The Journal of Systems & Software 203 (2023) 111736

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A benchmark generator framework for evolving variant-rich software ☆

Christoph Derks [a,*], Daniel Strüber [b,c], Thorsten Berger [a,b]

[a] Ruhr University Bochum, Universitätsstraße 150, Bochum, 44801, Germany
[b] Chalmers/University of Gothenburg, Department of Computer Science and Engineering, Gothenburg, 41296, Sweden
[c] Radboud University Nijmegen, Toernooiveld 212, Nijmegen, 6525 EC, The Netherlands

ARTICLE INFO

Article history:
Received 9 November 2022
Received in revised form 1 March 2023
Accepted 27 April 2023
Available online 4 May 2023

Keywords:
Generator
Variants
Product lines
Evaluation

ABSTRACT

Software often needs to exist in different variants, which account for varying customer requirements, environments, or non-functional aspects, such as energy consumption. Unfortunately, the number of variants can grow exponentially with the number of features. As such, developing and evolving variant-rich systems is challenging, since they do not only evolve "in time" as single systems, but also "in space" with new variants. Fortunately, many different methods and tools for variant-rich systems have been proposed over the last decades, especially in the field of software product line engineering. However, their level of evaluation varies significantly, threatening their relevance for practitioners and that of future research. Many tools have only been evaluated on ad hoc datasets, minimal examples, or unrealistic and limited evolution scenarios, missing large parts of the actual evolution lifecycle of variant-rich systems.

Our long-term goal is to provide benchmarks to increase the maturity of evaluation of methods and tools for evolving variant-rich systems. However, providing manually curated and sufficiently detailed benchmarks that cover the whole evolution lifecycle of variant-rich systems is challenging. We present the framework vpbench, which simulates the evolution of a variant-rich system and thereby generates an evolution enriched with metadata explaining the evolution. The generated benchmarks, i.e., the evolution histories and metadata, can serve as ground truth to check the results of tools applied on it. We formalize the claims we make about the generator and the generated benchmarks as requirements. The design of vpbench comprises modular generators and evolution operators that automatically evolve real codebases. We implement simple and advanced evolution operators—e.g., relying on code transplantation to incorporate features from real projects. We demonstrate how vpbench addresses its claimed requirements, also considering multiple degrees of realism, extensibility and language-independence of the generated benchmarks.

1. Introduction

Almost any software system needs to exist in multiple variants. Developers create variants to experiment with ideas and to address varying stakeholder requirements—including different markets, environments, and non-functional properties, such as performance or energy consumption. Unfortunately, *developing* variant-rich systems is challenging (Berger et al., 2020; Fogdal et al., 2016; Jepsen and Beuche, 2009; Jepsen et al., 2007; Krüger and Berger, 2020a,b; Kuiter et al., 2018; Vogel-Heuser et al., 2015). However, *evolving* them is even more complex (Berger et al., 2019; Krüger et al., 2020; Strüber et al., 2019) , especially compared to single systems. Organizations often start with clone&own– copying and adapting individual variants – as a simple and readily available strategy (Berger et al., 2013a, 2020; Dubinsky et al., 2013). During clone&own the different variants are evolved by manually propagating new features (end-user-visible functionality (Apel et al., 2013; Berger et al., 2015)), and other code changes (e.g., bug fixes) among the cloned variants. However, while cheap at first, the maintenance overheads quickly exceed the benefits of clone&own. Then, organizations need to integrate several or all variants into a configurable platform, realized with so-called variability mechanisms (Apel et al., 2013; Berger et al., 2013b; Van Gurp et al., 2001) that allow enabling or disabling features and their implementation via variation points in the code. A platform greatly reduces redundancies among the variants and allows to quickly derive new variants—often automatically, supported by model-based representations (e.g., feature models (Berger et al., 2013a; Kang et al., 1990)) and configurator tools. Still, evolving a platform is difficult, since

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: christoph.derks@rub.de (C. Derks), danstru@chalmers.se (D. Strüber), thorsten.berger@rub.de (T. Berger).

https://doi.org/10.1016/j.jss.2023.111736
0164-1212/

developers work on many variants at the same time, with variation points (e.g., using `#ifdef`) cluttering the source code and features being scattered across the codebase.

A huge portfolio of methods and tools has been proposed over the last three decades in the field of software product line engineering (Apel et al., 2013; Pohl et al., 2005) to support the evolution of variant-rich systems—software product lines (SPLs). These methods and tools provide techniques (Alves et al., 2006; Assunção et al., 2017; Krüger et al., 2020; Mahmood et al., 2021; Pfofe et al., 2016; Rattan et al., 2013; Rubin and Chechik, 2013; Schlie et al., 2020; Schulze, 2019; She et al., 2014) to automatically locate features in source code, to manage and identify clones, to propagate changes and features, to re-engineer cloned source code into a configurable platform, to manage and evolve feature models, to evolve the configurable platforms, and many other evolution scenarios. So, in summary, many different methods and tools exist that support individual parts of the typical evolution lifecycle of variant-rich systems, covering ad hoc clone&own, the migration to a configurable platform, and its evolution.

A core challenge is the evaluation of these techniques (Strüber et al., 2019). A recent study found that only 3 of 11 common evolution scenarios of variant-rich systems are fully supported by benchmarks (Strüber et al., 2019). While open-source variant-rich systems exist (Berger et al., 2013b), the main problem is the lack of large ground-truth datasets that challenge the techniques and provide detailed information about the actual evolution, to determine the techniques' performance (e.g., precision or recall). While a few open-source systems annotated with features exist, no dataset provides the whole evolution history of realistic systems, together with the necessary information to use them as a ground truth for the evaluation of methods and tools.

Consider for instance a *feature-location technique* (Rubin and Chechik, 2013). Evaluating it requires a codebase and recorded feature locations, since the latter are typically not recorded in software systems. The intuition is that developers implement features, being fully aware of what feature they implement (since features are part of the design), but they only write code, without recording the feature itself. Feature-location techniques recover such feature locations, relying on input such as code, where they exploit identifier names and other domain information, or they rely on the evolution history. Now, evaluating such techniques requires ground-truth information. Surprisingly, beyond smaller datasets with retroactively added features (Ji et al., 2015; Strüber et al., 2019), no dataset exists that resembles a substantial evolution lifecycle of a variant-rich system. A workaround for researchers was to study the evolution of optional features in software platforms with preprocessor-based variation points, where features are easily identifiable in code (via `#ifdef` annotations). While this strategy helps evaluating preprocessor-focused techniques (e.g., variability-aware type-checking of C code (Kästner et al., 2012)), it misses mandatory features, which are not annotated and which differ from optional features (Krüger et al., 2018); and it misses the early clone&own phases with redundant feature implementations among the cloned variants. Also, preprocessor-annotated code is only available for certain programming languages (e.g., C).

Consider as another example a *change-propagation or variant-synchronization technique* (Pfofe et al., 2016; Strüber et al., 2019). Evaluating it requires the code of the affected variants before and after synchronization, the exact source of the propagation (e.g., a specific feature in a variant), the exact target, and information how the variants were synchronized (e.g., code was merged or overwritten). Similarly to the feature-location example above, if developers would record this information (henceforth called *metadata*) when they evolve a system, this metadata with the codebase and its history could be used as a benchmark to evaluate respective techniques. Unfortunately, we are not aware of any such dataset. Adding the necessary information to an existing system would require clone detection and reverse-engineering of the exact evolution, which is laborious and error-prone, and has only be done for smaller systems (Ji et al., 2015).

In addition to these two examples, many more evolution scenarios and techniques supporting them exist (Strüber et al., 2019). Evaluating and improving them requires evolution histories with metadata to establish ground-truth datasets. Given the lack of substantial benchmarks resembling the evolution of variant-rich systems, researchers often resorted to simple proofs of concept or hand-crafted small datasets.

We strive to improve the situation and advocate the generation of evolution histories with metadata. We present the framework *vpbench* to generate software evolution histories reflecting common evolution scenarios found in variant-rich systems while recording the required metadata—exactly the information that is not recorded by developers in real systems, but that is necessary to evaluate methods and tools for variant-rich systems. Upon an initial codebase, **vpbench simulates the evolution of a variant-rich system** by automatically adding, removing, and cloning features, mutating implementation assets (e.g., code), and cloning variants. Feature addition is realized using automated code transplantation (Barr et al., 2015) from other software projects. We formulate the claims behind vpbench as requirements; intuitively, these express properties of the generator and the generated benchmarks. We show that automatically evolving a variant-rich system and recording the necessary metadata is feasible, where the generated evolution history adheres to the requirements. We show that this automated evolution can rely on a small set of feature-oriented evolution operators, all defined and implemented in an extensible framework.

The key requirements are: The generated evolution histories should (i) resemble the *evolution* of a software system, which should be evolved (ii) in a *feature-oriented* way and in different variants. The generated revisions should (iii) try to be *realistic*. Since complete realism is infeasible to achieve by a synthetic generation process, we defined and addressed three basic levels towards realism reflecting syntactic and semantic properties of the generated system. Finally, (iv) the evolution and especially the high-level intentions behind changes should be documented as *metadata*, e.g., whether a feature is added or not and where exactly. The generator itself should be (v) *extensible* and (vi) *programming-language-independent*.

We contribute:

- **requirements** for our benchmark generation framework;
- the **vpbench framework with a set of generators and evolution operations** to generate evolution histories, including a novel mechanism for **feature transplantation**, which transplants callable functionalities from external projects;
- an **evaluation** demonstrating its generation capabilities with respect to requirements and properties and the different levels of realism;
- an **online appendix** with our code and evaluation data, at bitbucket.org/VPBench/vpbench.

On a final note, research in software configuration management and product line engineering is hindered by a lack of benchmarks. This was a main outcome of a Dagstuhl seminar on variability and evolution (Berger et al., 2019). Without proper benchmarks with ground-truth information, tools cannot be evaluated sufficiently. However, such ground truths do not exist – the baseline for our paper – and recovering them from existing evolution histories is laborious and error-prone and has only been done for smaller-scale systems and specific aspects (Ji et al.,

2015). We address this problem with vpbench and show its capability to generate evolution histories enriched with metadata providing a ground truth.

2. Motivation and background

Evolution of Variant-Rich Systems. Modern software evolves in time and space (Ananieva et al., 2020; Berger et al., 2019). Evolution 'in time' reflects the natural evolution over *system revisions*, including adding new and changing existing code. In contrast, evolution in 'space' reflects the creation of system variants, which co-exist and also evolve themselves. These variants typically share common features (abstract, end-user-visible functional and non-functional aspects (Berger et al., 2015)) while they differ in other (variable) features, which are present only in some variants, and potentially customized towards the variant. Variants can be realized as clones (clone&own (Dubinsky et al., 2013)) or through a configurable platform (explained shortly) (Apel et al., 2013; Berger et al., 2020; Wasowski and Berger, 2023) that integrates all common and variable features in one codebase.

Often, organizations start with clone&own and transition towards a platform when the effort to maintain and evolve the cloned variants explodes (Berger et al., 2013a; Dubinsky et al., 2013; Fogdal et al., 2016; Krüger and Berger, 2020a; Lopez-Herrejon et al., 2022). Establishing a platform requires identifying the features that are part of the individual variants, and then declaring those in a tree-based representation called *feature model* (Berger et al., 2013a; Kang et al., 1990; Nešić et al., 2019). Feature models help developers keep an overview understanding of the platform and are an input to interactive configurator tools that allow deriving concrete variants – those that were migrated or new ones defined by new configurations – in an automated process. While a platform reduces redundancy and substantially shortens the development of new variants, evolving a platform is still challenging, since developers work on many different variants at the same time, and need to keep the feature model consistent with the codebase. A good overview over the challenges of evolving variant-rich systems, with a focus on re-engineering clone variants into a configurable platform is given by Lopez-Herrejon et al. (2022).

Virtual Platform. To simulate software evolution in a feature-oriented way, vpbench relies on the Virtual Platform (VP) (Mahmood et al., 2021), a framework that offers operations which developers can execute to manage and evolve variant-rich systems. vpbench reuses, extends, and automatically executes them to simulate an evolution. An important advantage of the VP is that its operations support a stepwise migration of clone&own-based variants into a configurable platform, reflecting actual evolution scenarios from practice (Krüger et al., 2020). The VP offers two kinds of operations: traditional, asset-oriented ones (e.g., *add asset*, *clone asset*) and novel, feature-oriented ones (e.g., *map asset to feature* or *add feature*), which also evolve a feature model and assure its consistency with the software assets. Operations are applied on the *asset tree*, an abstract-syntax-tree-like system representation inspired by *feature structure trees* (Apel et al., 2009). We introduce these concepts in more detail in the remainder.

Code Transplantation. A core part of software development is the addition of new features. As we strive for full automation, we build on the ideas of automated code transplantation (Barr et al., 2015) to clone whole features among variants—a common evolution scenario in variant-rich systems. Code-transplantation techniques extract some code of interest (the *organ*) from a *donor* system and *implant* it into a *host* system. Existing techniques typically require some type of user input to identify the organ and insertion point (Barr et al., 2015; Lu et al., 2018; Sidiroglou-Douskos et al., 2017; Zhang and Kim, 2017).

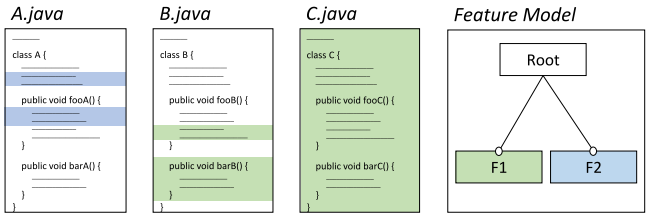


Fig. 1. Running example: initial revision of `V1`.

In our context, existing transplantation techniques are not sufficient and face the following three problems we address with vpbench. The first and the third problems arise from the absence of a user who provides input to the tools, whereas in vpbench we need to automate those steps.

Problem 1: How to identify transplantable features? Features come in many forms and with various facets (Berger et al., 2015) and need to be identified and located manually or automatically before they can be transplanted. This is trivial if the features are explicitly documented (e.g., in feature models), but may be hard if this knowledge is only implicitly contained in the implementation. While automated feature-location techniques exist (Rubin and Chechik, 2013), they are difficult to set up, need feature descriptions as input, and yield too many false positive results to be useful in practice.

Problem 2: How to extract a transplantable feature, i.e., the organ from the donor system? Transplanting a feature requires locating it in code and then not only transplanting the feature code, but also its dependencies. This includes the code that sets up an execution environment for the feature and the code that is called during feature execution, i.e., forward and backward dependencies (Barr et al., 2015).

Problem 3: How to integrate the transplanted feature into the host system? After extracting the organ from the donor system it needs to be integrated with the host system, so that the functionality can be called within the host. This includes identifying insertion points for the organ – which would be provided by the user with existing transplantation techniques – and finding a variable mapping to translate between the donor's and host's execution environments, e.g., variable names or types need to be converted between systems.

3. Running example

To exemplify vpbench's output and explain concepts throughout this paper, we introduce a running example. Let us assume a company developing the software system `V1` depicted in Fig. 1. It initially consists of three files implementing the features `F1` and `F2`. While `F2` is implemented solely in `A.java`, `F1`'s implementation is scattered over `B.java` and `C.java`. As the software is continuously evolving in time, the company is approached by a customer, who wants a similar, yet different variant of `V1`. Specifically, the customer does not require feature `F1`, but wants some additional functionality in feature `F3`. Later, the original customer of `V1` might hear about feature `F3` and request its inclusion into `V1`, too.

Fig. 2 shows on a high level how vpbench could simulate such evolution. It applies feature-oriented operations over a set of iterations and records the operation's type and targets as metadata (shown here schematically and explained in detail in Section 5.3). It also stores feature models, feature locations and clone traces

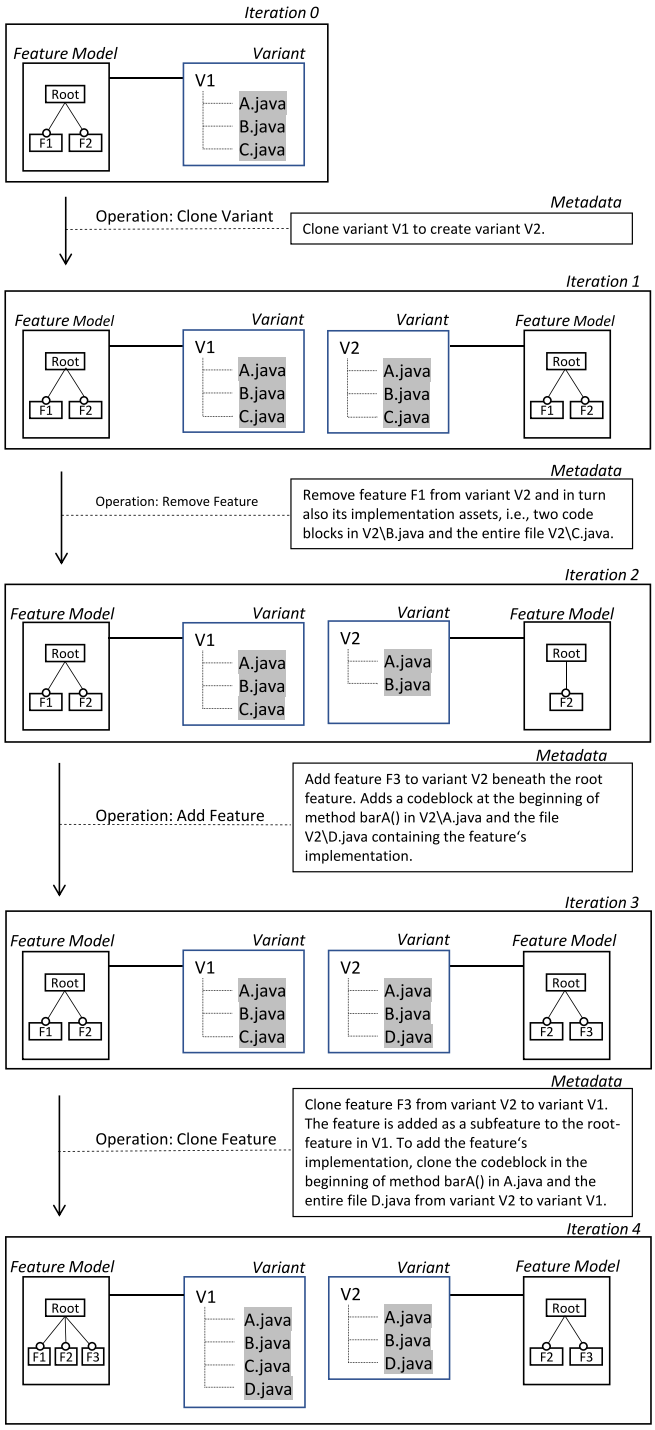


Fig. 2. Running example: evolution of V1 and V2.

between implementation assets (e.g., code) and features (the latter two not depicted for simplicity). Vpbench begins by cloning the existing variant `V1` to create variant `V2`, initially an exact copy (*It. 1*). Feature `F1` is removed by deleting it from the feature model and removing its implementation assets (see Fig. 1), including the entire file `C.java` (*It. 2*). Feature `F3` is added to `V2` as shown in Fig. 3, i.e., by inserting a code snippet into `A.java`, calling the feature code in the also newly added file `D.java`, effectively weaving the feature into the existing program (*It. 3*). Finally, `F3` is cloned into `V1`, including its implementation assets (*It. 4*). Of course, individual evolution in time of the variant might also occur in-between.

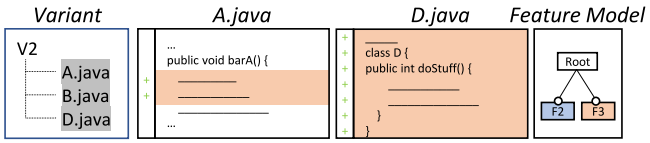


Fig. 3. Running example: adding F3 during iteration 3.

4. Requirements

Our goal is to establish a generation framework for evolution histories useful for benchmarking. It shall automatically simulate the evolution of variant-rich software, while recording detailed metadata about this evolution. We now present the requirements for this framework, which are inspired by existing evolution scenarios from Strüber et al. (2019). We reference the relevant scenarios throughout following the descriptions.

We first present *generation requirements*, which concern the generated evolution histories, and then *framework requirements*, which concern the (re-)usability of our framework.

4.1. Generation requirements

Feature-Orientation. Systems are developed by adding, reusing, removing, and changing features, similar to our running example. In fact, developers have features in their mind when writing code, even if they are typically not made explicit in the code and the evolution history, beyond commit messages. Consequently, the generator should evolve systems in a feature-oriented way, i.e., it should add, remove, change and reuse features across the typical lifecycle of a variant-rich system, comprising clone&own, migrating to a configurable platform, and evolving it.

An explicit representation of features and their evolution is required for several scenarios, such as *variant synchronization* in terms of features, *feature identification and location*, *constraint extraction*, *feature model synthesis*, *analysis of non-functional properties*, *visualization* and *co-evolution of problem space and solution space* (Strüber et al., 2019).

Metadata. The generated evolution history should serve as a ground-truth dataset for evaluating two kinds of tools: (i) tools that automatically extract information necessary for the evolution (to assess the precision of a tool), or (ii) tools that automatically evolve code (to assess the tools' results against the benchmark). Both require information that is typically not recorded during system evolution: feature locations, clone traces, and developer intentions. The latter conveys the high-level feature-oriented change the developer has in mind when changing code.

For example, in iteration 4 in Fig. 2, feature `F3` is cloned from `V2` to `V1` by propagating implementation assets such as `D.java` from `V2` to `V1`. A normal evolution history would only show the changes to the code, but not in which context they were made. This makes it impossible to realize that in fact a feature was cloned between two variants without carefully examining both variants' code.

Recording information on applied operations allows the extraction of concrete evaluation tasks for automatic evolution tools from an evolution history by identifying the point in time where a high-level operation took place, i.e., by filtering for specific high-level intentions (i.e., what operation was done on a feature level), and providing the respective changes in code as a solution to the task given by the simulated high-level intention. This is

required for *variant synchronization* and *integration* and *transformations*, while recorded feature locations and models, and clone traces may additionally support *feature location*, *feature model synthesis*, *visualization*, and *co-evolution of problem and solution space* (Strüber et al., 2019).

Evolution. A development history is a sequence of system snapshots, i.e., revisions, which should evolve considerably over time. We require evolution in both time and space.

This requirement is a necessity to support the benchmarking scenarios (Strüber et al., 2019) *variant synchronization* and *integration*, *transformations*, *test co-evolution* and *co-evolution of problem and solution space*, where tools automatically propose evolution steps. All these scenarios require two system revisions, one containing the problem that tools should solve and one containing its solution to evaluate tool output against. These may be extracted from full evolution histories, where a problem is identified and solved during evolution. System evolution may also support *historical feature location*.

Towards Realism. Achieving full realism with a fully automated generation technique is impossible. Instead, we take first steps towards realism using three basic levels of syntactic and semantic properties that are reasonable for a generation technique: *(1) The first level* reflects *evolution* as described above. *(2) The second level* requires *compilability*, a prerequisite to the stronger, but much more difficult to check executability. While of course, syntax errors might exist in SPLs (Kästner et al., 2008), independently developed variants should always adhere to basic compilability, just as non-variable software systems. *(3) The third level* requires *callability of new features*. Features that are added during system evolution should not be implemented stand-alone as dead code, but be integrated with the already existing system. We say that a feature is integrated, if it is called by the system and is thus *callable*. In our running example, the feature F3 included a code snippet that was added to `A.java` to include the code from `D.java` into the program flow. F3 was integrated into the existing system by being called, so F3 is callable. This is a small, but non-trivial step to improve the realism of our generated evolution histories.

Future work may build on top of these levels to create increasingly realistic evolution histories, e.g., by controlling the type and size of changes using real-world evolution histories or providing executable systems and also enhancing the naturalness of code (Hindle et al., 2016).

4.2. Framework requirements

Extensibility. The framework should be extensible with more mechanisms to simulate evolution (e.g., apply other types of changes or utilize different algorithms) or advance realism to support an increasing set of benchmarking scenarios and tools with growing realism.

Language Independence. The framework should make no assumption regarding specific programming languages or implementation technologies. Any language- or technology-dependent part should be replaceable, allowing to tailor the output to different benchmarking tools.

5. Vpbench

Vpbench takes as input a *configuration*, an external system representing the initial codebase, and multiple external systems serving as feature donors. The configuration guides the simulation of the evolution starting with the initial codebase. The feature donor systems are used to transplant new features into the evolving codebase. Vpbench iteratively applies changes to the input codebase and then serializes its internal system representation (asset tree, explained shortly) to generate the evolution history. Input and output of vpbench are illustrated in Fig. 4. We provide a brief overview of vpbench and follow up with more detailed explanations of each component.

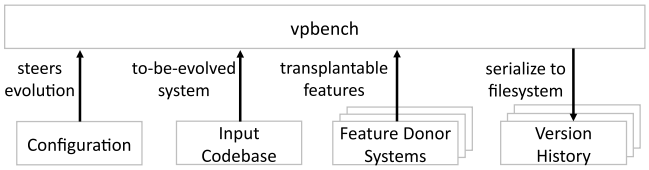


Fig. 4. Vpbench I/O.

Overview. The framework provides both *core modules* and *extensible modules*, as illustrated in Fig. 5. It represents the generated revisions (a codebase with folders and files) of the evolution history internally as an *asset tree* (Section 5.1) and modifies it using dedicated *operations* (Section 6), deterministic procedures automatically applying asset- or feature-oriented evolution steps. Operations may internally call *nested operations* and store *metadata*, detailing the evolution, i.e., which implementation asset or which feature was modified, and how. We call operations that call nested operations *higher-order operations*. These two concepts are reused and extended from the underlying VP (Mahmood et al., 2021).

Applying an operation requires identifying system elements that can sensibly be evolved, first. This task is handled by a set of matching *generators*, each catering to a specific type of operation, such as adding features or mutating implementation assets. Generators scan the entire system to find suitable operation targets, select one, and instantiate operations accordingly. Operations are not immediately executed, but embedded into *transactions*, which may check for the satisfaction of correctness criteria (e.g., compilability) before applying the operation on the evolved system.

This entire process is coordinated by the *runner*. It selects a generator to generate an operation, wraps it into a transaction, executes it and – on success – serializes the resulting system, which may consist of several variants. Through iteration, this creates a sequence of revisions, i.e., the version history. The user can configure the runner and the generators.

5.1. Generated system representation

The generation process outputs a version history in the form of ordered system snapshots , the first one being the input initial codebase. Each snapshot encompasses the structure and code of all cloned and individually maintained variants and includes feature models and feature locations, which are stored in the code as embedded annotations (Schwarz et al., 2020).

Internally, the variant-rich system is represented through assets, features and feature models, all stored within a tree structure (asset tree (Mahmood et al., 2021)). In our context, an asset is anything that gives structure to a software system, from a repository over folders and files to classes, methods and code blocks.

The asset tree keeps structure only to the extent necessary to realize operations, but is otherwise almost fully language-independent. Assets can map to features, which are stored inside of feature models that are associated with elements in the tree. The system is split into different repositories (which represent cloned system variants), all located beneath a synthetic root node.

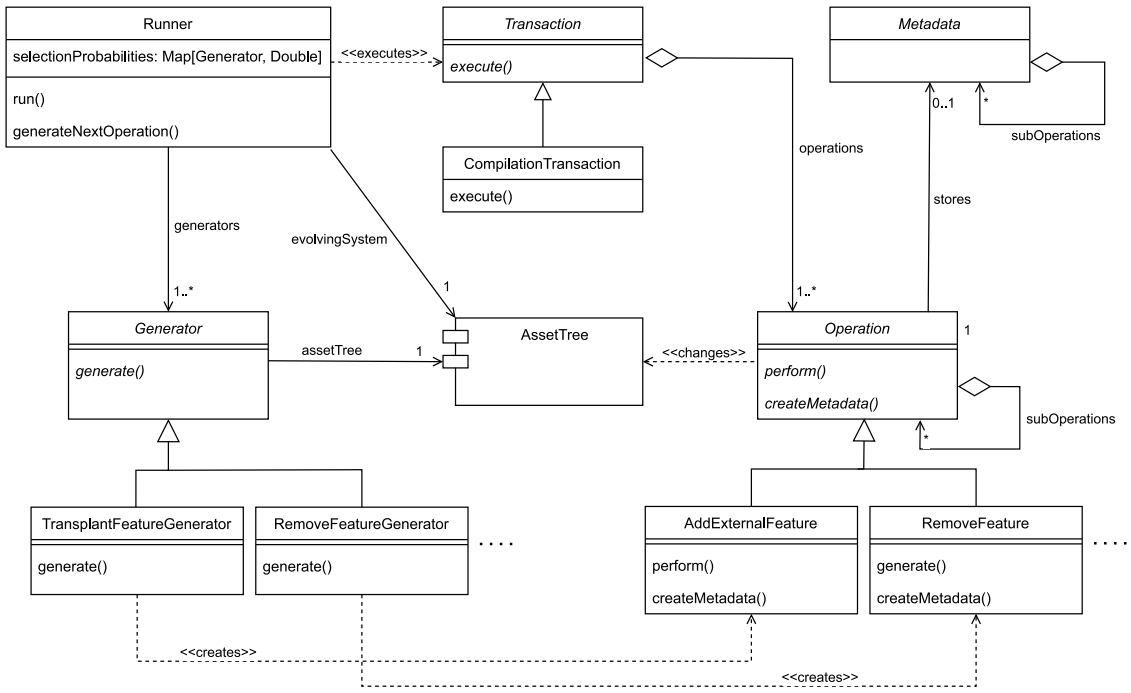


Fig. 5. Vpbench overview.

5.2. Coordinating the evolution

The runner coordinates the simulation process, iteratively delegating change generation to generators, embedding the resulting operations in transactions, and responding to their execution. The generator selection is guided by a probability distribution, assigning a selection probability to each generator. The resulting operation is wrapped and executed in a transaction, checking for compilability of the evolved system and persisting changes only on success. On failure, the runner retries (up to a configurable maximum) the same generator to create another operation. This mechanism is in place, since the generators are typically stochastic, so they might sometimes fail to produce valid changes. In case no valid operation was generated in all attempts, e.g., when querying a feature-deletion generator on an empty feature model, the runner proceeds to query the next generator. These steps are iterated for a user-specified number of iterations.

5.3. Recording metadata

To provide valuable ground truths for different types of problems, our framework provides four types of metadata as part of the generated evolution histories. It records features in a feature model and the feature locations in the asset tree, updating them as operations get executed. An important part of variant-rich system evolution is cloning. Vpbench stores clone traces in a trace database, when elements of the asset tree are cloned as part of operations. Additionally, it stores metadata of the simulated evolution to document the evolution steps and to allow for full replayability. More specifically, we store for each evolution step which operation was executed with what parametrization and which nested operations. Table 1 illustrates the format we use for storing the evolution metadata. We store general metadata about the simulation and code-generation process (`GenerationMetadata`), comprising the list of donor projects used for feature transplantation (`projects`; explained in Section 6.3) and metadata about every single evolution step (`history`, recorded as a list of `IterationMetadata`). Every `IterationMetadata` details one applied evolution step, storing when it was applied (`iterationNumber`) and which `operation` was executed within (stored as `OperationMetadata`). For the `OperationMetadata`, which stores operations' parameters, the involved elements (implementation assets, feature model, feature mappings) need to be uniquely referenced:

Implementation assets are uniquely identifiable via their position in the asset tree, stored as filesystem path and index.

Feature models are referenced using their associated asset, and features via their encompassing feature model and their least-partially-qualified path (Schwarz et al., 2020). Precisely what metadata is stored for each operation type is discussed in Section 6. Since operations are deterministic by definition, storing their parametrizations allows full replayability of the system's evolution. vpbench also stores all nested operations called during the execution of an operation, also to document which low-level changes belong to which high-level change.

5.4. Configuration options

Vpbench allows configuring the coordinating runner and every generator. The runner can be configured using six input parameters: The *number of iterations* defines the maximum number of operations that may be applied on the evolved system. Generation may end earlier if an optional user-specified *exit condition* is met after any iteration. The user can also define which *generators* should be used for the evolution process, and can pass a *probability distribution*, which defines a selection probability for each generator in any iteration (without this parameter, a uniform distribution is assumed). As the selection of generators and the generators themselves are stochastic, they may fail to generate a operation that maintains the compilability of the system on the first attempt. The user can, therefore, define a *maximum number of retries*, i.e., how often a generator may retry to generate an operation that maintains the compilability. Finally, the user can define a *compilation mechanism*, e.g., a build tool, to check whether an operation compiles. We discuss the generator's configuration options in Section 6.

Table 1
Evolution metadata.

GenerationMetadata	:= {projects: List[DonorProject], history: List[IterationMetadata]}
IterationMetadata	:= {iterationNumber: Int, operation: OperationMetadata}
OperationMetadata	:= {operationType: String, parametrization: OperationParametrization, nestedOperations: List[OperationMetadata] }

Table 2
Operation overview.

Operation	Parameters	Metadata
RemoveFeature	featureToRemove: Feature	featureToRemove
MutateAsset		
AddLine	changedAsset: Asset, lineNumber: Int, newLine: String	changedAsset, lineNumber, newLine
ReplaceLine	changedAsset: Asset, lineNumber: Int, newLine: String	changedAsset, lineNumber, newLine
DeleteLine	changedAsset: Asset, lineNumber: Int	changedAsset, lineNumber
TransplantFeature	insertionPoint: Tuple[Asset, Int], donor: DonorProject, testCase: TestCase, parentFeature: Feature, randomSeed: Int	insertionPoint, donor, testCase, parentFeature randomSeed, clonedAssets: Map[String, Asset]
CloneVariant	clonedRepo: Asset, cloneName: String	clonedRepo, cloneName
CloneFeature	clonedFeature: Feature, newParentFeature: Feature	clonedFeature, newParentFeature

6. Generators and operations

System evolution is encoded in operations—evolution patterns of variant-rich systems, e.g., *cloning variants* or *adding, removing, mutating, and cloning features* in our running example (see Section 3). Their execution requires parametrization, e.g., which feature should be cloned where. Determining this is the main task of a generator. It filters the asset tree down to elements on which an operation may sensibly be invoked, selects one, and instantiates the operation.

Every generator specializes in creating a specific operation type, so the set of generators determines the types of changes that may be applied to the evolving system. We provide a base set of five operation types, covering major evolution patterns of variant-rich systems, identified in a simulation study of a clone-based product line (Ji et al., 2015), that can be further extended in the future. Crucial for benchmarking, each operation records metadata, encompassing mostly the operation's parametrization as provided by the generator, but also some additional information on its execution in some cases. We give an overview of this base set of operations, including parametrization and metadata, in Table 2 and explain them with their corresponding generators in-depth in the following.

We use a set of helper functions for our generator explanations: `getAllAssets(asset, assetType)` returns all recursive children of `asset` of type `assetType`. `getAllFeatures(asset, condition)` returns all features that meet a `condition` contained in feature models associated with `asset` or any of its recursive children. The helper function `getContainingFeatureModel(asset)` returns the feature model (potentially recursively) containing `asset`, while `featureModel.getRootFeature()` returns the root feature of `featureModel`. The function `pickRnd(set)` returns a randomly selected element within `set`. Furthermore, `getContainingFolder(asset)` returns the folder asset containing `asset`, `testCase.getProject()` returns the project containing the `testCase`, and `getTestCases(donors)` returns all test cases implemented in any project within the set `donors`. Finally, `getRandomSeed()` returns a random seed for a pseudo-random number generator.

6.1. Removing features

Operation. Existing features might be removed from a variant for different reasons, e.g., due to updated requirements. We model this behavior using an existing operation from VP, called `RemoveFeature`. It removes a selected feature (`featureToRemove`) and its subfeatures from the feature model, also deleting all assets that are solely mapped to the removed features in the process (stored as nested operations), e.g., the entire file `C.java` and both mapped code blocks within `B.java` in our running example (compare Section 3).

Generator. Our generator selects a random non-root feature from the set of all included features over all repositories to invoke the operation. In our running example, any feature within both variants, `V1` and `V2`, could have been selected.

Input: An asset tree *at*
Output: A parametrized *RemoveFeature* operation
Function `generate()`**:**
  rmFeats ← *getAllFeatures*(*at*, *!isRootFeature*);
  featureToRemove ← *pickRnd*(*rmFeats*);
  return *RemoveFeature*(*featureToRemove*);

Algorithm 1: `RemoveFeature Generation`

6.2. Mutating implementation assets

Operation. Implementation assets often evolve without affecting the feature model, e.g., when fixing bugs or refactoring. Generally, all code changes can be described as a sequence of changes to single lines of code, where a single line can be added, deleted or edited (i.e., replaced). We cover such changes by extending the operation `ChangeAsset` from VP with three concrete *suboperations*, each catering to one specific change type. `AddLine` inserts a `newLine` at a specified `lineNumber`, `ReplaceLine` replaces the current `lineNumber` with `newLine`, and `DeleteLine` simply deletes the current `lineNumber`.

Generators. We define three separate generators to mutate assets, each catering to a specific suboperation. Common to all, the generators needs to select a mutable asset and define how to change it. The former is done similarly to the selection of `featureToRemove`: we randomly choose an asset representing a block of code. The latter depends on the concrete operation to generate. In all three cases, we select an edit point, a random `lineNumber` within the selected asset to insert a new line or replace or delete an old one. For adding or replacing, we select a `newLine` of code from any file within the same folder as the selected implementation asset to be edited. This strategy (both operations and generators) is inspired by the program transformations *add-Random*, *replace-Random*, and *delete*, as proposed by Baudry et al. (2014). Compared to the original implementation, our generators work on the line level, rather than the statement level. This makes our generators language-independent.

Input: An asset tree *at*
Output: A parametrized *MutateAsset* operation
Function `generate()`:
  mutAs ← *getAllAssets*(*at*, *codeBlockType*);
  changedAsset ← *pickRnd*(*mutAs*);
  changeAsset(*changedAsset*);
  return;
Function `AddLineGen.changeAsset(a)`:
  lineCount ← *a*.*content*.*size*();
  lineNumber ← *pickRnd*(*lineCount*);
  potLines ← *getPotentialLines*(*a*);
  newLine ← *pickRnd*(*potLines*);
  return *AddLineToAsset(a, lineNumber, newLine)*;
Function `ReplaceLineGen.changeAsset(a)`:
  lineCount ← *a*.*content*.*size*();
  lineNumber ← *pickRnd*(*lineCount*);
  potLines ← *getPotentialLines*(*a*);
  newLine ← *pickRnd*(*potLines*);
  return *ReplaceLineInAsset(a, lineNumber, newLine)*;
Function `DeleteLineGen.changeAsset(a)`:
  lineCount ← *a*.*content*.*size*();
  lineNumber ← *pickRnd*(*lineCount*);
  return *DeleteLineFromAsset(a, lineNumber)*;
Function `getPotentialLines(a)`:
  resultList ← *new List*();
  folder ← *getContainingFolder*(*a*);
  cBlocks ← *getAllAssets*(*folder*, *codeBlockType*);
  foreach *block in cBlocks* **do**
    resultList += *block*.*content*;
  end
  return *resultList*;

Algorithm 2: `MutateAsset` Generation

Since this strategy is fully random, it may create semantically ineffective and syntactically invalid changes. We added sensibility checks to discard some otherwise common, yet ineffective changes, e.g., addition of an empty line, with a parametrized probability (configuration option). Syntactically invalid changes are caught by vpbench's transaction mechanism checking for compilability and are not applied on the evolved system.

6.3. Adding features

Operation. Adding features is one of the most natural ways to evolve software, but poses a complicated problem for automation. While work exists that automatically creates new functionality (Harman et al., 2014), it requires defining test cases and ideally further guidance information. Instead of generating new features, our framework facilitates feature transplantation (Barr et al., 2015; Lu et al., 2018; Sidiroglou-Douskos et al., 2017; Zhang and Kim, 2017) from existing projects. We implement an operation `TransplantFeature` that automatically extracts and integrates a feature from a donor system into the evolved system. As input it requires an `insertionPoint` specified via a location in the asset tree and a `parentFeature` , a `randomSeed` to guarantee that certain parts of the operation remain deterministic as well as some information on the feature itself.

Identifying features. We approximate features using test cases. Similar to Li et al. (2017), we assume test cases to call features to test their functionality. So, a feature for transplantation is identified by a `testCase` in a `donor` system with the actual feature being the unit under test (solving *Problem 1* from Section 2). This only allows identifying features that are actually tested by the donor system, but that are consequently of reasonable quality.

```
public void foo() {
    ...
+   //&begin test_encrypt_EncryptsPlaintext
+   try {
+
+       AesEncryptionStrategy strategy = new AesEncryptionStrategy(
+           128, 1000, "06DC30A48ADEEE72D98E33C2CEAEAD3E",
+           "ED124530AF64A5CAD8EF463CF5628434", "password");
+
+       String ciphertext = strategy.encrypt("Hello world");
+       assertEquals("A/DzjV17WVS6ZAKsLOaC/Q==", ciphertext);
+
+   } catch (Exception e) {}
+   //&end test_encrypt_EncryptsPlaintext
    ...
    return;
}
```

Fig. 6. Transplantation example: achieving callability using test cases (test case from github.com/structurizr/java).

Extracting features. Given a feature to be transplanted, we recursively slice the donor project down to the test case's dependencies to extract the feature, the test case, and dependencies. As test cases build an execution environment (*arrange*) before executing their unit under test (*act*), this provides us with all required code to set up and execute the tested functionality (solving *Problem 2* as specified in Section 2). However, this also means that untested parts of the feature under transplantation are not included in the organ. Slicing is conducted on the level of implementation files to limit the difficulty of the integration step, which would otherwise skyrocket. One notable characteristic of our technique is that required implementation assets, which were already transplanted into a different repository before, may be cloned from there (*configurable on the generator level*). This maintains a sense of continuity and imitates clone&own. This selection is semi-random if the asset exists in multiple variants at the same time, so knowledge of the used random seed keeps this step deterministic.

Integrating features. This slice of the test case's dependencies is added into a separate folder in the target variant and integrated with past transplants from the same donor. While this technically "adds" the feature to the host, it does not integrate it. We ensure that the feature is also called by the host system. This requires us to set up a suitable execution environment and execute the feature within. We reuse the test case's implementation for both. We apply some preprocessing and add it at the specified insertion point, not as a test case, but as "ordinary" code calling a feature (solving *Problem 3* as specified in Section 2). An example for this is given in Fig. 6, showing the insertion of the test case's code for testing plaintext encryption using AES inside the evolving codebase. Note that we do not only extract and implant the test case, but also further dependent code, e.g., the test class' import statements or attributes. Extracting only the source code is not sufficient. We also need to manage its external libraries to achieve compilability. To this end, we transplant not only source code, but relevant build scripts and project structure, too. Build files may introduce a lot of complexity, a lot of which is not needed to achieve basic compilability. So, we slice not only the required code, but also the required build structure down to the basic configuration detailing code dependencies on external libraries. This step needs to be performed once for every donor project, from which we transplant functionality.

All resulting changes are encoded in nested operations, e.g., adding new assets to the asset tree or mapping them to newly introduced features. This is crucial as it allows executing operations on these added features in the future. In contrast to other operations, this operation stores an additional point of metadata, `clonedAssets`, which records which dependency assets were reused from which variants. This provides a quick lookup to users

who want to understand the changes without re-creating them using the stored `randomSeed`.

Generator. The generator is initialized with a set of donor projects, from which we identify annotated test cases using an existing technique (Mukelabai et al., 2021), to randomly select features for transplantation. An insertion point is sampled randomly from the set of functions defined in the original project in a random repository. We do not add features into transplanted code from other feature donor projects to avoid dependency circles. Similarly, we avoid strong feature tangling for now by defaulting the parent features within the selected variant. Test cases are blocklisted on selection, so that transplantation of a test case is only attempted once during a single generation process.

Input: An asset tree *at*, a list of donor projects *donors*, a list of already transplanted test cases *blocklist*, a list of assets *filterInsertion* beneath which the feature may be inserted
Output: A parametrized *TransplantFeature* operation
Function `generate()`**:**
  insertionPoint ← *selectInsPnt*(*at*, *filterInsertion*);
  testCase ← *selectTestCase*();
  donor ← *testCase*.*getProject*();
  parentFeature ← *selectParent*(*insertionPoint*.*getAsset*());
  randomSeed ← *getRandomSeed*();
  return *TransplantFeature(insertionPoint, donor, testCase, parentFeature, randomSeed)*;
Function `selectInsPnt(at, filterInsertion)`**:**
  if *filterInsertion* ≠ *null* **then**
    potAssets ← *getAllAssets*(*filterInsertion*, *methodType*)
  end
  else
    potAssets ← *getAllAssets*(*at*, *methodType*)
  end
  selAsset ← *pickRnd*(*potAssets*);
  potInsIndex ← *selAsset*.*children*.*count*();
  insIndex ← *pickRnd*(*potInsIndex*);
  return *(selAsset, insIndex)*;
Function `selectTestCase()`**:**
  potTestCases ← *new List*();
  foreach *testCase tc of getTestCases*(*donors*) **do**
    if *!blocklist*.*contains*(*tc*) **then**
      potTestCases += *tc*
    end
  end
  testCase ← *pickRnd*(*potTestCases*);
  blocklist += *testCase*;
  return *testCase*;
Function `selectParent(a)`**:**
  fm ← *getContainingFeatureModel*(*a*);
  parentFeature ← *fm*.*rootFeature*;
  return *parentFeature*;

Algorithm 3: `TransplantFeature` Generation

While our generator integrates features into the evolved system by default, i.e., makes them callable, the user may also configure the generator to only guarantee compilability and not callability.

6.4. Clone variant

Operation. Typical variant-rich system evolution begins with clone&own. Existing variants are cloned and developed independently. Given a variant to be cloned (`clonedRepo`) and a name for the new clone (`cloneName`), the VP operation `CloneAsset` creates an exact clone of the source and records a clone trace.

Generation. The corresponding generator randomly selects a variant to clone and generates a name by appending the selected variant's name with a unique id.
Input: An asset tree *at*, a variant counter *id*
Output: A parametrized *CloneVariant* operation
Function `generate()`**:**
  cloneableRepos ← *getAllAssets*(*at*, *repositoryType*);
  clonedRepo ← *pickRnd*(*cloneableRepos*);
  cloneName ← *clonedRepo*.*name*.*append*(*id*);
  id += 1;
  return *CloneVariant(clonedRepo, cloneName)*;

Algorithm 4: `CloneVariant` Generation

6.5. Clone feature

Operation. While variant-rich systems developed using clone& own are typically evolved largely independently, specific functionality might be propagated between variants. Cloning a feature `clonedFeature` and adding it beneath a `newParentFeature` in a different variant requires cloning and integrating assets that implement the feature with the assets already present in the target variant. Depending on whether an asset is already contained in the target or not, we have to solve two different problems: (1.) The asset is already contained in the target, but potentially in a different version. (2.) The asset is not contained in the target, but needs to be integrated (Lillack et al., 2019) with its siblings in the target, that might not exist beneath its parent in the source. This is especially difficult, yet crucial for code level assets. We extend the VP operator `CloneFeature`, which relies on user interaction for integration. For the former problem, we simply keep the version in the target, keeping variant-specific evolution. For the latter, we rely on the recorded clone traces to maintain the partial ordering of common assets between source and target. Cloning a feature from a donor system also requires setting up its build structure as well, similar to our transplantation operator. We perform the same steps if required.

Generator. A generator suggests a clonable feature and a parent feature in a different variant, beneath which the feature should be propagated. As our operation requires clone traces to relate previously cloned elements to each other, this constrains the applicability of this operation to repositories, that originated from each other. Our generator randomly chooses a feature (from a source variant), and defaults a parent feature (from a cloned & owned variant, which does not contain the selected feature). The user can define a maximum feature size (measured via the number of assets mapping to it) that the generator may clone.

7. Implementation

We implemented vpbench in Scala, building it upon the system representation (asset tree) and the evolution operation concept from the VP framework (Mahmood et al., 2021). The implementation effort for vpbench was substantial, especially the integration with Gradle due to its high flexibility and customizability. Overall, it took the main author over six months in full time (still excluding the evaluation).

We especially needed to extend all reused operations from VP so that they record the necessary metadata before operation invocation. They also need to be serialized afterwards, which we implemented using the library lift-json (github.com/lift/framework/tree/master/core/json). Our framework implementation includes the runner, extended or newly created operations and generators,

```
Input: An asset tree at
Output: A parametrized CloneFeature operation
Function generate():
    candidates ← new List();
    foreach CloneVariant-trace repoTrc in TraceDatabase do
        src ← repoTrc.source;
        tgt ← repoTrc.target;
        foreach feature f in getAllFeatures(src, !isRootFeature)
         do
            if f was not cloned between src and tgt then
                candidates += (src, tgt, f);
            end
        end
    end
    selection ← pickRnd(candidates);
    clonedFeature ← selection.getFeature();
    targetRepo ← selection.getTarget();
    targetFM ← targetRepo.featureModel;
    newParentFeature ← targetFM.getRootFeature();
    return CloneFeature(clonedFeature, newParentFeature);
```

Algorithm 5: `CloneFeature` Generation

all of which contain either a language-independent interface or implementation, and a transaction mechanism. In fact, only our implementation for adding and cloning features is not language- and build-tool-independent. Both change and simplify the donor project's build files to automate dependency management. To this end and as a compilation mechanism, we incorporate the build tool Gradle. We believe this does not constrain the applicability of our tool due to Gradle's widespread use in practice providing us with a large set of projects from which functionalities might be transplanted. Our implementation further specializes on Java code, using the dependency analyzer jdeps (docs.oracle.com/javase/9/tools/jdeps.htm) as a base for code slicing. We incorporated the implementation by Mukelabai et al. (2021), which builds on srcML (Maletic et al., 2002), as a mechanism to identify test cases and extended it to extract a selected test case's code dependencies including functions by searching for specific JUnit annotations, e.g., @Before. To implement the transaction mechanism, we also extended a small cloning library (github.com/kostaskougios/cloning).

8. Evaluation

We evaluate vpbench qualitatively and experimentally, addressing the following three questions.

RQ1. *How does vpbench address its requirements?* We provide detailed qualitative arguments based on our design decisions.

RQ2. *Can vpbench generate version histories of evolving, feature-oriented, and variant-rich systems?* We evaluate with real-world projects. We generate seven different version histories from three initial codebases, transplanting features from four open-source repositories we use as donor systems, to investigate vpbench's potential for simulating evolution.

RQ3. *What levels of realism can vpbench achieve?* We examine how well vpbench achieves the three levels of realism (cf. Section 4). We mainly focus on feature transplantation as the most complex operation and the only one having to fulfill callability of new features.

8.1. RQ1: Requirements addressed

Recall that the first four requirements are about properties of the generated version histories, and that the other two are about the generator framework itself.

Feature-Orientation. Vpbench includes a base set of five operation types reflecting evolution patterns identified in a simulation study of a real-world SPL (Ji et al., 2015). We cover the most frequent feature evolution patterns affecting problem and solution space, i.e., *adding or extending a feature* (**P1**), *removing or disabling a feature* (**P2**) and *propagating a feature* (**P8**; we call this *cloning features*) and two vital patterns for simulating clone&own, i.e., *cloning a project* (**P7**) and *evolving annotated assets* (**P9**).

Metadata. Vpbench records four types of metadata: feature models, feature locations, clone traces, and the simulated developer's change intentions. Change intentions are stored via the applied operations, i.e., we record the operation type, its parametrization and the suboperations, invoked during execution. Operation type and parametrization allow understanding the change conceptually, while the suboperations detail how the high-level intentions were realized on a low level, e.g., removing assets as part of removing a feature.

Evolution. Vpbench evolves an initial codebase by incrementally applying operations to it. These affect problem and solution space, adding, cloning and removing features including their implementing code, and also provide mechanisms to mutate code assets on their own (on a line-based level). These mechanisms evolve systems in both size and variability. We further evaluate this experimentally in RQ2 (see Section 8.2).

Towards Realism. As already discussed for *Evolution* and *Feature-Orientation*, vpbench evolves a given codebase by adding features and changing existing code (first level). Building up on that, the second level (compilability) is ensured by design. Through usage of a dedicated transaction type, we enforce that only changes resulting in a compiling system are applied. The third level (callability of new features) is achieved by transplanting not only the feature, but also inserting feature-executing code into the initial codebase. We examine this level further in RQ3 (see Section 8.3) and discuss avenues for extending realism in the next paragraph.

Extensibility. Vpbench is published as a framework, providing two main avenues for extension: operation & generator pairs, and transactions to improve on vpbench's generation capabilities and correctness criteria, i.e., different forms of realism. Operations can be reused from the available set of VP's operators, covering both conventional development activities and variability-specific ones, or implemented from scratch by extending the abstract operation class. The latter can also be helpful for realizing more complex changes, such as our approach to feature addition, which we implemented as a new operation encompassing multiple other operations. An important property of operations is determinism for a given parametrization to allow for their complete traceability through recorded metadata. Existing generators may be improved or new ones added by extending a Scala trait, giving access to the asset tree and requiring only an implementation of the generate-function to return an operation when called.

Transactions constrain applicable operations to achieve correctness criteria, e.g., compilability of the generated system. Stronger constraints such as executability, satisfaction of clean code properties or even accordance to a behavior specification may be implemented by extending the provided transaction class, supervising the operations' execution.

Language Independence. Vpbench includes concepts and implementations for five operations and corresponding generators.

While two implementations cater to a specific language and build tool, the general concepts do not. To further alleviate this, we provide language-independent implementation skeletons and interfaces, as well as a checklist summarizing where our implementation is language- or build-tool-dependent and thus where it needs to be changed:

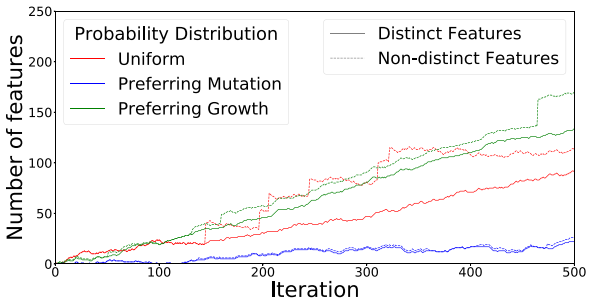
- **VP parser**: While the asset tree itself is language-independent it needs to be converted into this format. To this end, a parser needs to be created that parses a file in a given programming language into an implementation asset.
- **Compilation Mechanism**: We use Gradle to validate whether a system compiles. To support a different build tool, the compilation mechanism needs to be adjusted.
- **Feature Transplantation**: Supporting feature transplantation for a different language requires (i) a different slicing technique (we use an existing dependency analyzer for Java and srcML (Maletic et al., 2002) to extract the test case's implementation) and potentially (ii) some preprocessing for code integration (e.g., in Java we need to add an additional import statement to the test case's package so that the code can compile). Gradle is mainly required to (iii) copy, adapt, and simplify the donor's required build files and alter the initial system's build structure to achieve a compiling system. We also rely on Gradle during slicing to (iv) map identified dependencies to their implementation location in the donor system. All four points need to be adjusted to support different programming languages and build tools.
- **Cloning Features**: Cloning features is similar to transplanting them as you do not need to just deal with code, but with build files, too. We do not map excerpts of build files to features, so we cannot clone them in the same language- and build-tool-independent way that we clone code. Instead, we manually (i) identify which parts of the donor systems (i.e., Gradle (sub-)projects) need to be initialized and (ii) alter the initial system's build files, similar to our strategy during feature transplantation. Both need to be revised to support different build tools.

8.2. RQ2: Simulating evolution

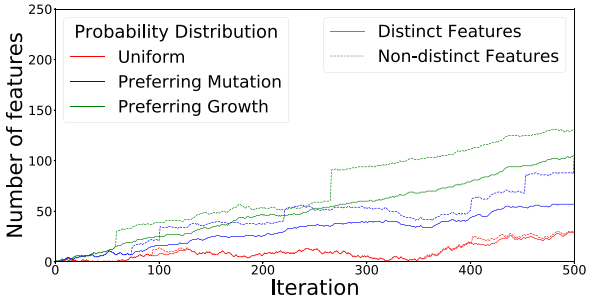
We assess whether we can actually generate versions that evolve in time and space. While changes to existing code are assured by respective operators, we quantitatively investigate whether the number of variants or the variability (evolution in space), as well as the system size increase over time.

Setup. We generate version histories for three different initial systems over 500 iterations using different parameterizations. The selected initial systems are a small calculator example with 62 LoC, an open-source json-parser for Java (github.com/stleary/JSON-java) with 11,837 lines of code (LoC) and a library for event-based programming in Java (github.com/ReactiveX/RxJava). For the latter two we cloned the repository and applied some small preprocessing steps, detailed in our replication package.

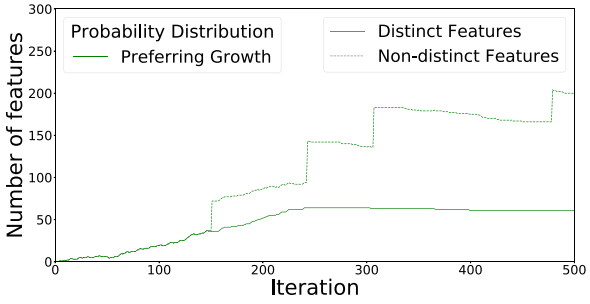
We simulate the evolution of these initial systems using three different probability distributions. In all three cases we set a selection probability of $p = 0.01$ to both cloning generators due to scalability issues in memory consumption and runtime. The remaining probability is split up in the following ways: a uniform distribution (*Uniform*) over the operations *TransplantFeature*, *RemoveFeature* and *MutateAsset* (splitting up the probability on all three generators), a distribution prioritizing mutation over other operations (*Preferring Mutation*), i.e., giving all mutation generators the same probability as feature addition and removal, and a distribution prioritizing system growth (*Preferring Growth*). The latter selects each asset mutation generator with a probability of $p = 0.2$, adds a new feature with $p = 0.29$, and removes a feature with $p = 0.09$. We evolve the first two systems using all three probability distributions and the third one using only the last distribution.



(a) Calculator



(b) JSON-java



(c) RxJava

Fig. 7. Evolution of variability over first 500 iterations.

A selected generator has 50 attempts to generate a compiling change before the runner proceeds to the next generator. Mutation generators discard ineffective changes with a probability of $p = 0.5$. We use four different feature donor systems from GitHub for transplantation: the Structurizr client library (github.com/structurizr/java), the HPC inter-thread messaging library LMAX Disruptor (github.com/LMAX-Exchange/disruptor), an educational library of algorithm- and datastructure-implementations (github.com/williamfiset/Algorithms) and the dex to java decompiler jadx (github.com/skylot/jadx).

We screen all four donors for callable features (cf. Section 8.3 for details) and use this subset as input for transplantation. This preselection is done for performance reasons only, as it reduces the retries required to find a transplantable feature. It does not improve the generation output in any way.

Evolution in Space. Fig. 7 shows the evolution of the amount of features as a measurement of variability inside the generated systems over time. We display both the number of distinct features

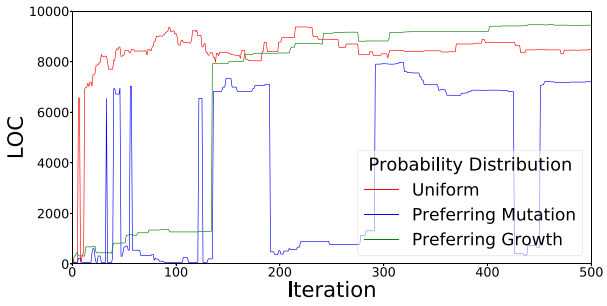
and the number of non-distinct features, i.e., features existing in different variants and thus potentially versions. The probabilities for adding and removing features are the same for *Uniform* and *Preferring Mutation*. As such, features that are added might quickly be removed again, resulting in a constantly evolving low-variability system. While this behavior can be well observed in the early phases of the two former examples, the amount of distinct features starts growing fairly steady at some point. The reason for this is that variants and their included features are getting cloned at some point, requiring all feature clones to be removed to delete the entire feature, making the complete removal of features from the entire variant space less likely than the addition of new ones. Constant addition and removal of features can instead be seen in the number of non-distinct features, which stays typically similar, apart from larger differences, when a new variant is cloned (comp. Fig. 7(b) from iteration 200). Finally, *Preferring Growth* adds features more frequently than it removes them, building up a solid feature base quickly and resulting in a higher variability after 500 iterations in both cases were we used all three probability distributions. It is noticable that while the first 250 iterations of the RxJava evolution bear similarity to the other two systems, no additional features get added from this point. This is due to the fact that a large portion of transplantable features for the other two initial systems stem from the *Algorithms* donor system. For *RxJava* the same transplantations were attempted, but none of these compiled, indicating compatibility issues between the two systems.

Evolution in Size. The evolution of the size of the generated versions can be seen in Fig. 8. We visualize the evolution in lines of code (LoC) of the initial repository, that was read in as the initial codebase. All evolution histories contain large changes in LoC. The reason for this is that our coarse slicing approach identifies a large number of dependencies for a significant portion of the transplanted features, which are added along with the test case, if they are currently not present in the system. This can result in code additions of multiple thousands of LoC for a single feature. We observed this behavior especially within test cases transplanted from the structurizr donor project. Of course, once the dependencies of a feature and respective test case are added, they will be reused by other features from the same donor with similar dependencies. Consequently, less dependency code needs to be added by the following transplantations, easing the evolution.

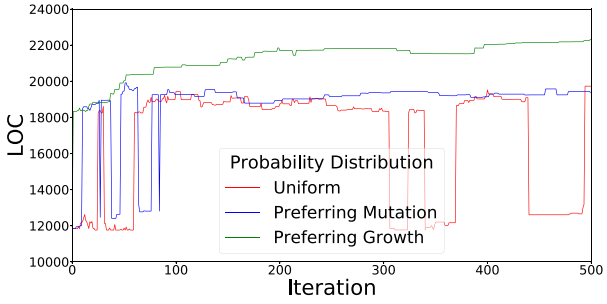
However, removing the only feature mapping to a large set of dependencies deletes not only the feature, but all dependencies, too. This results in a sharp cutback in code. This happens especially frequently for *Uniform* and *Preferring Mutation*, as these add and remove features with equal probability.

While this can also happen for *Preferring Growth* (compare Fig. 8(c)), it achieves a smooth evolution more consistently as it adds more features with similar dependencies faster and can thus more reliably safeguard large dependency chunks from being removed again. Similar to above, the evolution of *RxJava* differs from the other two, as no new features could be added to the system starting from evolution step 241.

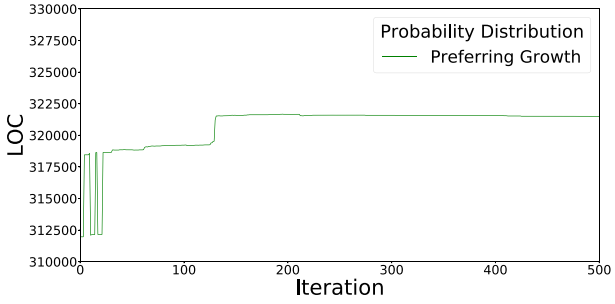
Our results show that vpbench is capable of evolving a variant-rich system in both variability and size over time. As expected, depending on the used probability distributions over generators, systems are evolved in very different ways. It is also important to note that enough donor systems should be compatible with the initial system to ensure the addition of new features throughout the entire evolution.



(a) Calculator



(b) JSON-java



(c) RxJava

Fig. 8. Evolution of system size over first 500 iterations.

8.3. RQ3. Levels of realism

The first level is achieved for the generated version histories, as shown in our experimental results presented in Section 8.2. Also, all generation results compile without exception.

Callability of new features can sensibly only be handled by our operation-generator-pair for feature addition. Thus, we focus on this mechanism and examine how often our approach succeeds in achieving transplantation fulfilling different realistic properties: from pure transplantation without compilability and callability to both.

Setup. We use the same GitHub repositories as in Section 8.2, preprocess and build them, and extract all features using our proposed approach. For preprocessing, we delete all multi-line comments within the projects' test classes due to a bug in our used test case identification tool and remove one task from the Structurizr projects' main build file, which caused compilation to fail otherwise.

All features are then inserted into the same predefined location in a basic dependency-free system. The experiment is split up in three phases, testing the different properties: We check for which test cases (1.) the transplantation terminates without throwing an exception, (2.) the transplantation results in

Table 3
Transplantable features.

Donor	#Total		#Compiled		#Callable	
	Class	Case	Class	Case	Class	Case
Structurizr	71	799	50	498	17	122
Disruptor	27	97	27	97	4	8
Algorithms	70	521	70	521	46	250
Jadx	446	623	440	595	3	7

a compiling system, (3.) the transplantation additionally succeeds in making the feature callable inside the evolving system. Each consecutive step in the pipeline takes as input all test cases that passed the previous step. The first two steps select only one test case per test class, as our implementation prototype shows no test case- but only test class-specific behavior for transplantation without callability.

Assessing Applicability. Table 3 shows the experiment's results. We omitted the results of the first evaluation step (passing without an exception) from the table, as not a single case failed here (i.e., the results being the same as #Total). The majority of test cases can be transplanted to produce compiling systems, showing the general usability of the technique. Raising the targeted realism degree to transplanting callable functionality poses a much more difficult problem. While we received good results in two cases (Structurizr and Algorithms), only a small percentage of available test cases was convertible into callable features in the other two projects.

Open Problems. We manually examined a small set of test cases failing to insert both compiling and callable features, to understand which problems our approach can currently not solve. We identified two issues for compilability and three issues for callability: For compilability, our implementation does not support the entire flexibility of Gradle build scripts, e.g., we support only two out of three main dependency types, and our dependency analysis failed to identify dependencies that were only defined in annotations. For callability, our implementation fails to extract some types of code from the test classes, e.g., attributes initialized during declaration, non-annotated methods and in general code from other test classes. A more conceptual problem is package-private members being accessed by test cases, as the test case's code is moved to a different package.

While our prototype does not solve all problems, most identified ones (apart from accessing package-private members) are solely implementation-related and can be solved to support even more test cases. Even so, it transplants compiling features from over 80% and callable features from almost 20% of all test cases of all four donors, showing the technique's general feasibility. In fact, these numbers may be higher in practice, as at least dependency adding would become a non-issue if the missing dependencies had already been added in an earlier transplantation.

Limitations for Feature Transplantation. The constraints that exist for our proposed form of feature transplantation differ based on the level of abstraction. Most exist on the level of our concrete implementation, which is limited to transplanting features via test cases between Java systems using Gradle as a build tool. We discussed some further implementation issues in the previous subsection. However, on a conceptual level we pose less requirements. The generator algorithm introduced in Section 6.3 is not constrained to specific programming languages of host and donor systems and might even support multilingual transplantation (Marginean, 2021) in the future. The sole constraint that remains on the level of the presented `TransplantFeature` generator is the existence of test cases for features. On the level of vpbench as a whole, this constraint might be less important, since we may implement new generators using alternative forms of feature transplantation (not utilizing test cases). Thus, on a framework level we only require the existence of donor projects (as input) for system evolution. We allow and welcome improvements and extensions on all abstraction levels.

8.4. Threats to validity

External validity. Our evaluation considers only one programming language and build tool and evolves only three initial systems, restricting external validity. While experimentally exploring a broader selection of programming languages and build tools would be desirable, vpbench is by no means specific to Java and Gradle. In fact, our framework has been designed to be conceptually language-independent and only two modular generators need to be reimplemented to support different languages or build tools. We showed that all our generators can generate valid changes on both a toy example and two different-sized real-world systems. This includes transplanting functionality from four donor systems, stemming from different domains.

Internal validity. Our technique relies on a number of parameters. In our evaluation, we observed that the configuration (as explained in Section 5.4) and specifically the used probability distribution strongly affects the plausibility of the generated version histories. While we found a configuration that leads to plausible outcomes, these parameters have to be tuned every time a new generator is available. Guiding this tuning process systematically is a desirable direction for future work.

Construct validity. Our operationalization of realism relies on a set of common activities during the evolution of variant-rich systems (Ji et al., 2015) and qualitative understanding of basic code quality measures (*towards realism*). Future work might extend the implemented levels of realism as discussed in Section 8.1 or assess vpbench's output against real version histories by guiding the parametrization of our technique using a comparable activity distribution found in real projects. Second, our evaluation of vpbench's usefulness is solely based on a set of requirements, derived from an existing set of benchmark scenarios. An actual application of our generated systems to different benchmarking scenarios and tools is out of scope of this work, but subject to future work.

9. Related work

Benchmark Generation. A plethora of work on system generation for benchmarking purposes exists. Techniques typically follow one of three strategies: (i) generating a system from scratch (Jasper et al., 2019; Mendonca et al., 2009; Segura et al., 2012; Steffen et al., 2014a,b; Wägemann et al., 2017), (ii) modifying a given initial system (Bui et al., 2007; Kashyap et al., 2019; Nassar et al., 2020; Szárnyas et al., 2018; Varró et al., 2018; Weiss et al., 2013; Wu, 2018; Zhu et al., 2007), and (iii) reproducing a given system in a different way (Jasper et al., 2019; Martinez et al., 2018; Richards et al., 2011; Van Ertvelde and Eeckhout, 2010; Wang and Provan, 2010). Vpbench fits the second category. While covering a wide set of domains including variability-related ones, none of the identified techniques include historical information.

We identified only two approaches actively incorporating version histories. Michelon et al. (2021) extract feature revisions from version histories of C preprocessor SPLs to be identified in preprocessed variants. And, closest to our work, Schultheiß et al. (2022) propose VEVOS, a tool deriving version histories

of clone&own based systems from real SPLs' version histories. Their benchmarks include source code, feature models, configurations, locations and clone tracing. Both approaches are extractive, i.e., they draw benchmarks from existing version histories, in this case of SPLs. The system's variants' evolution is directly mapped from the evolution of the entire SPL. While this accesses realistic code bases, it sacrifices the validity of the evolution, i.e., the independence of clone&own based systems (Schultheiß et al., 2020), the focus of VEVOS (Schultheiß et al., 2022). Vpbench focuses on more variety in evolution, allowing for variant drift (Schultheiß et al., 2020) and the addition of new features through transplantation, and takes first steps to towards achieving realism. Finally, due to their focus on the task of *feature location*, Michelon et al. (2021) only provide such as metadata. VEVOS (Schultheiß et al., 2022) aims at larger benchmark applicability, but crucially misses information detailing the evolution itself, i.e., the changes applied by the developer.

Code Transplantation. Automated code transplantation is a rather recently established branch of research. It was first proposed in 2013 (Harman et al., 2013) and realized in 2015 with the tool μScalpel (Barr et al., 2015). μScalpel extracts an annotated organ and its execution environment for it, i.e., the *vein*, using program slicing, and adapts and implants it at a user-specified insertion point using genetic programming. It validates the operation's success through user-defined test cases. Similarly, CodeCarbonCopy (Sidiroglou-Douskos et al., 2017) requires the user to provide the organ and insertion point. It limits its applicability to programs working on the same input type to convert data representations between host and donor code. It extracts the specified functionality using a compile-time dependency graph and inserts it at the given insertion point. Lu et al. (2018) propose a search-based way of adding new functionality. Grafter (Zhang and Kim, 2017) enables test reuse between code clones through transplantation. It replaces a tested piece of code with its clone using five transplantation rules guaranteeing compilability on termination. Finally, PatchWeave (Shariffdeen et al., 2020) tackles the patch transplantation problem on two similar programs. It utilizes the donor's version history to extract a bugfix and find an insertion point by identifying change locations in the donor and relating them to the erroneous host.

Other approaches to feature transplantation rely on user input (Barr et al., 2015; Lu et al., 2018; Sidiroglou-Douskos et al., 2017; Zhang and Kim, 2017) or similarity of host and donor systems (Shariffdeen et al., 2020; Sidiroglou-Douskos et al., 2017; Zhang and Kim, 2017). Our novel approach does neither, apart from requiring donor and host to use the same build tool. We note however, that this degree of automation is only possible in the context of simulating software evolution as proposed.

10. Conclusion

We presented vpbench, a framework for the generation of version histories. It aims to lift the maturity of current and future methods and tools for evolving variant-rich systems. It simulates the evolution of a variant-rich system while proactively recording metadata. It uses an extensible set of generators, emulating the execution of evolution tasks, including simple ones (e.g., changing assets, deleting features) and much more advanced ones (e.g., adding features through transplantation). We discussed vpbench's design with regards to requirements useful for benchmarking purposes, and evaluated its generation capabilities by generating seven evolution histories evolving three initial codebases. Vpbench contributes an important step towards a consolidated benchmarking infrastructure.

Valuable future work is to extend vpbench's generation capabilities to further improve its realism. Specifically, we plan to leverage more modern code-generation techniques, such as those based on novel language models. Using such models to generate even more consistent more comprehensible code with a better domain-orientation of the feature model, can go a long way. Especially the feature model together with its features and their organization in a hierarchy is highly domain-specific. Having long been the sole responsibility of a domain expert, we believe that such modern language models can substantially enhance the generation.

CRediT authorship contribution statement

Christoph Derks: Conceptualization, Software, Validation, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Daniel Strüber:** Conceptualization, Methodology, Resources, Writing – original draft, Writing – review & editing, Supervision. **Thorsten Berger:** Conceptualization, Methodology, Resources, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A replication package including code and evaluation data is available and linked in the paper

References

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C., 2006. Refactoring product lines. In: GPCE.
Ananieva, S., Greiner, S., Kühn, T., Krüger, J., Linsbauer, L., Grüner, S., Kehrer, T., Klare, H., Koziolek, A., Lönn, H., Krieter, S., Seidl, C., Ramesh, S., Reussner, R., Westfechtel, B., 2020. A conceptual model for unifying variability in space and time. In: SPLC.
Apel, S., Batory, D., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Science & Business Media.
Apel, S., Kästner, C., Lengauer, C., 2009. FEATUREHOUSE: Language-independent, automated software composition. In: ICSE.
Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A., 2017. Reengineering legacy applications into software product lines: A systematic mapping. 22, (6), pp. 2972–3016,
Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J., 2015. Automated software transplantation. In: ISSTA.
Baudry, B., Allier, S., Monperrus, M., 2014. Tailored source code transformations to synthesize computationally diverse program variants. In: ISSTA.
Berger, T., Chechik, M., Kehrer, T., Wimmer, M., 2019. Software evolution in time and space: Unifying version and variability management (dagstuhl seminar 19191). In: Dagstuhl Reports. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik.
Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K., 2015. What is a feature? A qualitative study of features in industrial software product lines. In: SPLC.
Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A., 2013a. A survey of variability modeling in industrial practice. In: VaMoS.
Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K., 2013b. A study of variability models and languages in the systems software domain. IEEE Trans. Softw. Eng. 39 (12), 1611–1640.
Berger, T., Steghöfer, J.-P., Ziadi, T., Robin, J., Martinez, J., 2020. The state of adoption and the challenges of systematic variability management in industry. Empir. Softw. Eng. 25, 1755–1797.
Bui, N.B., Zhu, L., Gorton, I., Liu, Y., 2007. Benchmark generation using domain specific modeling. In: ASWEC.
Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K., 2013. An exploratory study of cloning in industrial software product lines. In: CSMR.
Fogdal, T., Scherrebeck, H., Kuusela, J., Becker, M., Zhang, B., 2016. Ten years of product line engineering at danfoss: Lessons learned and way ahead. In: SPLC.

Harman, M., Jia, Y., Langdon, W.B., 2014. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In: SSBSE.
Harman, M., Langdon, W.B., Weimer, W., 2013. Genetic programming for reverse engineering. In: WCRE.
Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131.
Jasper, M., Mues, M., Murtovi, A., Schlüter, M., Howar, F., Steffen, B., Schordan, M., Hendriks, D., Schiffelers, R., Kuppens, H., Vaandrager, F.W., 2019. RERS 2019: Combining synthesis with real-world models. In: TACAS.
Jepsen, H.P., Beuche, D., 2009. Running a software product line: standing still is going backwards. In: SPLC.
Jepsen, H.P., Dall, J.G., Beuche, D., 2007. Minimally invasive migration to software product lines. In: SPLC.
Ji, W., Berger, T., Antkiewicz, M., Czarnecki, K., 2015. Maintaining feature traceability with embedded annotations. In: SPLC.
Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21.
Kashyap, V., Ruchti, J., Kot, L., Turetsky, E., Swords, R., Pan, S.A., Henry, J., Melski, D., Schulte, E., 2019. Automated customized bug-benchmark generation. In: SCAM.
Kästner, C., Apel, S., Thüm, T., Saake, G., 2012. Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. (TOSEM) 21 (3), 1–39.
Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., Batory, D., 2008. Language-independent safe decomposition of legacy applications into features. Tech. Rep. 2, School of Computer Science, University of Magdeburg, Germany.
Krüger, J., Berger, T., 2020a. Activities and costs of re-engineering cloned variants into an integrated platform. In: VaMoS.
Krüger, J., Berger, T., 2020b. An empirical analysis of the costs of clone- and platform-oriented software reuse. In: FSE.
Krüger, J., Gu, W., Shen, H., Mukelabai, M., Hebig, R., Berger, T., 2018. Towards a better understanding of software features and their characteristics: A case study of marlin. In: VaMoS.
Krüger, J., Mahmood, W., Berger, T., 2020. Promote-pl: A round-trip engineering process model for adopting and evolving product lines. In: SPLC.
Kuiter, E., Krüger, J., Krieter, S., Leich, T., Saake, G., 2018. Getting rid of clone-and-own: Moving to a software product line for temperature monitoring. In: SPLC.
Li, Y., Zhu, C., Rubin, J., Chechik, M., 2017. FHistorian: Locating features in version histories. In: SPLC.
Lillack, M., Stanciulescu, S., Hedman, W., Berger, T., Wąsowski, A., 2019. Intention-based integration of software variants. In: ICSE.
Lopez-Herrejon, R.E., Martinez, J., Assunção, W.K.G., Ziadi, T., Acher, M., Vergilio, S., 2022. Handbook of Re-Engineering Software Intensive Systems into Software Product Lines. Springer Nature.
Lu, Y., Chaudhuri, S., Jermaine, C., Melski, D., 2018. Program splicing. In: ICSE.
Mahmood, W., Strüber, D., Berger, T., Lämmel, R., Mukelabai, M., 2021. Seamless variability management with the virtual platform. In: ICSE.
Maletic, J.I., Collard, M.L., Marcus, A., 2002. Source code files as structured documents. In: IWPC.
Marginean, A., 2021. Automated Software Transplantation (Ph.D. thesis). UCL (University College London).
Martinez, J., Ziadi, T., Papadakis, M., Bissyandé, T.F., Klein, J., le Traon, Y., 2018. Feature location benchmark for extractive software product line adoption research using realistic and synthetic eclipse variants. Inf. Softw. Technol. 104, 46–59.
Mendonca, M., Wąsowski, A., Czarnecki, K., 2009. SAT-based analysis of feature models is easy. In: SPLC.
Michelon, G.K., Obermann, D., Assunção, W.K.G., Linsbauer, L., Grünbacher, P., Egyed, A., 2021. Managing systems evolving in space and time: Four challenges for maintenance, evolution and composition of variants. In: SPLC.
Mukelabai, M., Berger, T., Borba, P., 2021. Semi-automated test-case propagation in fork ecosystems. In: ICSE-NIER.
Nassar, N., Kosiol, J., Kehrer, T., Taentzer, G., 2020. Generating large EMF models efficiently. In: FASE.
Nešić, D., Krüger, J., Stănciulescu, S., Berger, T., 2019. Principles of feature modeling. In: FSE.
Pfofe, T., Thüm, T., Schulze, S., Fenske, W., Schaefer, I., 2016. Synchronizing software variants with variantsync. In: SPLC.
Pohl, K., Böckle, G., Van Der Linden, F., 2005. Software Product Line Engineering, vol. 10. Springer.
Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: A systematic review. Inf. Softw. Technol. 55 (7), 1165–1199.
Richards, G., Gal, A., Eich, B., Vitek, J., 2011. Automated construction of JavaScript benchmarks. In: OOPSLA.
Rubin, J., Chechik, M., 2013. A survey of feature location techniques. In: Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (Eds.), Domain Engineering: Product Lines, Languages, and Conceptual Models. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-36654-3, pp. 29–58.
Schlie, A., Schulze, S., Schaefer, I., 2020. Recovering variability information from source code of clone-and-own software systems. In: VaMoS.
Schultheiß, A., Bittner, P.M., El-Sharkawy, S., Thüm, T., Kehrer, T., 2022. Simulating the evolution of clone-and-own projects with VEVOS. In: EASE.
Schultheiß, A., Bittner, P.M., Kehrer, T., Thüm, T., 2020. On the use of product-line variants as experimental subjects for clone-and-own research: a case study. In: SPLC.
Schulze, S., 2019. Analysis techniques to support the evolution of variant-rich software systems (Ph.D. thesis). Habilitationsschrift, Magdeburg, Otto-von-Guericke-Universität Magdeburg, 2019.
Schwarz, T., Mahmood, W., Berger, T., 2020. A common notation and tool support for embedded feature annotations. In: SPLC.
Segura, S., Galindo, J.A., Benavides, D., Parejo, J.A., Ruiz-Cortés, A., 2012. Betty: Benchmarking and testing on the automated analysis of feature models. In: VaMoS.
Shariffdeen, R.S., Tan, S.H., Gao, M., Roychoudhury, A., 2020. Automated patch transplantation. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (1), 1–36.
She, S., Ryssel, U., Andersen, N., Wąsowski, A., Czarnecki, K., 2014. Efficient synthesis of feature models. Inf. Softw. Technol. 56 (9), 1122–1143, Special Sections from "Asia-Pacific Software Engineering Conference (APSEC), 2012" and " Software Product Line conference (SPLC), 2012".
Sidiroglou-Douskos, S., Lahtinen, E., Eden, A., Long, F., Rinard, M., 2017. CodeCarbonCopy. In: FSE.
Steffen, B., Howar, F., Isberner, M., Naujokat, S., Margaria, T., 2014a. Tailored generation of concurrent benchmarks. Int. J. Softw. Tools Technol. Transfer 16 (5), 543–558.
Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M., 2014b. Property-driven benchmark generation: synthesizing programs of realistic structure. Int. J. Softw. Tools Technol. Transfer 16 (5), 465–479.
Strüber, D., Mukelabai, M., Krüger, J., Fischer, S., Linsbauer, L., Martinez, J., Berger, T., 2019. Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems. In: SPLC.
Szárnyas, G., Izsó, B., Ráth, I., Varró, D., 2018. The train benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. 17 (4), 1365–1393.
Van Ertvelde, L., Eeckhout, L., 2010. Benchmark synthesis for architecture and compiler exploration. In: IISWC.
Van Gurp, J., Bosch, J., Svahnberg, M., 2001. On the notion of variability in software product lines. In: Proceedings Working IEEE/IFIP Conference on Software Architecture. IEEE, pp. 45–54.
Varró, D., Semeráth, O., Szárnyas, G., Horváth, Á., 2018. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In: Heckel, R., Taentzer, G. (Eds.), Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig. Springer International Publishing, Cham, ISBN: 978-3-319-75396-6, pp. 285–312.
Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M., 2015. Evolution of software in automated production systems: Challenges and research directions. J. Syst. Softw. 110, 54–84.
Wägemann, P., Distler, T., Eichler, C., Schröder-Preikschat, W., 2017. Benchmark generation for timing analysis. In: RTAS.
Wang, J., Provan, G., 2010. A benchmark diagnostic model generation system. IEEE Trans. Syst. Man Cybern. A Syst. Hum. (ISSN: 1558-2426) 40 (5), 959–981.
Wasowski, A., Berger, T., 2023. Domain-Specific Languages: Effective Modeling, Automation, and Reuse. Springer, URL http://dsl.design.
Weiss, C., Westermann, D., Heger, C., Moser, M., 2013. Systematic performance evaluation based on tailored benchmark applications. In: ICPE.
Wu, H., 2018. Step 0: An idea for automatic OCL benchmark generation. In: STAF.
Zhang, T., Kim, M., 2017. Automated transplantation and differential testing for clones. In: ICSE.
Zhu, L., Bui, N.B., Liu, Y., Gorton, I., 2007. MDABench: Customized benchmark generation using MDA. J. Syst. Softw. 80 (2), 265–282.

Christoph Derks is a Ph.D. student at Ruhr University Bochum. His research interests are in software transplantation, variant-rich systems and software evolution.

Daniel Strüber is a senior lecturer at Chalmers | University at Gothenburg, Sweden, and an assistant professor at Radboud University in Nijmegen, the Netherlands. His research interests are in model-driven engineering, AI engineering, and variant-rich systems. He was awarded his doctoral degree from Philipps University Marburg, Germany, and worked as a post-doctoral researcher at University of Koblenz and Landau, Germany, and Gothenburg University, Sweden. He is a co-author of over 80 papers with six Best Paper Awards. He has been a Program Committee member of several leading conferences, including ASE, FASE, MODELS. He is the lead developer of Henshin, a model transformation language used in more than 15 countries.

Thorsten Berger is a Professor in Computer Science at Ruhr University Bochum in Germany. After receiving the Ph.D. degree from the University of Leipzig in Germany in 2013, he was a Postdoctoral Fellow at the University of Waterloo in Canada and the IT University of Copenhagen in Denmark, and then an Associate Professor jointly at Chalmers University of Technology and the University of Gothenburg in Sweden. He received competitive grants from the Swedish Research Council, the Wallenberg Autonomous Systems Program, Vinnova Sweden (EU ITEA), and the European Union. He is a fellow of the Wallenberg Academy—one of the highest recognitions for researchers in Sweden. He received two best-paper and two most-influential-paper awards. His service was recognized with distinguished reviewer awards at the tier-one conferences ASE 2018 and ICSE 2020. His research focuses on model-driven software engineering, program analysis, and empirical software engineering.