantation nomenclature. More operational parameters are available at the project: webpage [50].

Figure 1 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 [23] to represent each donor system and the product base evolution, PRODSCALPEL first transplants the *side-panel* feature, extracted from GEdit itself. This transplantation demonstrates that PRODSCALPEL can transplant features into a product base that comes from the same codebase. Next, PRODSCALPEL transplants the *split_pane* feature from NEdit. It shows how PRODSCALPEL manages to transplant features from distinct codebases, which is not possible without manual effort using the current state-of-art to SPL re-engineering. Finally, PRODSCALPEL transplants the *presentation* feature from GNOME's Evince renderer.

[6]https://wiki.gnome.org/Apps/Gedit
[7]https://sourceforge.net/projects/nedit/
[8]https://wiki.gnome.org/Apps/Evince

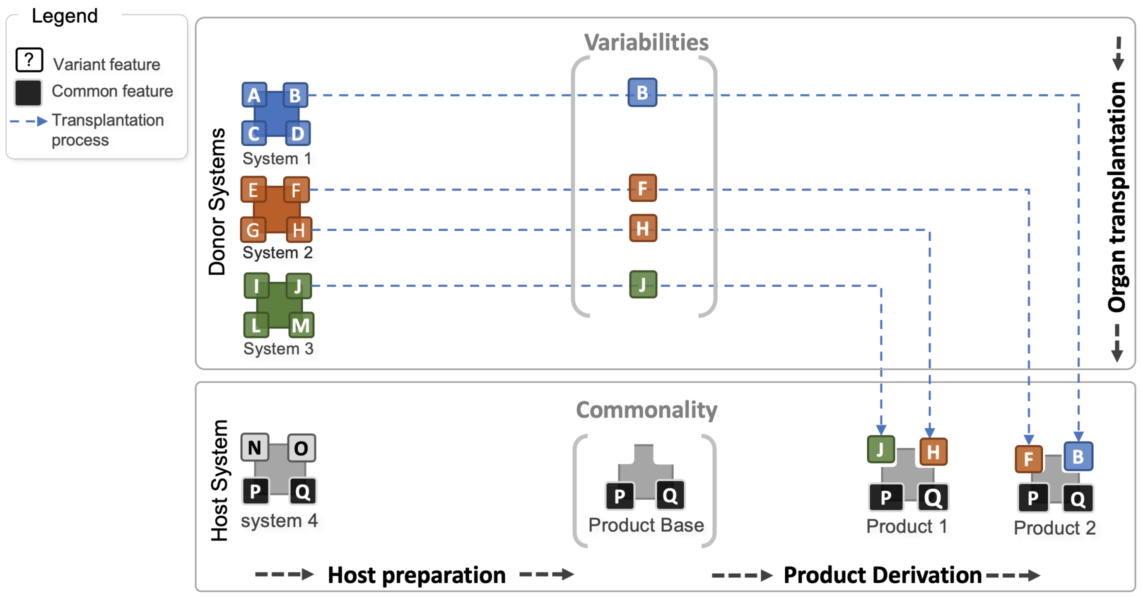


Fig. 2. An overview of how new products are derived from a product line based on ST.

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.

3 FOUNDRY

Based on software transplantation 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. 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 [4]. An over-organ can be specialized to became an organ that preserves the original behavior of the feature in a different host codebase [4].

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 transplantation process, as illustrated in Figure 2. This idea opens new ways for SPLE area by automated construction of different products by transplanting multiple organs into a product base.

It is important to note that Foundry offers two ways of creating products by transplanting organs from a pre-established/created *transplantation platform*, a repository of transplantation assets, or by directly extracting and transplanting an organ from a donor into a product base, even if the over-organ is not present in the transplantation platform. The two ways can also be combined to create specialized products.

In the rest of this section, we overview Foundry's workflow, describing how it applies software transplantation idea to re-engineering of product lines from existing systems. Foundry is independent of the programming language, and supports SPL's *domain engineering* and *application engineering* [12] processes at the code level.

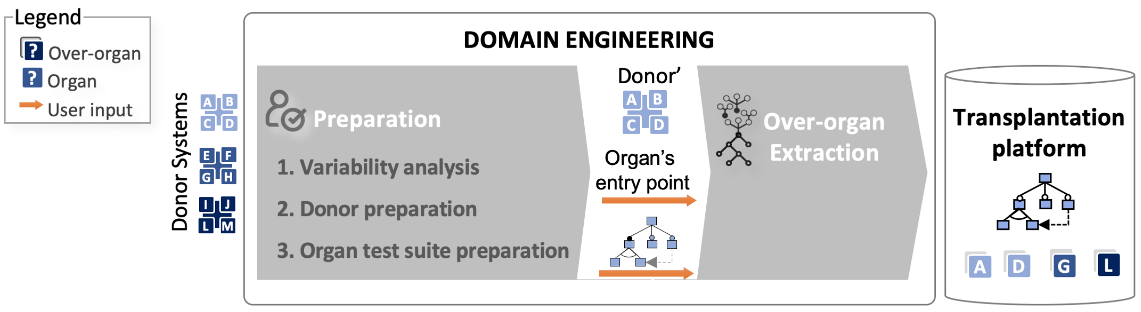


Fig. 3. Domain engineering process supported by FOUNDRY. *Four over-organs (A, D, G, L) are extracted from three donor systems and kept in the transplantation platform, with product base consisting of 2 features shared across all products (P and Q).*

3.1 Domain Engineering

In SPL's lifecycle the domain engineering corresponds to the process of establishing a reusable platform of core assets [12]. 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 corresponds to the process of establishing a product line composed of a product base and a set of reusable over-organs extracted, both stored in the transplantation platform. Figure 3 illustrates the stages of the domain engineering process.

As in medicine, Foundry has a *preoperative* stage where donors and the host are prepared for the transplantation process and a *postoperative* stage where we evaluate if the transplantation was successful (Section 3.2 for more details on the postoperative stage). The preoperative stage defines pre-transplantation tasks, responsible for the variability analysis process, the organ's test suite, donor and host preparation.

Variability Analysis. The preoperative stage starts with a variability analysis process to discover features in existing products with the potential to create the target product line. This process aims to create a variability model to express the valid combinations of features among the donor systems from which they will be extracted.

A variability model can be represented by using a feature model [23]. We augmented the traditional feature model representation to incorporate software transplantation inputs required to manipulate and maintain the over-organs in the product line. Each over-organ representation in the feature model is annotated with its corresponding entry point in the donor codebase. 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 [4]. To determine the organ's entry point, the SPL engineer needs to provide the name of the function implemented in the donor codebase.

Donor Preparation. This task in the preoperative process consists of cleaning up of donor codebases. These can contain some code 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 [51]) commonly used to control code extensions related to features. Although useful for the donor program, such code, if transplanted as part of the target organ, will generate dead code [51] that will never be executed in any transplanted product.

Previous work [4] with focus on transplantation of a single feature did not concern with donor clean-up. However, even when transplanting a single organ, dead code, if not removed apriori, can lead to unnessasary bloat and lower efficiency of the over-organ adaptation process (Section 3.2). In Foundry, the donor clean-up can be performed in a manual or automated way. During the process the source code structure of the program needs to be preserved (indentation, spacing, number formats, etc.), to prevent future bugs.

Organ Test Suite Preparation. For product derivation, an SPL engineer must supply test suites, called *ice-box tests* [4]. They are used to guide genetic programming in the over-organ adaptation process to create an organ that is fully executable when implanted in the product base (see Section 4.3). Ice-box tests can be easily implemented as proposed by [4] and integrated into the transplantation platform to be used for new transplantations of the target organ. These can be quick to develop, using existing test generation tools, or even adapted from donor's unit tests, when available.

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 transplantations and reuse of a single over-organ.

Host preparation. The SPL engineer has to select a product base. It is 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.

In case it is necessary, the product base can be reduced to its basic form, keeping only mandatory or features relevant to the target product line. The reduction process consists of removal of all code that implements all features which will not be required to create the target products. In a removal scenario, an additional attention is necessary to find and remove all portions of code that implement each unwanted feature without compromising the product base.

A product base, once reduced, can be used multiple times as a base for new products. 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 other products belonging to the same or similar domain.

Over-organ Extraction. Once the donor is prepared, it is possible to start the over-organ extraction process. In this stage, all source code related to the organ to be transplanted, that implements the target feature, must be identified and extracted.

Conceptually, an over-organ is composed of an organ and its vein, in keeping with the transplantation analogy [4]. While the organ implements the functionality we wish to transplant, the vein is a path from the donor entry that builds and initializes an execution environment for the organ [4]. Thus, all the code belonging to the organ and its vein must be captured in the extraction process.

In practice, organs can consist of several lines, one or more classes, as long as they fully implement a specific functionality [53]. We consider an organ to be a functional implementation of a feature of interest to an emergent product line. This makes the code extracting process a challenging task since it also involves identifying all code elements for the organ to be kept functional even out of the donor's environment. The extraction of a target over-organ can thus 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 [53]. For instance, the feature `FEAT_DIFF` implemented in VIM has more than 5k LOCs scattered across 33 of its 166 source files. Previous work extracted features from a single file [4].

Here we have four challenges that need special attention. First challenge concerns the integrity of the extraction when the extracted code appears in several files, issue also identified by Wang et al. [53]. In this case, the extracted organ also needs to preserve the original multi-file structure. Otherwise, it may be even more difficult to keep it functioning outside of the donor's environment or to propagate eventual changes to the desirded feature (bug fixes, enhancements, etc.). Second, defects might be introduced due to code redundancy stemming from multiple organ transplantations. Multiple organs can use the same code, creating code duplication and possible errors, if not handled correctly. Third, an organ's vein can contain a large amount of code, especially when the host and the donor have very different structures. This requires extensive modification during the adaptation process. According to Barr et al. [4], inlined function calls constitute the second largest number of code lines transplanted. Fourth challenge is that an extracted over-organ may itself contain multiple smaller features, which its functionality depends on. For example, a `spell_checker` feature might depend on a memory-resident database feature. Thus, the extraction process has to implicitly learn feature's dependencies, by including them in its over-organ. Such issues, if not solved, make impracticable the generation and maintenance of product lines using the software transplantation approach.

To solve all additional challenges highlighted above, we have evolved the organ extraction process, introduced by Barr et al. [4], to compute slices in multiple files. Thus, even if an over-organ is contained in multiple files, Foundry can obtain a practical over-organ for transplantation, maintaining its original structure.

Given an entry point in the donor provided by the user, Foundry uses conservative slicing to automatically extract a feature into an "over-organ", completely automating the extraction task. Based on the method introduced in [4], Foundry slices *forward* and *backward* from the given organ entry point to identify the organ and one of its veins.

At the end of the extraction process, the source code of the over-organ is then stored in the transplantation platform, together with other over-organs that compose the product line. All over-organs in the platform are available to be reused during the application engineering process.

3.2 Application Engineering

In SPL, *application engineering* corresponds to the phase where features are assembled to create a product. This is the phase where variability is realized so that artefacts customization takes place.

In Foundry, application engineering corresponds to the phase of developing customized products through the organ transplantation process. After the execution of multiple iteractions of organs transplantation a new product is derived as new organs are transplanted. Thus, the flexibility required to customize products is provided by extracted over-organs that are combined with a product base.

As illustrated in Figure 4, application engineering process is supported by Foundry by applying four stages of software transplantation: (i) over-organ selection, (ii) over-organ reduction and adaptation, (iii) organ implantation and (iv) postoperative stage.

Over-organ Selection. At the start of the transplantation process, an SPL engineer selects the target features that will be transplanted into the product base to create the target product. The choice is guided by the feature model generated during the variability analysis process, so as, to support the SPL engineer to handle eventual relationship and restrictions among transplanted organs. Once the target feature is chosen in the feature model, it is possible to find the corresponding over-organ in the transplantation platform.

Over-organ Reduction and Adaptation. The over-organ reduction and adaptation processes involve specialising the organ to the host environment, i.e., 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.

Barr et al. [4] automated the over-organ reduction and adaptation process using a GP algorithm. GP reduces an over-organ and specialises it to the host environment. It thus creates an organ that preserves the original behaviour of the feature at a given insertion point in the host environment. However, Barr et al.'s approach does not support the

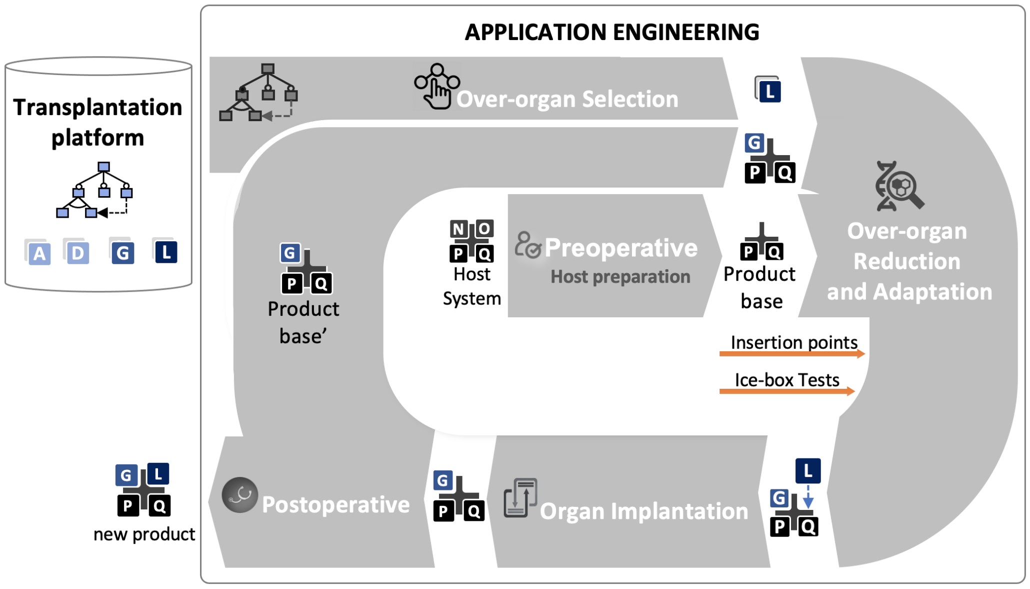


Fig. 4. Application engineering process supported by FOUNDRY. *A new product is derived after two software transplantation iterations (organs G and L).*

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 overcome these issues FOUNDRY introduces an organ-host wrapper. This layer is responsible for providing access to the organ from the target host. It is automatically constructed on demand, according to a given implantation point in the product base.

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

Organ Implantation. In this stage of the process, the organ is ready to be implanted into the product base. However, to make the application of this technique in generating SPL 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 [46]. Dependencies among features are established by means of structural dependencies in the source code shared between elements of different features [11]. 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. When this happens, the *organ collision* problem occurs since both organ A and B extracted from the same donor must contain the shared function.

Figure 5 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 of organ collision, transplantation process can insert code that is duplicated from multiple organ transplantations. Such a problem, if it is not managed, will lead to unwanted duplicated code, possibly breaking the postoperative product.

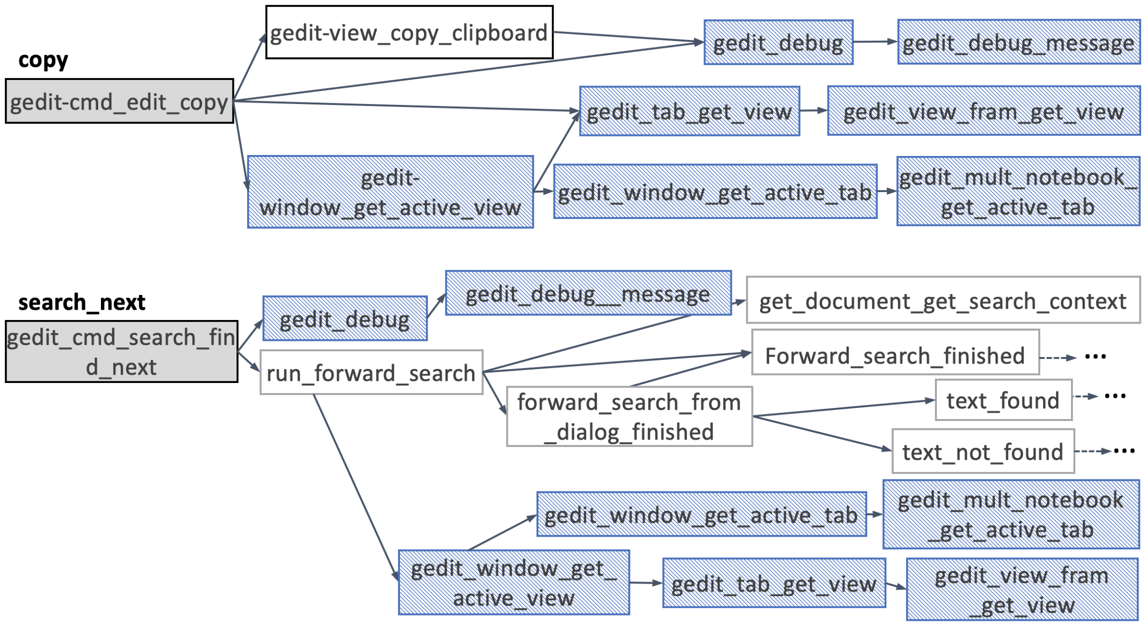

copy
gedit-cmd_edit_copy
gedit-view_copy_clipboard
gedit_debug
gedit_debug_message
gedit_tab_get_view
gedit_view_fram_get_view
gedit-window_get_active_view
gedit_window_get_active_tab
gedit_mult_notebook_get_active_tab
search_next
gedit_cmd_search_find_next
gedit_debug
gedit_debug__message
get_document_get_search_context
run_forward_search
Forward_search_finished
forward_search_from_dialog_finished
text_found
text_not_found
gedit_window_get_active_tab
gedit_mult_notebook_get_active_tab
gedit_window_get_active_view
gedit_tab_get_view
gedit_view_fram_get_view

Fig. 5. 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.*

In Foundry, code elements (functions, directives, constants, declarations of several global variables and their definitions) already belonging to the beneficiary or to more than one organ are characterized as *implicit connection points* since they can represent a connection or dependence points among two or more organs.

To correctly work, the implantation process need to identify all duplicated code elements and insert them only once into the product base, avoiding code duplication. However, hosts tend to have large input spaces into which code is inserted. In this way, finding the implicit connection 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 "clones".

Such particular aspect represents a new challenge in the software transplantation field for SPL, handled by Foundry. We solve this challenge by using code clone detection, to avoid duplicated code insertion.

Postoperative Stage. As in medicine, Foundry requires checking the side-effects of the transplantation operation. Based on [4], Foundry's postoperative stage introduces three validation steps, as outlined in Figure 6. Extending the validation process proposed in [4, 19], we highlight the three test suites that Foundry uses to evaluate the quality of a transplant: *Regression*, *Regression++*, and/or *Acceptance* tests. Once selected, organ has been successfully transplanted, and the postoperative product has passed all postoperative validation steps, we incorporate the Regression++ and Acceptance tests into the host's existing regression test suite for use in the next transplantation.

After the postoperative stage, new iterations of organ transplantation can be performed; thus, in a stepwise and incremental way, a new product is derived as organs are transplanted into a product base.

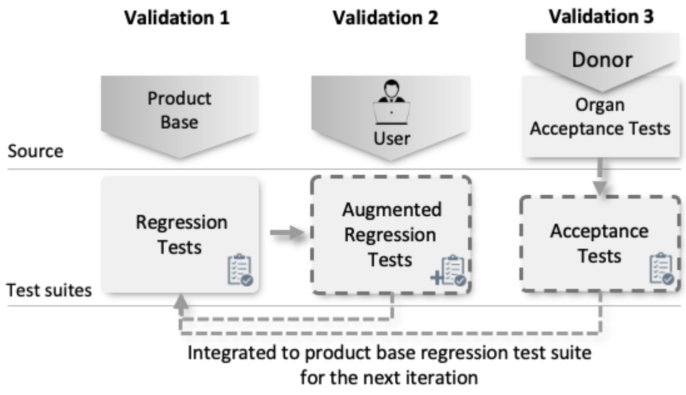


Fig. 6. Three validation steps; the dashed boxes are test cases added into the host regression test suite after each transplant iteration.

4 IMPLEMENTATION

We now present prodScalpel, our realisation of Foundry for C. prodScalpel extends μScalpel [4] to transplant multiple organs at once and implant them into a single host codebase. This feature is essential for supporting Foundry, since many features, especially those existing, SPL-oblivious codebases, span multiple files.

The prodScalpel's architecture is composed by five modules as shows in Figure 7. The domain engineering process is supported by modules donor preparation and organ extraction. The donor preparation module is responsible for cleaning up donor codebases by removing all unwanted preprocessor directives, in case it is present in the codebase. The organs extraction module, in turn, is responsible for extracting an over-organ using the program slicing technique.

The application engineering process is supported by the modules host preparation, organ reduction and adaptation, as well as, the organ implantation module. The preparation module is responsible by preparing the product base. It removes all unwanted features from the host codebase surrounded by preprocessor directives. The module organs reduction and adaptation uses GP to reduce an over-organ selected from the transplantation platform and adapt it to execute into the target product base. The last module, organ implantation, is implemented with a clone detector that identifies code elements duplication and dependencies while implanting them into the product base.

More details on how prodScalpel automates the Foundry process is provided in the following sections.

4.1 Automating Feature Removal

First we need to select and prepare our product base. The selected program might contain some unwanted features for the target product, thus we want to remove them before transplantation takes place. To automate removing unwanted features and dead code from the donor and host systems, prodScalpel implements a *Reconfigurator*. To work, prodScalpel initially requires engineers to provide as input a textual list of preprocessor directives corresponding to each feature and annotation to be removed. From this, the tool searches for pieces of code that implement features limited by preprocessor directives, removing them from the product base. prodScalpel then maps and searches selectively for pieces of code that implement features limited by such directives. Then, prodScalpel removes all features source code from the product base while it keeps the source code structure belonging to the product base unchanged and ready to receive the transplanted organs. Figure 8 gives a example of a portion of code after prodScalpel cleaned up unused directives.

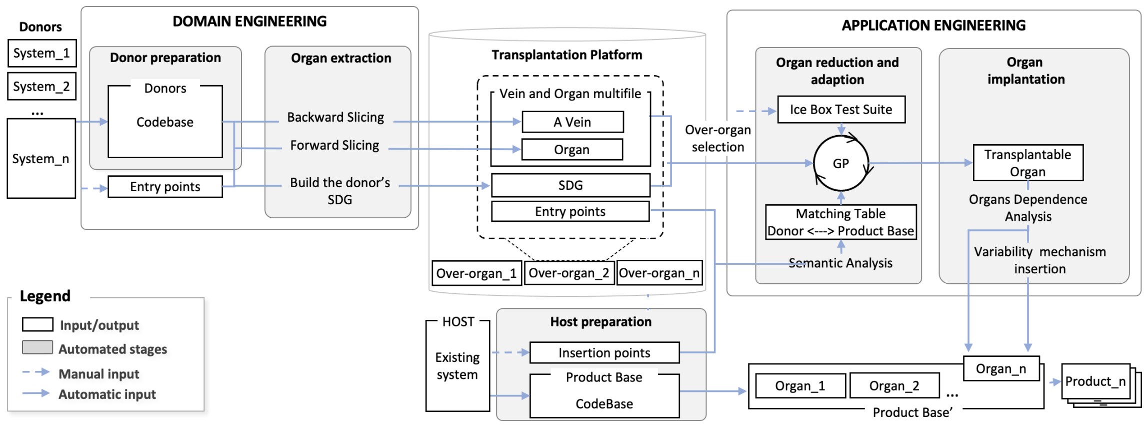
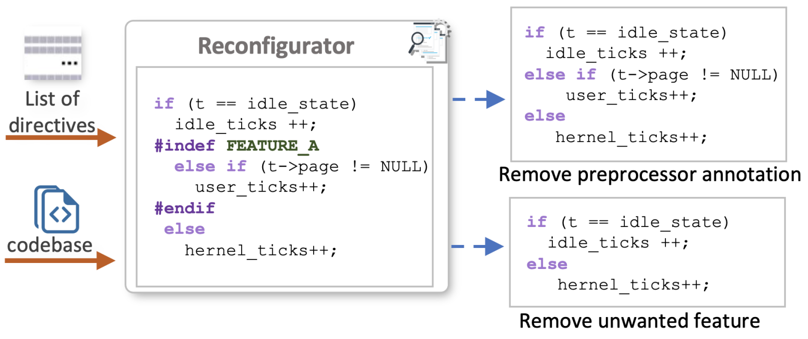


Fig. 7. Overall architecture of prodScalpel. *An SDG is a system dependency graph that the genetic programming phase uses to constrain its search space.*



```
if (t == idle_state)
  idle_ticks ++;
#indef FEATURE_A
  else if (t->page != NULL)
    user_ticks++;
#endif
 else
   hernel_ticks++;
```

```
if (t == idle_state)
  idle_ticks ++;
else if (t->page != NULL)
    user_ticks++;
else
   hernel_ticks++;
```

```
if (t == idle_state)
  idle_ticks ++;
else
   hernel_ticks++;
```

Fig. 8. The reconfigurator used to automated removal of preprocessor annotation from donor codebase and unwanted features from product base.

4.2 Automating Vein and Over-Organ Extraction

As optimisations, prodScalpel extracts the vein from multiple files. prodScalpel uses backward slicing to identity each function belonging to the vein and saves it in a copy of its souce file in the transplantation platform. Then, it includes all functions calling existing into vein source code in the *array of over-organ statements* used by GP to achieve an organ from the over-organ [4]. Thus, a function/method in the vein shared between two organs is not duplicated when transplanted.

prodScalpel avoids the problem of vein redundancy when a great part of the vein is common for multiple organs transplanted from the same donors. The vein duplication is discovered by the implantation module 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 looks for duplication also in the vein code of organ A. For example, imagine that a vein has a function `fx()` implemented in file `Fċ` and belonging to both over-organs A and B. When prodScalpel 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, prodScalpel does not insert it into the host again. Instead, the function `fx()` already transplanted also is used by organ B.

prodScalpel also identifies occurrences of mutually recursive functions, even an occurrence of an indirect recursion. Technically, prodScalpel inlines the vein code while puts each function found in a stack of functions. When prodScalpel finds an occurrence of a recursive function it recovers the beginning of the recursive call and does not inline it. Instead, prodScalpel 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, prodScalpel can produce a search space for GP with a finite amount of program statements even from recursive functions.

To handle transplantation of organs spread in multiple files, prodScalpel records the name of the files where the slice is and its location with respect to other slices from the over-organ. 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, prodScalpel computes the resulting slices in copies of its original files in the transplantation platform, without breaking the over-organ.

4.3 Automating Over-organ Reduction and Adaptation

prodScalpel 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, prodScalpel uses GP, as in previous work [4], 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, prodScalpel 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.

prodScalpel uses GP 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 prodScalpel 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.

In the end, prodScalpel synthesises a call to the extracted organ to execute and test it from the wrapper constructed.

4.4 Automating Multiple Organ Implantation

Once the organ is adapted to correctly work on the host it can be automatically implanted into the product base. To correctly insert an organ into the tool must identify and handle potential implicit connection points by avoiding the insertion of code duplication into the product base.

Figure 9 illustrates our solution to avoid the organ collision problem. To sum up, the clone detector checks if a specific code element is already present in the beneficiary's environment. To do this, prodScalpel constructs two lists, one with elements in the organ and other with elements in the host. Then, it checks if there is some element in the target organ which is already presented in a list of code elements transplanted. Once a potential element duplication is identified, prodScalpel decomposes both organ and host elements in ASTs to detect semantic dependencies between organ implementations previously transplanted, preventing shared elements to be inserted into the host again. Then,

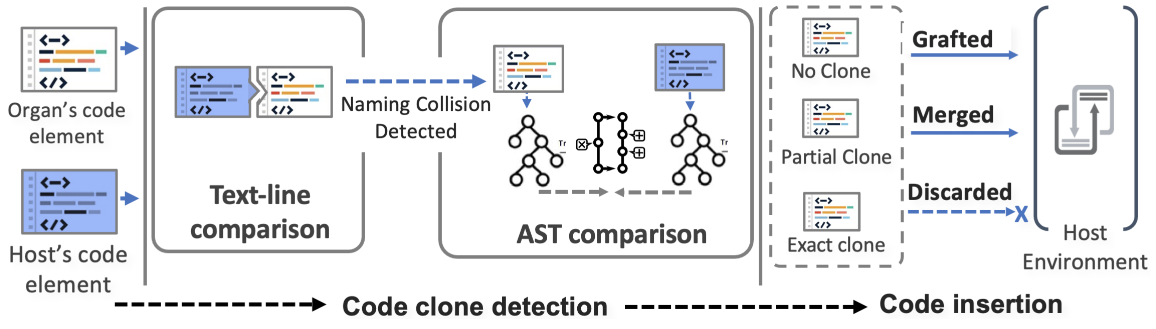


Fig. 9. Code clone detector. *It performs two comparison steps: text-line and ASTs comparisons to identify code clones and make decision if a code elements must be graft our not into the target product base environment.*

the code element is entirely *grafted*, *discarded*, or *merged*. In this last case, prodScalpel introduces additional line breaks such that potential variances within statements and other structures can be accurately inserted by using sub-abstract tree comparison.

We augmented prodScalpel with a *code clone detector*, based on NiCad [47]. This clone detector finds exact clones over arbitrary program fragments in the organ and host source code by using ASTs. Thus, we exploit the benefits of *Program differencing* [25] technique and TXL [13] to identify and compare potential syntactic code duplication using text-line and ASTs comparison [47].

5 CASE STUDIES

We evaluate prodScalpel on two case studies for generating new product variants from real-world systems. This section explains the objectives, research questions, and subject systems. Additionally, it presents the results of our study and discusses the implications. Our corpus and data collected are available at the project webpage [50].

5.1 Objective and Research Questions

The objective of this study is to evaluate the proposed approach and tool, thereby demonstrating the potential of automated software transplantation for product line generation. We aim to answer the following research questions:

RQ1. *How much effort is required to generate products from a product line created using prodScalpel?*

Given the complexity of the task at hand, it seems unreasonable to expect the product derivation process to be instantaneous. Still, it will need to be fast enough to be incorporated into a development cycle and demonstrate its advantage over manual effort. To answer this question, we computed the time required, and the number LOCs transferred to productize software.

RQ2. *Which features can prodScalpel productize?*

We discuss the characteristics of features that prodScalpel can and those it cannot yet handle. Given the inherent complexity of the process, it is important to clearly define its current limitations.

5.2 Methodology

Subject systems. We chose two text editors, *VI* and *VIM*, as product bases. For donors we chose subjects from two different domains: code analysis software and text editors. We chose *GNU Cflow*, a call graph extractor from C source

Table 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

code, as a donor. This is to show that donors can also 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.

Our donors are presented in Table 1. 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 1 presents more details on the subjects.

Procedure and execution. Initially, we automatically remove dead-code from both donors and host codebases by using prodScalpel. We also reduced each host to its basic form removing all optional features. Thus, both donors and host 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.

Case study environment. 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.

5.3 Results and Discussion

Table 2 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 has a total of 28k LoC and 40 features while product B has a total of 745k LOC and 121 features. Together, donors provided three feature variants to the product line, approximately 7.8k LOC to product A and 8.1k to product B, including a feature removed and re-transplanted in *VI*.

To answer RQ1, on average, prodScalpel spent 4h31min/1KLoC for transplanting three features into VI, and 4h40min/1KLOC for transplanting the same three features into VIM.

To answer question **RQ2**, 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.

prodScalpel uses Doxygen [52], 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

Table 2. 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* prodScalpel*'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 Procut 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	1234	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	1294	184
Product B	8,113	77	27	1,890	532

limitations made our tool unable to automatically extract the larger organs from VIM. We can, in the future, turn to *Dynamic analysis* [14] and other code manipulating tools to optimise the efficiency of the slicing process, to efficiently identify those statements in the program slice which have influence on the target organ. Investigating more precise techniques is future work.

To answer RQ2, prodScalpel was able to productise three features from three different systems. However, transplantation of organs from a larger donor exposed some limitations of the technologies used by our tool.

In summary, 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. 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 also extensible and flexible in other domains, opening perspectives for future work.

6 EXPERIMENTAL EVALUATION

In the previous case studies, we showed the viability of our software transplantation approach, implemented in prod-Scalpel, to generate customized products 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 [2, 29]. To the best of our knowledge, there is currently no comparable tool that manages to automatically transplant features from distinct donor systems to generate a product line. Therefore, we compare our approach with current state-of-the-art, namely human effort. We conducted an experiment that reflects a real-world process of product line migration from existing codebases [27]. In this section, we state our research objectives and describe in detail the experimental setting.Our corpus and data collected are available at the project webpage [50].

6.1 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 [54], we have framed our research objectives using the Goal Question Metric (GQM) method suggested by Basili [5]. Our goal is to:

Analyse 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.

6.2 Research Questions

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

RQ3. *How successful is* PRODSCALPEL *at feature migration when compared with human effort?*

We would like to understand how well our approach automatically transfers all required code so that the target feature can run in an emergent product line and compare it with the manual process.

RQ4. *How much feature migration time can be gained using* PRODSCALPEL *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 and compare these with the same processes when using PRODSCALPEL.

6.3 Measures

With the objective to answer our research questions, we defined the measures that must be computed. For each question, one measure was defined. These are described as follows:

M1. For RQ3, the accuracy of our approach is computed by verifying if PRODSCALPEL successfully migrated new functionalities to a product line by passing all the regression, augmented regression and acceptance test suites. This will check whether or not the output of the transplanted feature is correct with respect to the given test suites.

M2. For RQ4, we simply report the time that is spent on each activity to transfer the target features. These activities are: *code extraction*, *adaptation*, and *merging*. The time for each of these activities was collected individually.

6.4 Methodology

We use two donor systems in our experiment: *NEATVI*[9], text editor extended from VI for editing bidirectional UTF-8 text and *Mytar*[10], an archive manager. Table 3 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 [37], 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*[11] and *Group B*[12] consisted of a process description

[9]https://github.com/aligrudi/neatvi
[10]https://github.com/spektom/mytar
[11]The guideline for perform the transplantation in scenario I is available at https://rb.gy/vydhbb
[12]The guideline for perform the transplantation in scenario II is available at https://rb.gy/covz62

Table 3. Details of donors and product base systems used in our studdy.

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		

and systems documentation. Additionally, we provided training on re-engineering software systems to SPL 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 4. We also conducted a post-survey to better understand participants' problems encountered.

6.5 Experimental design

We answer our research questions by simulating a real reengineering process where two features must be transferred to a product line built over a product base. The experimental design was inspired by documented real product-line migration scenarios [2, 32].

In scenario I, we gathered a group of 10 SPL experts (called Group A) where each one of them had to manually 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 re-engineering to SPL where system variants came from both similar (scenario I) and distinct codebases (an archive manager providing features to a text editor's product line - scenario II) as Figure 10 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 line generation process and the payoff for using our approach. That is, to analyze the success of our approach (RQ3), we evaluate how often the transplanted features pass all the provided test cases. To analyze the performance of the approach (RQ4), 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.

6.6 Pilot Study

First, 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 for 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.

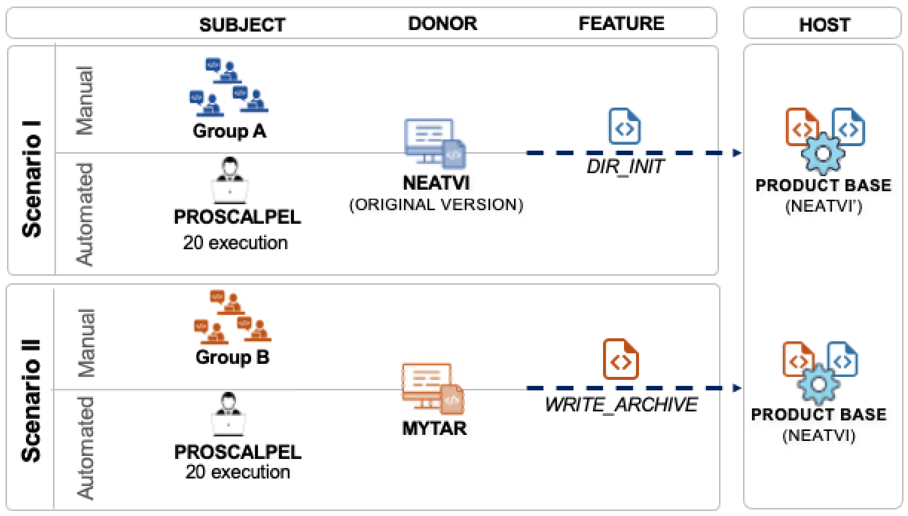


Fig. 10. Experimental design: one factor with two treatments applied in two re-engineering scenarios.

6.7 Participants

After the pilot study, we recruited 20 new 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.

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[13]. We created balanced groups (A and B) of participants for each product line generation scenario, based on their experience. Table 4 shows the details of the participants involved in the experiment.

6.8 Operation

Before the participants receive their tasks, we introduced the experiment with a *tutorial* on 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, 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

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

Table 4. *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)

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 emergent product line with a new feature transferred from the correspondent donor system, all participants were required to perform three activities based on the migration of system variants to the target product line [26, 37]: 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, it had to insert the feature's code into the product base, and validate its correctness via regression testing.

6.9 Data collection

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 manual re-engineering process.

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.

6.10 Data Analysis

We used 22 pre-existing regression test suites designed by the *NEATVI* developers to assess the success of prodScalpel and manual transplantations and answer our **RQ3**. 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 re-engineering process. To achieve 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 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.

6.11 Results and Discussion

We summarise our results in Table 5. 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 [14]. 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 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 line 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.

RQ3: Our results show that prodScalpel outperformed participants in both scenarios with only one run timing out. Eight of twenty participants (considering both scenarios) were not able to transplant the target features without breaking tests.

As stated in the definition of measure **M2** and to answer **RQ4**, we evaluate the payoff of prodScalpel. Figure 11 graphically shows the time spent on each activity performed in the re-engineering to SPL process. In summary, Group

[14]The time measured for the participants is available at rb.gy/ant4g8

Table 5. **Scenario I:** Donor: NEATVI - Product Base: NEATVI without the desired feature. *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 runs of* prodScalpel*. We highlight the execution time of the participant that moqt 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.*

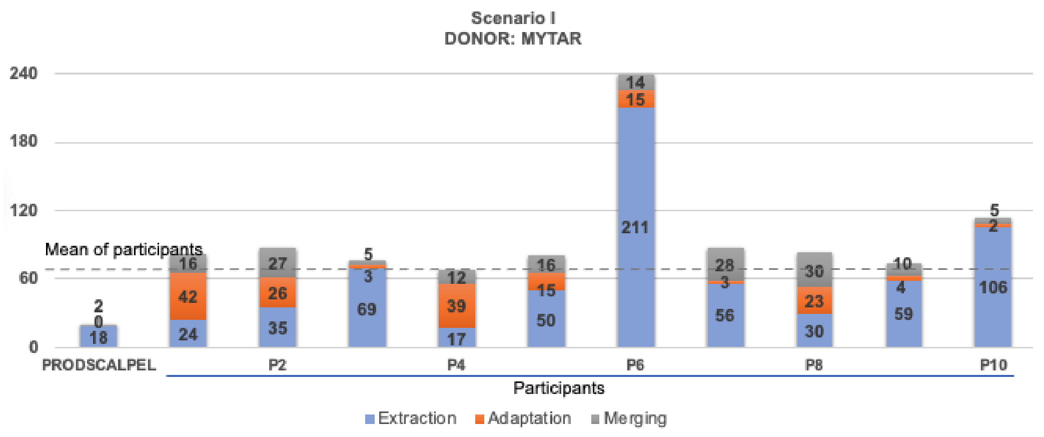
Participants	Product Line Status	Execution Time (minutes)	Test Suites Regre. (59%) PASSED	Regre.++ (70.1%) PASSED	Accept. (72.5%) PASSED
P1	OK	**82**	22/22	30/30	3/3
P2	OK	**88**	22/22	30/30	3/3
P3	OK	**77**	22/22	30/30	3/3
P4	OK	**68**	22/22	30/30	3/3
P5	OK	**81**	22/22	30/30	3/3
P6	Broken	**Timeout**	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 6. **Scenario II:** Donor: Mytar - Product Base: NEATVI. All other columns are the same as in the previous table.

Participants	Product Line 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**

A transferred the target feature from *NETVI* to the product base in 1h24 minutes on average. prodScalpel turned out to be quicker, successfully transplanting this feature in all 20 trials, taking an average of 20 minutes.

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.



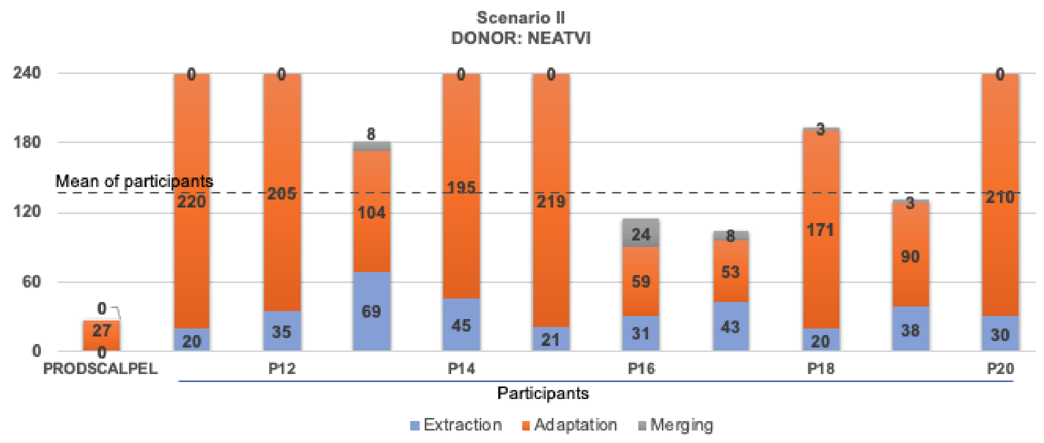


Fig. 11. Time (in minutes) spent by participants and prodScalpel on performing the three stages of SPL reengineering: feature *extraction*, *adaption* and *merging*. The graph highlights the average time spent by participants who successfully generated new products.

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

Statistical analysis of performance. To establish statistical significance of our results we first establish statistical distribution of runtimes for each scenario through the Shapiro-Wilk [48] test. Next, we use ANOVA [18] 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 12 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.

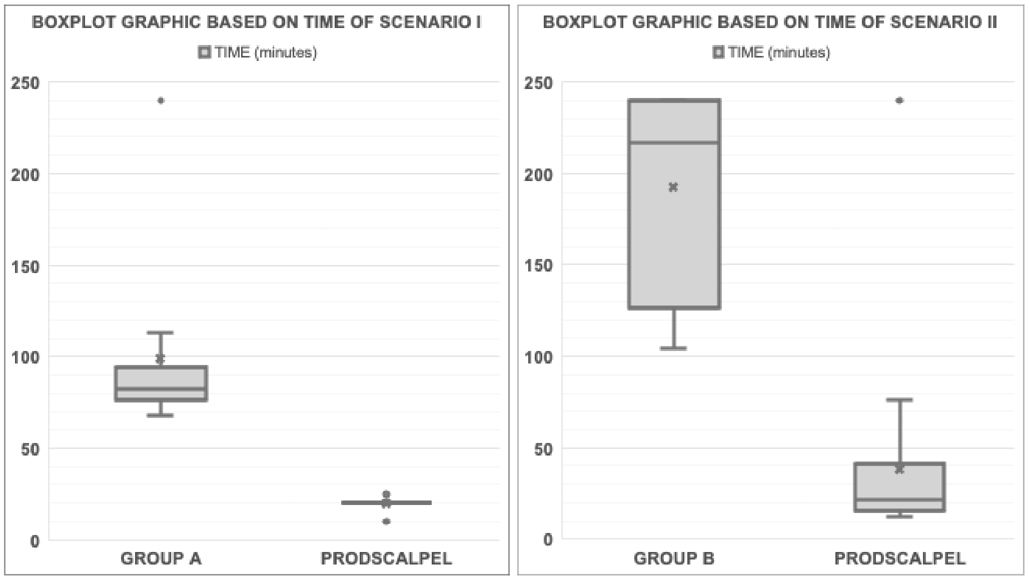


Fig. 12. Time results grouping automated and manual in both scenarios. Scenario I: *NEATVI* - Product Base; Scenario II: *Mytar* - Product base.

We can conclude that prodScalpel 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 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.

RQ4 The results confirmed that prodScalpel outperformed participants transplanting two features 4.8 times faster, on average, compared to participants who finished the process within timeout.

7 THREATS TO VALIDITY

In this section we present threats to validity of our studies.

External Validity. The relatively small number and diversity of systems used for derive new products pose an external threat to validity. For our user study 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 codebases and variations of the same real-world systems. Additionally, given that our approach was helpful even for small programs, we argue that it 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 [2] and without considerable adaption effort.

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 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, which has been confirmed in our post-study survey. We conducted a pilot study to mitigate threats arising from such issues. Even with a limited execution time, we were able to transplant features from donors of significant size, considering the number of code lines of the subject systems, as shown in Table 3. Most participants were also able to complete the given tasks within the time limit. We also use testing as means of validating our approach, which cannot provide a formal proof of 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 AUTOMATED SUPPORT FOR PRODUCT LINE RE-ENGENEERING

Automated software transplantation, as presented in Foundry, can provide a set of opportunities to improve and automate the product line re-engineering process. This alone might justify the effort to explore research opportunities in software transplantation for SPLE. Here we elaborate on and discuss such opportunities.

Automating re-engineering of existing systems into SPL. Principally Foundry is proposed as a new way of re-engineering existing systems into SPL. Companies can use Foundry just for the initial conversion of an existing codebase into SPL. In this mode, Foundry first extracts over-organs from the donor systems, then, produces a product base, created from an existing system by removing all unwanted features to the product line. Then, adapts and implants the organs into this product base, identifying and removing cross-over-organ redundancies. The resulting organs become the SPL's shared set of features. Foundry can itself be used as a variability mechanism to construct new products by implanting features into a product base. Even with this one-off application of Foundry, a company can also use Foundry to surround the transplanted feature with a conventional variability mechanism. For existing SPL codebase, such as the one it created. Foundry's variability mechanism insertion also supports surrounding implanted organs with feature flags or preprocessor directives, which permit enabling and disabling of features, to facilitate its integration into an existing SPL codebase that uses them.

Automating clone-and-own technique. Foundry can automate *clone-and-own* [15, 16], especially the task of synchronising changes to a feature shared across two products created by clone-and-own. For example, consider the case where the copy of a shared feature in one of the two products is patched to fix a bug. Foundry permits transplantation of the fixed version over the top of the unpatched copy of the feature.

Automating Reactive Product Line Adoption Process. Certain companies already have product lines, but adapt a reactive process, where a new feature is added only when a need for such a feature arises. Foundry can be used initially to generate product variants, and then, as the demand for specific products increases, it could be used to generate product lines.

Automating a symbiotic SPL. Foundry permits a wholly new form of SPL, *symbiotic SPL*: in this mode, the donor can be oblivious to a parallel, ongoing SPL reorganisation of its codebase. To capture improvements in the donor, prodScalpel periodically refreshes its set of features by re-transplanting them into its donor product base and hence into host products. For instance, one could use prodScalpel to produce lightweight, specialised text editors from the *Vim* project.

Supporting Product Line Maintenance The maintenance of assets and products is a challenging task and an inherent characteristic of a product line. When we include organs as assets in a product line the process becomes even more challenging. In fact, by trying to maintain an over-organ individually in the platform, a developer might introduce errors into the product line or in products derived from it, since their organs eventually share elements —such as variables and functions—with the maintained organ. A solution would be to re-implant the changed organ.

Nevertheless, a crucial problem here is how to re-implant the organ changed to match a different version of it already transplanted in the target product.

We envision two ways to avoid this problem and *maintain* a created product line. Firstly, one can re-transplant the features if the original source codebase changes. Secondly, one can maintain the extracted over-organs, and re-run the adaptation and implantation stages, as need be.

In both scenarios of product line maintenance, FOUNDRY can be set up to evolve a changed organ with feature flags. Surrounding an organ with feature toggles just before implantation is an interesting alternative for continuous deployment of organs. It thus separates inserted code and makes it easier to maintain. To support continuous deployment, the implantation process in FOUNDRY can be configured to involve each organ with feature flags to connect new, unreleased code to production. Once a transplanted organ is ready for production, developers can turn off the flag and reveal the new organ or its changes to users.

Providing a Variability Mechanism. FOUNDRY provides a variability mechanism itself. It allows developers to include or not a given feature, to instantiate different products at any moment only by transplanting them into the target product. It avoids the problem of maintaining and evolving a product line that consists of a massive number of products. Additionally, it avoids feature flag removal and technical debt around unremoved feature flags.

9 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 re-engineering of systems into SPL, clone-and-own, variability in SPL and software transplantation.

Re-engineering of Software Systems into SPL. Diverse academic proposals and industrial experience reports addressing re-engineering 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 re-engineering process [29].

Martinez et al. [38] 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 [20] is a reactive approach [26] 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.

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 [30, 56].

Fischer et al. [16], 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.

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 [33] and feature flags [28, 39], 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 [28]. Liegib et al. [33] 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. [31] 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. [22] 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, 33]. Preprocessor directives can lead to code bloat and reduced maintainability, while feature flags can add complexity and overhead to the codebase. Rahman et al. [45] 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. [39] 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 [36, 55], the use of feature flags and/or preprocessor directives can lead to a large codebase in the emergent product line [39]. 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 separate, 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 like 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.

Software Transplantation. As far as the literature on automated software transplantation is concerned, Petke et al. [42, 43] were the first to transplant snipets of code from various versions of the same system to improve its performance, using genetic improvement [41]. One year later, Barr et al. successfully transplanted a feature from one program into another [4].

Stelios Sidiroglou-Douskos et al. [49] 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 [8] 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 [8, 21].

More recently, Liu et al. [35] 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 works, we are the first to use automated software transplantation 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 software transplantation literature, such as code redundancy, organ dependence and feature interactions.

10 CONCLUSIONS AND FUTURE WORK

In this work, we propose an approach, Foundry, and a tool, prodScalpel, that use software transplantation to speed conversion to and maintenance of SPL. Both approach and the tool have been validated through case studies. We generated two products through the transplantation of 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 significant time effort improvements when using prodScalpel. The tool 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 the migration to SPL transplantation-based in contrast to a configuration-based software product line 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 behavior. It can automatically propagate feature changes. That is, it provides solutions for problems often cited in re-engineering of systems into SPL literature.

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

11 COPYRIGHT

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] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley.

[6] 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.

[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] 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.

[9] 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.

[10] 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).

[11] Bruno B.P. Cafeo, Elder Cirilo, Alessandro Garcia, Francisco Dantas, and Jaejoon Lee. 2016. Feature dependencies as change propagators: An exploratory study of software product lines. *Information and Software Technology* 69 (2016), 37–49.

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

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

[14] 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.

[15] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, USA, 25–34.

[16] 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.

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

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

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

[20] 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]

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

[22] 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.

[23] 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.

[24] 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.

[25] Brian W. Kernighan and Rob Pike. 1983. *The UNIX Programming Environment.* Prentice Hall Professional Technical Reference.

[26] 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.

[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] 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).

[29] 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.

[30] 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.

[31] 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.*

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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: 2023-04-25.

[51] 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.

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

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

[54] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[55] 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.

[56] 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.