Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems

Daniel Strüber[1], Mukelabai Mukelabai[1], Jacob Krüger[2], Stefan Fischer[3], Lukas Linsbauer[3], Jabier Martinez[4], Thorsten Berger[1]
[1]Chalmers | University of Gothenburg, Sweden, [2]University of Magdeburg, Germany, [3]JKU Linz, Austria, [4]Tecnalia, Spain

ABSTRACT

The evolution of variant-rich systems is a challenging task. To support developers, the research community has proposed a range of different techniques over the last decades. However, many techniques have not been adopted in practice so far. To advance such techniques and to support their adoption, it is crucial to evaluate them against realistic baselines, ideally in the form of generally accessible benchmarks. To this end, we need to improve our empirical understanding of typical evolution scenarios for variant-rich systems and their relevance for benchmarking. In this paper, we establish eleven evolution scenarios in which benchmarks would be beneficial. Our scenarios cover typical lifecycles of variant-rich system, ranging from clone & own to adopting and evolving a configurable product-line platform. For each scenario, we formulate benchmarking requirements and assess its clarity and relevance via a survey with experts in variant-rich systems and software evolution. We also surveyed the existing benchmarking landscape, identifying synergies and gaps. We observed that most scenarios, despite being perceived as important by experts, are only partially or not at all supported by existing benchmarks—a call to arms for building community benchmarks upon our requirements. We hope that our work raises awareness for benchmarking as a means to advance techniques for evolving variant-rich systems, and that it will lead to a benchmarking initiative in our community.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software evolution**;

KEYWORDS

software evolution, software variability, product lines, benchmark

ACM Reference Format:
Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3336294.3336302

1 INTRODUCTION

Evolving a variant-rich software system is a challenging task. Based on feature additions, bugfixes, and customizations, a variant-rich system evolves in two dimensions: (1) in its variability when new variants are added over time, and (2) in each individual variant, as variants are continuously modified. From these dimensions, various evolution scenarios arise. For example, variability may be managed using clone & own [25], that is, by copying and modifying existing variants. In this case, changes performed on one variant are often propagated to other variants (*variant synchronization*). When the number of variants grows, a project initially managed using clone & own might be migrated to an integrated product-line platform [8, 13, 50], comprising a variability model [19, 38] and implementation assets with variability mechanisms (e.g., preprocessor annotations or composable modules). In this case, all assets in all variants that correspond to a given feature must be identified (*feature location*). Supporting developers during such scenarios requires adequate techniques, many of which have been proposed in recent years [2, 3, 7, 8, 10, 20, 27, 29, 37, 39, 48, 50, 60, 72, 77, 79, 87, 90, 94, 95].

The maturity of a research field depends on the availability of commonly accepted benchmarks for comparing new techniques to the state of the art. We define a benchmark as *a framework or realistic dataset that can be used to evaluate the techniques of a given domain.* Realistic means that *the dataset should have been initially created by industrial practitioners; it may be augmented with meta-data that can come from researchers.* In the case of evolving variant-rich systems, despite the progress on developing new techniques and tools, evaluation methodologies are usually determined ad hoc. To evaluate available techniques in a more systematic way, a common benchmark set has yet to emerge.

Inspired by a theory of benchmarks in software engineering [91], we believe that the community can substantially move forward by setting up a common set of benchmarks for evaluating techniques for evolving variant-rich systems. With this goal in mind, we follow typical recommendations for benchmark development [91]: to lead the effort with a small number of primary organizers, to build on established research results, and to incorporate community feedback to establish a consensus on the benchmark. As such, our long-term goal is to establish a publicly available benchmark set fulfilling the requirements of successful benchmarks [91]: clarity, relevance, accessibility, affordability, solvability, portability, and scalability.

In this paper, as a step towards this long-term goal, we lay the foundations for a benchmark set for evaluating techniques for evolving variant-rich systems. We conceive the scenarios that the benchmark set needs to support, show the relevance and clarity of our descriptions based on community feedback, and survey the state of the art of related datasets to identify potential benchmarks.

We make the following contributions:

- Eleven scenarios for benchmarking the techniques that support developers when evolving variant-rich systems (Sec. 2), including sub-scenarios, requirements, and evaluation metrics;
- A community survey with experts on software variability and evolution, focusing on the clarity and relevance of our scenarios (Sec. 3) and relying on an iterative, design-science approach;
- A survey of existing benchmarks for the scenarios (Sec. 4), selected upon our experience and the community survey;
- An online appendix with further information (e.g., links to benchmarks) and a replication package with the questionnaire and its data: https://bitbucket.org/easelab/evobench/

We observed that various scenarios are only partially or not at all supported by existing benchmarks. We also identified synergies between scenarios and available benchmarks, based on the overlap of required benchmarking assets. Based on the positive feedback regarding the clarity and relevance of our benchmark descriptions, we believe that our work paves the way for a consolidated benchmark set for techniques used to evolve variant-rich systems.

2 EVOLUTION SCENARIOS

We establish eleven scenarios for techniques that support developers during the evolution of variant-rich systems. For each scenario, we argue how the relevant techniques can be evaluated with a benchmark. We introduce each scenario with a description, a list of more detailed sub-scenarios, a list of requirements for effective benchmarks, and a list of metrics for comparing the relevant techniques.

2.1 Methodology

To select the scenarios and construct the descriptions, we followed an iterative process involving all authors. We took inspiration from our experience as experts in software product line research, our various studies of evolution in practice [12, 13, 15, 17, 34, 35, 37, 42, 54, 56, 59, 67, 73, 74], and the mapping studies by Assunção et al. [8] and Laguna and Crespo [48]. Based on these sources, an initial list of scenarios emerged in a collaborative brainstorming session. Each scenario was assigned to a responsible author who developed an initial description. Based on mutual feedback, the authors refined the scenario descriptions and added, split, and merged scenarios and their descriptions. Each scenario description was revised by at least three authors. Eventually, a consensus on all scenario descriptions was reached. Afterwards, we performed a community survey to assess the clarity and relevance of the descriptions. The final version of the descriptions, as shown below, incorporates feedback from the survey (see the methodology description in Sec. 3).

2.2 Running Example

As a running example for the evolution of variant-rich systems, consider the following typical situation from practice.

Initially, a developer engineers, evolves, and maintains a single system, for instance, using a typical version-control system (e.g., Git). At some point, a customer requests a small adaptation. The developer reacts by adding a configuration option and variation points (e.g., based on `if` statements) in the code. Later, another customer requests a more complex adaption. The developer reacts by copying the initial variant (i.e., creating a clone) of the system and adapting it to the new requirements (a.k.a., clone & own). Over time, further customers request specific adaptations and the developer uses either of these two strategies.

When the number of variants grows, this ad hoc reuse becomes inefficient. Namely, it becomes challenging and error-prone to identify which existing variant to clone and which parts (i.e., features) of other variants to incorporate in the new variant. The same applies to maintenance, as it is not clear which variants are affected by a bug or update. Any bug or update then needs to be fixed for each existing variant individually. Furthermore, an increasing number of configuration options challenges developers through intricate dependencies that need to be managed; and variation points clutter the source code, challenging program comprehension.

2.3 Scenario Descriptions

We now introduce our scenarios based on the running example, providing descriptions, sub-scenarios, benchmarking requirements and evaluation metrics. We focus on evaluation metrics that are custom to the scenario at hand. Some additional characteristics of interest, such as performance and usability, are important in *all* scenarios and should be supported by adequate metrics as well. Assessing the correctness or accuracy of a technique may require a *ground truth*, a curated, manually produced or (at least) checked set of assets assumed to be correct. Some scenarios involve the design choice of picking a metric from a broader class of metrics (e.g, similarity metrics); in these cases we specify only the class.

We visualize each scenario by means of a figure. Each figure provides a high-level overview of the respective scenario, representing the involved assets with boxes, techniques with rounded boxes, relationships with dashed arrows, and actions with solid arrows. In cases where a scenario has multiple sub-scenarios with varying kinds of assets, we show the superset of all required assets from all sub-scenarios. Each figure includes a large arrow on its left-hand side, indicating the direction of system evolution.

Variant Synchronization (VS). When evolving a variant-rich system based on clone & own, the developer frequently needs to *synchronize variants*. Bugfixes or feature implementations that are performed in one variant need to be propagated to other variants—a daunting task when performed manually. An automated technique (illustrated in Fig. 1) could facilitate this process by propagating changes or features contained in a variant [77, 78].

Sub-scenarios

- VS1: Propagation of changes across variants
- VS2: Propagation of features across variants

Benchmark requirements

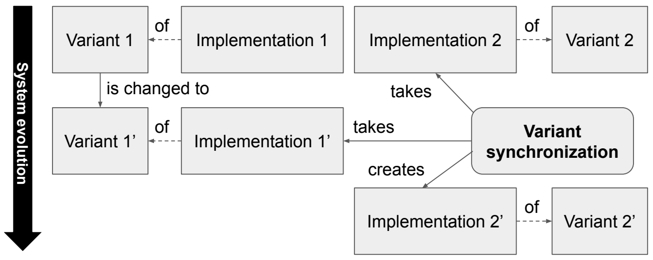

Figure 1: Variant synchronization (VS)

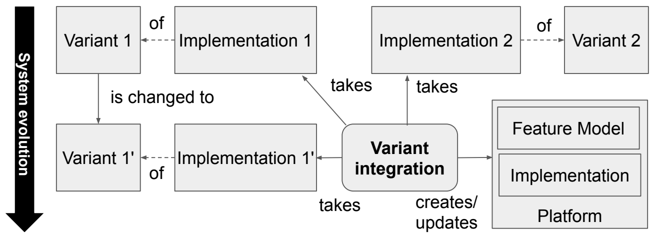

Figure 2: Variant integration (VI)

- VS1/2: Implementation code of two or more variants
- VS1/2: Implementation code of variants after correct propagation *(ground truth)*
- VS1: Changes of at least one variant
- VS2: Feature locations of at least one variant

Evaluation metrics

- Accuracy: A metric for measuring the similarity between ground truth and computed variant implementation

Variant Integration (VI). Due to the drawbacks associated with clone & own [6, 25], a developer may deem it beneficial to manage the variant-rich system as a product-line platform. Such a platform comprises a variability model (e.g., feature [38] or decision model [19]) and implementation assets with a variability mechanism (e.g., preprocessor annotations or feature modules) that supports the on-demand generation of product variants. From the decision to move towards a product-line platform, two major *variant integration* tasks (a.k.a., extractive product-line adoption [43]) arise (illustrated in Fig. 2).

The first task is to enable the transition from the cloned variants to a platform [8]. Available techniques for this purpose take as input a set of products and produce as output a corresponding product-line platform [60]. Yet, further evolving the resulting platform can be challenging due to its variability—assets may be difficult to comprehend and modify. Therefore, the second task is to support extending and evolving a product line by means of individual, concrete product variants [51, 94]. This allows engineers to focus on concrete products during evolution to then feed the evolved product back into the platform to evolve it accordingly. Such techniques can be supported by variation control systems [51, 94] and approaches for incremental product-line adoption [6] from cloned variants.

Sub-scenarios

- VI1: Integrate a set of variants into the product-line platform
- VI2: Integrate changes to variants into the product-line platform

Benchmark requirements

- VI1: Set of individual variants
- VI2: Set of revisions of a product-line platform
- VI1/2: Product-line platform after correct integration *(ground truth)*

Evaluation metrics

- Accuracy: A metric for measuring the similarity between the ground truth and the computed product-line platform

Feature Identification and Location (FIL). Both, as an aid to better support clone & own development and to prepare the migration to a product-line platform, developers may wish to determine which features exist in the system and which features are implemented in which assets (e.g., source code, models, requirements or other types of artifacts). For this purpose, they may rely on *feature identification* and *feature location* techniques (illustrated in Fig. 3). Feature identification aims to determine which features exist, whereas feature location aims to define the relationship of features to assets.

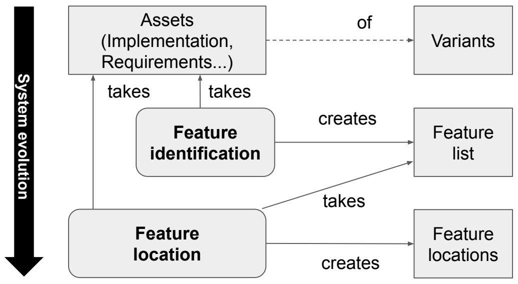

Figure 3: Feature identification and location (FIL)

Feature identification is useful when the knowledge about features is only given implicitly in the assets, rather than explicitly as in a feature model. The objective is to analyze assets to extract candidate feature names. This can involve techniques to study domain knowledge or vocabulary of the considered domain, workshops to elicit features from experts [42], or automated techniques [61, 70, 100].

When done manually, feature location is a time-consuming and error-prone activity [45]. It has a long tradition for maintenance tasks (e.g., narrowing the scope for debugging code related to a feature), but is also highly relevant for identifying the boundaries of a feature at the implementation level to extract it as a reusable asset during re-engineering [47]. In this sense, it is related to traceability recovery. Feature location is usually expert-driven in industrial settings, however, several techniques based on static analysis, dynamic analysis, and information retrieval, or hybrid techniques, exist [8].

Sub-scenarios

- FIL1: Feature identification in single variants
- FIL2: Feature identification in multiple variants
- FIL3: Feature location in single systems
- FIL4: Feature location in multiple variants

Benchmark requirements

- FIL1/2/3/4: Assets representing variants, such as: implementation code, requirements, documentation, issue tracker data, change logs, version-control history
- FIL1/2/3/4: List of features (*ground truth* for FIL1/2)
- FIL3/4: Feature locations in sufficient granularity, such as files, folders, code blocks (*ground truth*)

Evaluation metrics

- Accuracy: Precision and Recall. Some authors in the literature use metrics, such as Mean Reciprocal Rank, that assess the accuracy of a *ranking* of results [18, 99].

Constraints Extraction (CE). In a variant-rich system, some features may be structurally or semantically related to other features. Initially, this information is not explicitly formalized, which makes it harder for the developer to understand these relationships. To this end, the developer may use an automated *constraints extraction* technique (illustrated in Fig. 4).

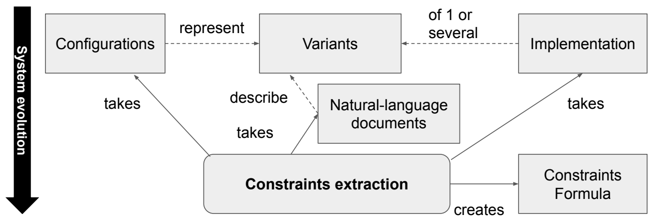

Figure 4: Constraints extraction (CE)

Constraints extraction is a core prerequisite for feature-model synthesis. However, even if the goal is not to obtain a model, explicitly knowing the constraints can help checking the validity of platform configurations, reducing the search space for combinatorial interaction testing (CIT, see below), and documenting features with their dependencies. The benchmark can be used to evaluate the extraction of constraints from various inputs, specifically, the product-line implementation (either code of individual variants or of a platform, [68, 69]), a set of example configurations [22], or natural-language artifacts, such as documentation. Over the development history, when a feature model exists, the constraints in the feature model would be annotated with their source (e.g., a def-use dependency between function definition and function call or domain dependency from hardware [69]). Considering cloned systems, constraints extraction can also be helpful to compare the variability that is implemented in different variants.

Sub-scenarios
- CE1: Constraints extraction from example configurations
- CE2: Constraints extraction from implementation code
- CE3: Constraints extraction from natural-language assets

Benchmark requirements
- CE1: Example configurations
- CE2: Implementation code of one or several variants
- CE3: Natural-language assets (e.g., documentation)
- CE1/2/3: Correct constraints formula *(ground truth)*

Evaluation metrics
- Accuracy: Similarity of configuration spaces (likely syntactic approximation; semantic comparison is a hard problem)

Feature Model Synthesis (FMS). To keep an overview understanding of features and their relationships, developers may want to create a feature model. *Feature model synthesis* (illustrated in Fig. 5) is an automated technique that can provide an initial feature model candidate. As input, it can rely on a given set of configurations, a set of variants (together with a list of features that each variable implements) or a product matrix to produce a feature model from which these assets can be derived.

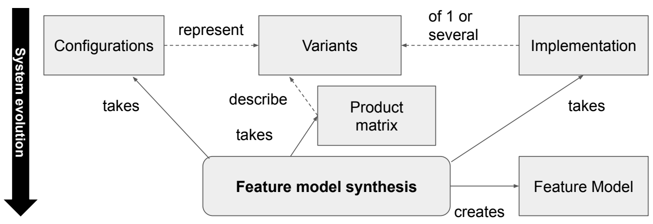

Figure 5: Feature model synthesis (FMS)

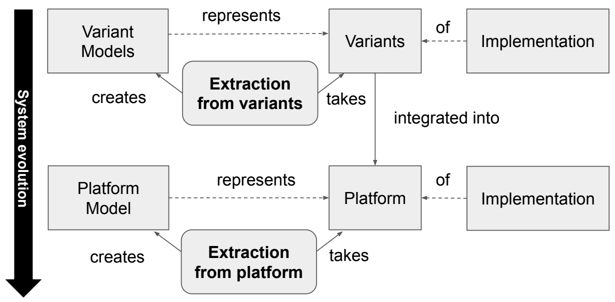

Figure 6: Architecture recovery (AR)

Various synthesis techniques [3, 86–88] are available. Their primary benefit is to identify a possible feature hierarchy, but they can also identify feature groups. Constraints extraction (CE, see above) can be incorporated as a component to identify constraints.

Sub-scenarios
- FMS1: Feature model synthesis from a set of configurations
- FMS2: Feature model synthesis from an implementation
- FMS3: Feature model synthesis from a product matrix

Benchmark requirements
- FMS1: Example configurations
- FMS2: Implementation code of one or several variants
- FMS3: Product matrix
- FMS1/2/3: Correct feature model *(ground truth)*

Evaluation metrics
- Accuracy: Precision and Recall of recovered hierarchy edges and feature groups; similarity of the configuration spaces represented by the synthesized feature model and the input

Architecture Recovery (AR). When migrating cloned variants to a product-line platform, the developer may want to define a reference architecture for the resulting platform, using architectural models. Architectural models provide a different abstraction of the system structure than feature models, focusing on details and dependencies of implemented classes. *Architecture recovery* techniques (illustrated in Fig. 6) can extract architectural models automatically.

Various works [26, 41, 84, 92] focus on reverse engineering and comparing architectures from cloned variants to propose architectural models as a starting point for product-line adoption. Such models can include class, component, and collaboration diagrams that may be refined later on. For instance, the initial models may be used as input for a model-level variant integration technique, producing a platform model with explicit commonality and variability. Additional use cases include analyzing and comparing models to identify commonality and variability, or performing an automated analysis based on models.

Sub-scenarios
- AR1: Architecture extraction from a configurable platform
- AR2: Architecture extraction from a set of variants

Benchmark requirements
- AR1: Implementation code of one or several variants
- AR2: Implementation code of product line platform
- AR1/2: Correct architectural models *(ground truth)*

Evaluation metrics
- Accuracy: Similarity of extracted to ground truth models

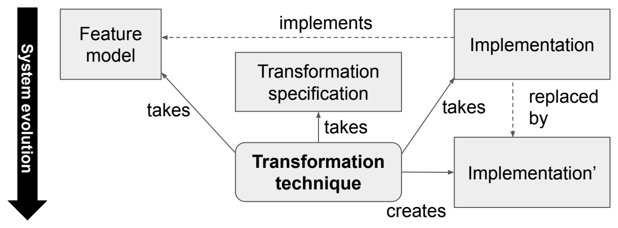

Figure 7: Transformations (TR)

Transformations (TR). To reduce manual effort during evolution tasks, such as refactoring or synchronization of multiple dependent assets in a variant-rich system, the developer may rely on *transformation* techniques. Transformation techniques are used to change system assets in an automated way. Tool support ranges from lightweight refactoring tools in IDEs to advanced model transformation languages with dedicated execution engines. Model transformations are used for manifold practical purposes, including translation, migration, and synchronization of assets [55].

When transforming a product-line platform (illustrated in Fig. 7), three sub-scenarios arise: First, to refactor the platform, improving its structure while behavior preservation is ensured for each variant [82]. Second, to partially refactor the platform [72] in such a way that only a controlled subset of all variants is changed. Third, to lift a given transformation from the single-product case to the platform, changing all variants consistently [80].

Sub-scenarios

- TR1: Refactoring of a product-line platform
- TR2: Partial refactoring of a product-line platform
- TR3: Lifting of a model transformation to a product-line platform

Benchmark requirements

- TR1/2: Product-line platform with feature model and implementation code
- TR3: Product-line platform with feature model and implementation model
- TR1/2/3: Transformation specification; for example, reference implementation
- TR1/2/3: Transformed implementation *(ground truth)*

Evaluation metrics

- Correctness: Number of errors
- Conciseness: Number of elements or lines of code of the given transformation

Functional Testing (FT). After evolving the variant-rich system, it is important to ensure it still behaves in the expected way. For instance, the variants that were available before the evolution should

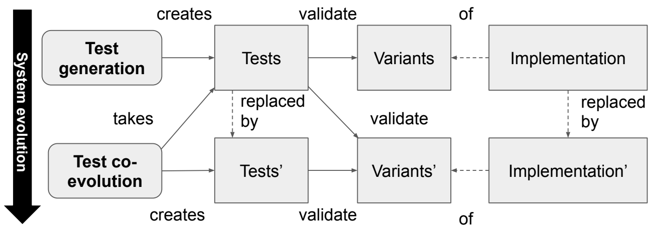

Figure 8: Functional testing (FT)

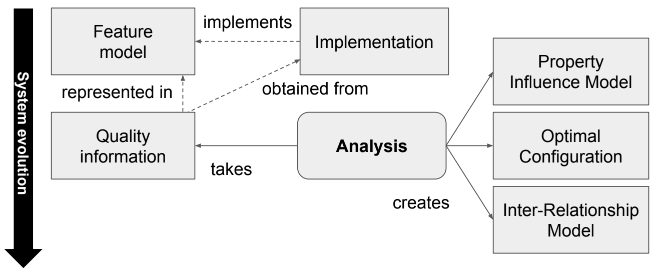

Figure 9: Analysis of non-functional properties (ANF)

still work after evolving the system. Regression testing aims to identify faults that may arise after the system has been changed and functionality does no longer work as before. *Functional testing* of variable software (illustrated in Fig. 8) adds challenges compared to conventional software testing, due to the variability that can influence the functionality of the variants.

For a product-line platform, we can divide testing into two phases: First, *domain testing* of common parts of the system. Second, *application testing* of variant-specific parts and interactions [24, 49]. In the case of clone & own, we can only do application testing for individual variants. To reduce testing effort, existing techniques aim to reuse test assets as much as possible. Assets from domain testing are reused in application testing, while trying to only test parts that are specific to selected variants to avoid redundancies. Similarly, it is useful to avoid redundancies after the evolution of the system, to test only parts relevant for the changes that have been applied. Moreover, for application testing it is unrealistic to test all possible variants. The most common technique used for the selection of variants is Combinatorial Interaction Testing (CIT), which identifies a subset of variants where interaction faults are most likely to occur, based on some coverage criteria [23]. Finally, evolution potentially makes some test cases outdated, because they no longer fit the evolved system. In such cases, system and tests must co-evolve [44].

Sub-scenarios

- FT1: Test generation for domain testing
- FT2: Test generation for application testing
- FT3: Test co-evolution

Benchmark requirements

- FT1/2/3: Implementation code from product line platform
- FT1/2/3: Known faults *(ground truth)*
- FT3: Tests to be co-evolved

Evaluation metrics

- Efficiency: Number of faults detected in relation to number of known faults
- Test effort: Number of tested variants, number of executed tests, execution time of tests only if all tests are executed on the same system, reuse of test assets

Analysis of Non-Functional Properties (ANF). Various non-functional or quality properties can be important for variant-rich systems, for example, performance in a safety-critical system [33], memory consumption in an embedded system with resource limitations [32], and usability aspects in human-computer interaction systems [58]. Therefore, the *analysis of non-functional properties*

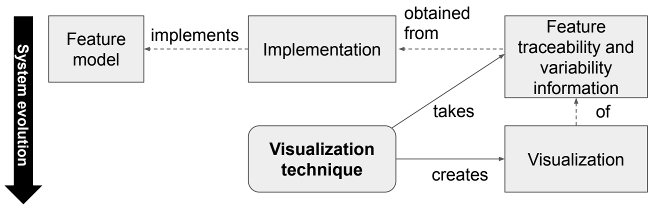

Figure 10: Visualization (VZ)

in variant-rich systems (illustrated in Fig. 9) is crucial [67], as constraints on non-functional properties can be violated when the system evolves.

Developers would like to know the effect of specific features and feature interactions on the investigated quality property, particularly to identify possible improvements or regressions when changes were introduced. Such effects can be captured using a *property influence model* for the quality property under study, for instance, a performance influence model in the case of Siegmund et al. [89]. Also, an important analysis scenario is to identify *optimal configurations* that maximize one or multiple quality criteria while satisfying certain quality constraints [90]. This analysis is relevant for evolution when trying to balance various conflicting quality properties and understanding their relationships and trade-offs [76]. To this end, an *inter-relationship model* can be derived by analyzing the pareto front obtained during multi-criteria optimization. The considered analyses can be expensive, not only because of the combinatorial explosion in large systems, but also because computing non-funcional properties can be a resource-intensive task.

Sub-scenarios

- ANF1: Analysis of impacts of features and feature interactions on quality properties
- ANF2: Optimization of configurations towards given quality criteria
- ANF3: Analysis of trade-offs between relationships among non-functional properties

Benchmark requirements

- ANF1/2/3: Feature model
- ANF1/2/3: Quality information, either given by annotations (e.g., extended feature models [11]), or by a method to calculate or estimate for a given product the quality metrics under study
- ANF1: Reference property influence model *(ground truth)*
- ANF2: Reference configuration *(ground truth)*
- ANF3: Reference inter-relationship model *(ground truth)*

Evaluation metrics

- Accuracy: Similarity between computed and reference model (ANF1/3), fitness of computed configuration in comparison to reference configuration (ANF2)

Visualization (VZ). To facilitate incremental migration [6] of clone & own-based variants to a product-line platform, the developer may want to visually inspect relations between features and implementation assets. Such a relation-visual inspection can be provided by *visualization* techniques (illustrated in Fig. 10).

During product-line engineering, visualizing variability in software assets can be useful for scenarios, such as product configuration [71, 76], testing (e.g., pairwise testing) [53], and constraint discovery [62]. Andam et al. [5] propose several feature-oriented views that exploit feature annotations [37] embedded by developers in the source code during development for tracing feature locations. A benchmark could be used to evaluate the effectiveness of several visualization techniques addressing the same sub-scenario. The main goal of benchmarking is to assess developer performance when using different techniques, which requires experimentation with human participants on selected development tasks.

Sub-scenarios

- VZ1: Visualizations for feature evolution and maintenance
- VZ2: Visualizations for constraint discovery
- VZ3: Visualizations for feature interaction assessment

Benchmark requirements

- VZ1/2/3: Implementation code with feature locations (preferably embedded feature traceability annotations, instead of only variability annotations for optional parts of source code)
- VZ1/2/3: Scenario-related tasks for developers, such as code comprehension and bug-finding tasks, based on generated visualizations

Evaluation metrics

- Developer performance: correctness, completion time in scenario-related tasks

Co-Evolution of Problem Space and Solution Space (CPS). After migrating the variant-rich system to a product-line platform and to further evolve it, the developer has to evolve both, the problem space (feature model) and the solution space (assets, such as architecture models and code). Evolving the solution space first can lead to outdated feature models that are inconsistent with the implementation. Evolving the problem space first limits the effects that changes to the implementation are allowed to have. To address these issues, an automated technique (illustrated in Fig. 11) may recommend *co-evolution steps* to keep both in sync.

For instance, when evolving the solution space first, the technique could extract updated feature dependencies (e.g., an additional dependency on another feature) based on their modified implementation (e.g., due to an additional method call) and suggest modifications to the problem space that reflect the changes made to the solution space. An important property is that problem space and solution space are consistent after every evolution step.

Sub-scenarios

- CPS1: Co-evolving the solution space based on problem space evolution
- CPS2: Co-evolving the problem space based on solution space evolution

Benchmark requirements

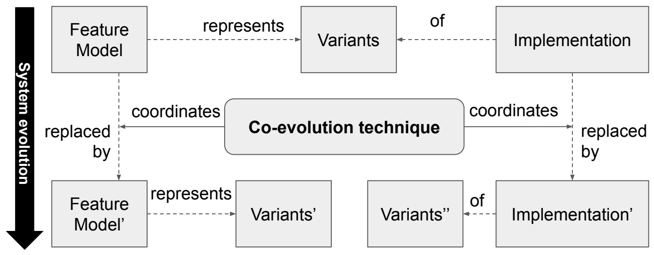

Figure 11: Co-evolution of problem and solution space (CPS)

- CPS1/2: Product-line platform with feature model and implementation code
- CPS1/2: Sequence of revisions for feature model and implementation code *(ground truths:* implementation revisions for *CPS1*, feature model revisions for *CPS2)*

Evaluation metrics

- Accuracy: Similarity of computed and ground truth asset at a certain revision
- Correctness: Consistency between feature model and code

3 COMMUNITY SURVEY

To develop benchmarks, Sim et al. [91] suggest that incorporating community feedback is essential to establish consensus. We followed this recommendation by performing a questionnaire survey with members from the community on software variability and evolution. To gather feedback on the clarity and relevance of our scenario descriptions, two crucial quality criteria for a successful benchmark [91], our survey focused on two research questions:

RQ$_1$ How clear are our scenario descriptions?
RQ$_2$ How relevant are the described scenarios?

In the following, we report the details on our methodology, the results, and threats to validity.

3.1 Methodology

We performed our questionnaire survey in March 2019. The participants for our survey were recruited from two sources: First, we contacted all participants (excluding ourselves) of a Dagstuhl seminar on variability and evolution, the two most relevant research areas (https://dagstuhl.de/en/program/calendar/semhp/?semnr=19191). Second, we contacted authors of recent papers on the same topic. We invited 71 individuals, 41 of them Dagstuhl participants. A total of 20 individuals completed our survey in the given timeframe.

Our questionnaire comprised three parts. First, we presented the general context of our benchmark, including the running example description we introduced in Sec. 2.2. Second, we described the eleven scenarios that we presented in Sec. 2. For each, we included the textual description as well as the list of sub-scenarios. We asked the participants to rate the clarity (using a 5-point Likert scale) of each scenario description (**RQ$_1$**) with the question: *To which extent do you agree that the scenario is clearly described with respect to its usage context and purpose for benchmarking?* Then, we asked the participants to assess the relevance of each overall scenario and its sub-scenarios (**RQ$_2$**) with the question: *To which extent do you agree that supporting the following sub-scenarios is important?* To assess the completeness of our descriptions, we asked the participants to name relevant sub-scenarios not yet considered. Finally, as a prerequisite for our survey of benchmarks (cf. Sec. 4), we asked the participants to name relevant benchmarks they were aware of. A replication package with the questionnaire and all data is found at: https://bitbucket.org/easelab/evobench/.

The initial responses to our survey pointed out a number of shortcomings in the scenario descriptions with respect to clarity. We used these responses to revise the questionnaire after the first 12 responses, presenting an improved version of the scenario descriptions to the remaining eight participants. This intervention is justified by the methodological framework of design science [75], which emphasizes the continuous improvement of research artifacts based on actionable feedback, thus presenting a *best-effort approach*. The most significant change was to remove two sub-scenarios (one from the *variant synchronization* and one from the *transformation* scenario). In other cases, we reworded the scenario descriptions to add more explanations, examples, and avoid potentially confusing wording. To make sure that our revision indeed led to an improvement, we checked the clarity scores after the revision. We found that the clarity scores improved in all cases.

3.2 Results

Figure 12 provides an overview of the results. For each scenario, we show the distribution of answers to our questions about clarity (**RQ$_1$**) and relevance (**RQ$_2$**). We further explain the results based on the textual feedback provided along with the answers.

RQ$_1$: Clarity. For all scenarios, a majority of the participants gave a positive score for clarity. A ratio between 55 % and 90 % gave an *agree* or *strongly agree*. The scenario receiving the most negative scores (21 %) was *variant synchronization*. From the textual feedback provided for this scenario, we observed that several participants struggled to understand a sub-scenario related to the classification of changes into either evolutionary or functional. For example, one participant stated that "*it is not entirely clear how an evolutionary change differs from a functional one.*" After we removed this sub-scenario and its description in the revision, we found that 86 % of the remaining participants gave a positive score. For the *transformation* scenario, we observed the same increase of positive scores (to 86 %) after we removed a sub-scenario related to the replacement of the used variability mechanism. For the other scenarios with comparatively many neutral or negative answers, we did not find any repeated issues occurring in the textual explanations.

RQ$_2$: Relevance. A majority of participants (between 55 % and 95 %) assessed the relevance of each scenario positively. Interestingly, despite the lower scores for clarity, *variant synchronization* is among the two scenarios deemed relevant by 95 % of all participants. To study this discrepancy further, we analyzed the scores per sub-scenario. We found that most participants considered the sub-scenario that we removed in the revision (*classify changes*, 33 % positive responses) less relevant than the remaining *variant synchronization* sub-scenarios. Likewise, *transformations* attracted 100 % positive scores for overall relevance after we removed the least relevant sub-scenario (*exchange variability mechanism*, 33 % positive responses). In other cases with comparatively fewer positive scores (*architecture recovery* and *problem-solution space co-evolution*; 60 % and 63 % positive scores, respectively), it was not obvious from the textual comments how these scores can be explained. An interesting case is visualization. Despite the overall mid-range responses, two participants deemed it particularly relevant, but hard to benchmark: *"I believe visualization has much potential to improve many tasks in evolution of variant-rich systems. [...] Evaluation itself, in terms of measuring the impact, is harder."'*

The participants' feedback confirms the clarity and relevance of our benchmark descriptions. The scenarios *variant synchronization*, *feature identification & location*, and *constraints extraction* were considered most relevant.

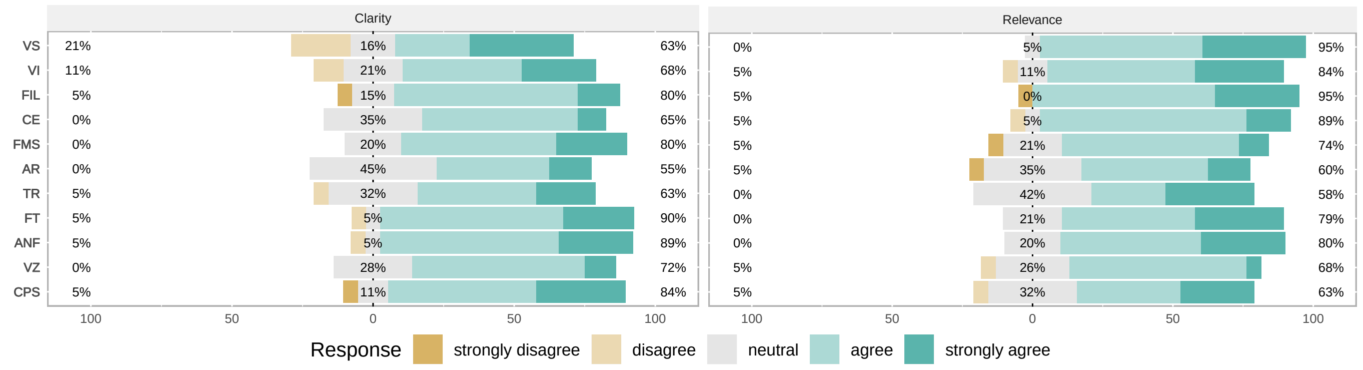

Figure 12: Results of the survey concerning clarity and relevance for scenarios: Variant Synchronization, Feature Identification and Location, Constraints Extraction, Feature Model Synthesis, Variant Integration, Architecture Recovery, Functional Testing, Analysis of Non-Functional Properties, Visualization, and Co-Evolution of Problem & Solution Space.

3.3 Threats to Validity

The external validity of our survey is threatened by the number of participants. However, since we focus on a highly specialized population—the community of variability and evolution experts—valid conclusions about that population can be supported by a smaller sample than a large population would require. By inviting the attendees of a relevant Dagstuhl seminar, we benefit from a pre-selection of experts in this area. Regarding conclusion validity, the confidence in our clarity scores could be improved by asking the participants to solve comprehension tasks, rather than having them rate the description clarity. However, such an experiment would have taken much more time and, therefore, would have risked to affect the completion rate.

4 SURVEYING EXISTING BENCHMARKS

In this section, we survey a selection of benchmarks with regard to their applicability to the scenarios we introduced in Sec. 2.

4.1 Methodology

Selection. As starting point for our selection of benchmarks, we collected a list of benchmarks that we were aware of, due to our familiarity with the field (convenience sampling). To get a more complete overview in a systematic way, we gathered additional benchmarks using a dedicated question in our community survey, in which we asked the participants to name benchmarks that they are aware of. Finally, since we found that a dedicated benchmark was not available for each scenario, we also considered benchmarks from related areas, such as *traceability* research, and identified whether they match our scenarios. From these steps, we derived an initial list of 17 benchmark candidates.

Based on our definition of benchmark, as given in Sec. 1, we defined the following inclusion criteria:

- I1 The availability of a dataset based on one or more systems created by industrial practitioners, and
- I2a The availability of a ground truth for assessing the correctness of a given technique, or
- I2b The availability of a framework for assessing other properties of interest.

From the initial 17 benchmark candidates, nine satisfied the inclusion criteria, meaning that they provided a suitable dataset, and either a ground truth or a framework for assessing a relevant technique. We focused on these nine benchmarks in our survey and excluded eight additional ones that did not satisfy all criteria. The excluded candidates can be considered as notable datasets, as they may still offer some value for benchmarking. We discuss the selected benchmarks in Sec. 4.2, and the notable datasets in Sec. 5.

Assessment. To determine how well our eleven scenarios are supported by the identified benchmarks and to identify synergies between benchmarks and scenarios, we assessed the suitability of each benchmark for each scenario. To this end, for a given benchmark candidate, we considered the requirements given in the benchmark descriptions (Sec. 2) and checked whether it fulfills the requirements and provides the artifacts that we defined.

4.2 Results

In Table 1, we provide an overview of the considered benchmarks and scenarios. The area from which the benchmark originally stems is given as *original context* in the table. A full circle indicates full support for at least one sub-scenario, a half-filled circle indicates partial support (i.e., a subset of the required artifacts is available) for at least one sub-scenario, and an empty circle indicates no support of the given scenario by means of the given benchmark. In the following, we briefly introduce the benchmarks and explain the cases in which a scenario is fully or partially supported.

ArgoUML-SPL FLBench [57] has a long tradition as benchmark for feature location in single systems and in families of systems [56]. The ground truth consists of feature locations for eight optional features of ArgoUML at the granularity of Java classes and methods. A feature model is available. The framework allows to generate predefined scenarios (a set of configurations representing a family) and to calculate metrics reports for a given feature location result. Given that this benchmark only contains eight optional features, we argue that it only partially satisfies the needs for feature location.

Drupal [81] is a dataset of functional faults in the variable content management system Drupal. For each of the faults, it contains the feature or feature interaction responsible for triggering the fault. Moreover, the faults are reported over two different versions of

Table 1: Mapping of existing benchmarks to scenarios, with circles indicating fulfillment of scenario requirements

Benchmark	Original Context	VS	VI	FIL	CE	FMS	AR	TR	FT	ANF	VZ	CPS
ArgoUML-SPL FLBench. [57]	Feature location	○	○	◓	○	○	○	○	○	○	○	○
Drupal [81]	Bug detection	○	○	○	○	○	○	○	●	○	◓	○
Eclipse FLBench [63]	Feature location	○	○	◓	◓	◓	○	○	○	○	○	○
LinuxKernel FLBench. [98]	Feature location	○	○	◓	◓	◓	○	○	○	○	○	○
Marlin & BCWallet [42]	Feature location	○	○	◓	○	○	○	○	○	○	◓	○
ClaferWebTools [37]	Traceability	○	○	◓	○	◓	○	○	○	○	◓	○
DoSC [101]	Change discovery	○	◓	◓	○	◓	○	◓	○	○	◓	○
SystemsSwVarModels [15]	FM synthesis	○	◓	○	●	●	○	○	○	○	◓	○
TraceLab CoEST [40]	Traceability	○	○	◓	○	○	○	○	○	○	○	○
Variability bug database [1]	Bug detection	○	○	◓	○	○	○	○	●	○	◓	○

Drupal, which may indicate faults that were introduced by the evolution of the system. This dataset is useful for the scenario of functional testing, to evaluate whether the selected variants for application testing cover relevant feature interactions that are known to contain faults. Moreover, the information of feature interactions could be used to partially benchmark visualization.

Eclipse FLBench [63] is a benchmarking framework for feature location techniques in single systems and in families of systems. Since the ground-truth traces map features to components (i.e., Eclipse plugins), the granularity is coarse and there are no cross-cutting features, thus justifying only partial support of feature location. This benchmark supports different Eclipse releases, each containing around 12 variants, 500 features, and 2,500 components. The Eclipse FLBench also contains information about feature dependencies and hierarchy, but only "requires" constraints, thus justifying partial support of constraints extraction and FM synthesis.

Linux Kernel FL Bench [98] is a database containing the ground-truth traces from a selection of features of the Linux Kernel to corresponding C code. It contains the locations of optional features within 12 product variants derived from three Linux kernel releases. For each variant, we have around 2,400 features and 160,000 ground-truth links between features and code units. The database contains information about "requires" and "excludes" feature constraints, as well as the feature model hierarchy, making it a suitable ground truth for constraints extraction and feature-model synthesis. However, as it was not its intended usage, more complex feature constraints are not captured.

Marlin & BCWallet [42] is a dataset of feature locations (represented as embedded code annotations), feature models, and feature fact sheets of two open-source systems, all of which can serve as ground truth for feature identification and location techniques. It comprises both mandatory and optional features. The annotations can also serve as input for feature dashboards that provide visualizations with several metrics [5], for instance, assets related to a feature, scattering degrees, and developers associated with each feature.

ClaferWebTools [37] is a dataset with feature locations of both mandatory and optional features, as well as feature models, together with an evolution history. ClaferWebTools is a clone & own-based system that evolved in four variants. Like Marlin & BCWallet, the locations are embedded into the source code. It can be used to evaluate feature-location techniques exploiting historical information, or visualization techniques showing the evolution of features.

DoSC (*Detection of Semantic Changes* [101]) is a dataset with revision histories of eight Java projects for benchmarking semantic change detection tools. Semantic changes are commits that correspond to entries from an issue tracking system; they can be considered as features in a broader sense. Traces from semantic changes to implementation elements are included, thus providing a ground truth for feature location (partially supported, since only optional features are considered) and a basis for visualization. The revision histories also provide a rich data source for benchmarking transformation and variant integration. However, full support is prohibited by the lack of a feature model and available ground truths.

SystemsSwVarModels [15] comprises a corpus of 128 extracted real-world variability models from open-source systems-software, such as the Linux kernel, the eCos operating system, BusyBox, and 12 others. The models are represented in the variability modeling languages Kconfig [85] and CDL [14], with the benchmark providing tools to analyze and transform these models into their configuration space semantics (expressed as Boolean, arithmetic, and string constraints), abstracted as propositional logics formulas. As such, these formulas can be used to benchmark constraints extraction from codebases and feature model synthesizes. To some extent, the corpus can be used to benchmark feature-oriented visualizations (e.g., slicing feature models) and problem & solution space co-evolution.

TraceLab CoEST [40] is an initiative of the Center of Excellence for Software and Systems Traceability gathering a set of case studies on traceability recovery with their corresponding ground-truth traces. We can find benchmarks with traces from requirements to source code, from requirements to components, from high- to low-level requirements, from use cases to source code, and other types of traces that partially satisfy the needs of evaluating feature location techniques in single systems.

Variability Bug Database [1] is an online database of 98 variability-related bugs in four open-source repositories: The Linux kernel, BusyBox, Marlin, and Apache. The meta-data provided for bug entries include a description, a type (e.g., "expected behavior violation"), a configuration, and pointers to a revision where the bug appears and where it is fixed. This database is especially useful for functional testing, as it provides a ground truth in the form of faults together with the configurations in which they appear. The projects contain `#ifdef` directives that can be considered as variability annotations, rendering the database partially suitable for benchmarking feature location and visualization.

While we identified synergies between the scenarios and existing benchmarks, the overall coverage is still low: A complete benchmark is only available for three of the eleven considered techniques. Four scenarios lack any benchmark: variant synchronization, analysis of non-functional properties, architecture recovery, and co-evolution of problem & solution space. The former two were deemed as particularly relevant in our community survey.

5 RELATED WORK

Besides the benchmarks we analyzed in the previous section, we are aware of several datasets and proposals that aim to achieve similar ideas, and benchmarks from related areas.

Repositories. Some repositories collect artifacts or full projects in the domain of software-product-line engineering. For example, spl2go (http://spl2go.cs.ovgu.de/) provides a set of various software product lines. However, most of these systems are based on student projects and they provide solely downloadable code. A more extensive overview especially on extractive software-product-line adoption is offered by the ESPLA catalog [56]. ESPLA collects information from existing papers, rather than providing data or an infrastructure by itself. Similarly, tools like FeatureIDE [65] and PEoPL [9] provide some complete example product lines, but are neither industrial nor do they have ground truths.

Case Studies and Datasets. Some case studies have been introduced that partly aimed to provide the basis for establishing benchmarks. The potentially best-known and first of such studies is the graph product line introduced by Lopez-Herrejon and Batory [52]. McGregor [64] reports experiences of using the fictional arcade product line in teaching, but focuses solely on reporting established practices. Recently, several case studies have been reported that particularly aim to provide suitable data sets for evaluating techniques for the evolution and extraction for software product lines. For example, Martinez et al. [59] extracted a software product line from educational robotic variants. The Apo-Games [46] are a set of real-world games, realized using clone & own, with which the authors aim to provide a benchmark for the extraction of software product lines based on community contributions. Two recent works in fact provide datasets detailing the migration of dedicated subsets of the cloned games into product line platforms [4, 21]. BeTTy [83] is a feature model generator, focused on benchmarking and testing automated analysis techniques for feature models. Tzoref-Brill and Maoz [96] provide a dataset for assessing co-evolution techniques for combinatorial models and tests. A combinatorial model, similar to a configuration, is a set of bindings of parameters to concrete values. Finally, SPLOT [66] provides a set of 1,073 feature models and constraints, also including an editor and analysis framework. It mostly includes academic feature models and toy examples. None of these works represent a benchmark according to our criteria, namely that they are based on assets created by practitioners *and* provided together with a ground truth or assessment framework.

Benchmarks in Related Areas. Various benchmarks have been proposed in areas that are closely related to variability engineering. SAT solvers are often applied in the context of software variability, specially for family-based analyses. The annual SAT competitions [36] provide various benchmarks and are important enablers for the SAT community. In the area of software-language engineering, the language workbench challenge [28] is an annual contest with the goal of promoting knowledge exchange on language workbenches. Model transformations provide the capability to represent product composition as transformation problem and have several established benchmarks, for instance, on graph transformation [97] and scalable model transformations [93]. While these benchmarks are complementary to the ones we consider in this paper, they report best practices that should be applied when implementing our envisioned benchmark set.

6 CONCLUSION AND ROADMAP

In this paper, we aimed to pave the way for a consolidated set of benchmark for techniques that support developers during the evolution of variant-rich systems. We studied relevant scenarios, investigated the clarity and relevance of the scenarios in a survey with variability and evolution experts, and surveyed the state of the art in benchmarking of these techniques.

Results. In summary, our main results are:

- We identified 11 scenarios covering the evolution of variant-rich systems, together with requirements for benchmarking the relevant techniques.
- Community feedback shows that our scenarios are clearly defined and important to advance benchmarking in the area.
- Only three out of the 11 scenarios are completely supported by existing benchmarks, highlighting the need for a consolidated benchmark set with full support for all scenarios.

Roadmap. Our results suggest the following research roadmap to eventually achieve such an envisioned benchmark set.

As a key goal, we aim to set up a common infrastructure for all scenarios presented in this paper. This way, we can utilize synergies between benchmarks, specifically by means of shared datasets and assets. Where available, we may reuse publicly available implementations of benchmark frameworks and integrate them.

Most scenarios require a manually curated ground truth. Therefore, creating ground truths for the available datasets is a substantial effort, a call for investing more resources into benchmarking.

A further important goal is to broaden the scope of datasets. Most available datasets are based on open-source projects from traditional embedded systems. It is worthwhile to include datasets from upcoming domains that need variability handling, including data-analytics software [30], service robotics [31], and cyber-physical systems [16].

Raising awareness for the challenges and opportunities for benchmarking takes a concerted effort. Developers of new techniques shall be encouraged to use our benchmark infrastructure, articulate gaps in the benchmark literature, and fill them by contributing their own benchmarks. We plan to advertise our initiative in the appropriate mailing lists and social media.

Acknowledgments. Supported by ITEA project REVaMP^2 funded by Vinnova Sweden (2016-02804). We thank the participants of Dagstuhl seminar 19191, all survey participants, and Tewfik Ziadi for input and comments on earlier versions of this paper.

REFERENCES

[1] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology* 26, 3 (2018), 10:1–10:34.

[2] Hadil Abukwaik, Andreas Burger, Berima Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*. IEEE.

[3] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On Extracting Feature Models from Product Descriptions. In *VaMoS*. ACM.

[4] Jonas Akesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *23rd International Systems and Software Product Line Conference (SPLC), Challenge Track*.

[5] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. 2017. Florida: Feature Location Dashboard for Extracting and Visualizing Feature Traces. In *VaMoS*. ACM.

[6] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ştefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *ICSE*. ACM.

[7] Patrizia Asirelli, Maurice H. Ter Beek, Alessandro Fantechi, and Stefania Gnesi. 2010. A Logical Framework to Deal with Variability. In *IFM*. Springer.

[8] Wesley K. G. Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.

[9] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE*. IEEE.

[10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[11] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *CAISE*. Springer.

[12] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *SPLC*. ACM.

[13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*. ACM.

[14] Thorsten Berger and Steven She. 2010. *Formal Semantics of the CDL Language.* Technical Report. Department of Computer Science, University of Leipzig, Germany.

[15] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[16] Hamid Mirzaei Buini, Steffen Peter, and Tony Givargis. 2015. Including variability of physical models into the design automation of cyber-physical systems. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*. ACM, 1–6.

[17] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *ICSME*. IEEE.

[18] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques using ArgoUML-SPL. In *VaMoS*. ACM.

[19] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*. ACM.

[20] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2011. A Systematic Mapping Study of Software Product Lines Testing. *Information and Software Technology* 53, 5 (2011), 407–423.

[21] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Java-Based Apo-Games into a Composition-Based Software Product Line. In *23rd International Systems and Software Product Line Conference (SPLC), Challenge Track*.

[22] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *SPLC*. ACM.

[23] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Information and Software Technology* 56, 10 (2014), 1183–1199.

[24] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. 2012. Strategies for Testing Products in Software Product Lines. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–8.

[25] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *CSMR*. IEEE.

[26] Wolfgang Eixelsberger, Michaela Ogris, Harald Gall, and Berndt Bellay. 1998. Software Architecture Recovery of a Program Family. In *ICSE*. IEEE.

[27] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of Feature Locations with the Tool FeatureDashboard. In *23rd International Systems and Software Product Line Conference (SPLC), Tools Track*.

[28] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *SLE*. Springer.

[29] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *ICSME*. IEEE.

[30] Amir Gandomi and Murtaza Haider. 2015. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management* 35, 2 (2015), 137–144.

[31] Sergio García, Daniel Strüber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. 2019. Variability Modeling of Service Robots: Experiences and Challenges. In *VaMoS*. ACM, 8:1–6.

[32] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *Journal of Systems and Software* 84, 12 (2011), 2208–2221.

[33] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wąsowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering* 23, 3 (2018), 1826–1867.

[34] Haitham S. Hamza, Jabier Martinez, and Carmen Alonso. 2010. Introducing Product Line Architectures in the ERP Industry: Challenges and Lessons Learned. In *SPLC*.

[35] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, Paul Grünbacher, Florian Reisinger, and Alexander Egyed. 2018. Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems. In *ETFA*. IEEE.

[36] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012), 89–92.

[37] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*. ACM.

[38] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. Carnegie-Mellon University, Pittsburgh, PA, USA.

[39] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering* 40, 1 (2014), 67–82.

[40] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn. 2012. TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions. In *ICSE*. IEEE.

[41] Rainer Koschke, Pierre Frenzel, Andreas P. J. Breu, and Karsten Angstmann. 2009. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Journal* 17, 4 (2009), 331–366.

[42] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.

[43] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *SPFE*. Springer.

[44] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards Automated Test Refactoring for Software Product Lines. In *SPLC*. ACM.

[45] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. *Software Engineering for Variability Intensive Systems.* CRC Press, Chapter Features and How to Find Them: A Survey of Manual Feature Location, 153–172.

[46] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *SPLC*. ACM.

[47] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *SPLC*. ACM.

[48] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming* 78, 8 (2013), 1010–1034.

[49] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *SPLC*. ACM.

[50] Max Lillack, Ştefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. 2019. Intention-Based Integration of Software Variants. In *ICSE*. ACM.

[51] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *GPCE*. ACM.

[52] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *GPCE*. Springer.

[53] Roberto E. Lopez-Herrejon and Alexander Egyed. 2013. Towards Interactive Visualization Support for Pairwise Testing Software Product Lines. In *VISSOFT*. IEEE.

[54] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *SPLC*. Springer.

[55] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. 2016. Model Transformation Intents and Their Properties. *Software and Systems Modeling* 15, 3 (2016), 647–684.

[56] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC*. ACM.

[57] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *SPLC*. ACM.

[58] Jabier Martinez, Jean-Sébastien Sottet, Alfonso García Frey, Tewfik Ziadi, Tegawendé F. Bissyandé, Jean Vanderdonckt, Jacques Klein, and Yves Le Traon. 2017. Variability Management and Assessment for User Interface Design. In *Human Centered Software Product Lines*. Springer.

[59] Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. 2018. Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study. In *SPLC*. ACM.

[60] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In *SPLC*. ACM.

[61] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Name Suggestions During Feature Identification: The Variclouds Approach. In *SPLC*. ACM.

[62] Jabier Martinez, Tewfik Ziadi, Raul Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2014. Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines. In *VISSOFT*. IEEE.

[63] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2018. Feature Location Benchmark for Extractive Software Product Line Adoption Research Using Realistic and Synthetic Eclipse Variants. *Information and Software Technology* 104 (2018), 46–59.

[64] John D. McGregor. 2014. Ten Years of the Arcade Game Maker Pedagogical Product Line. In *SPLC*. ACM.

[65] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[66] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. SPLOT: Software Product Lines Online Tools. In *OOPSLA*. ACM.

[67] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *ASE*. ACM.

[68] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *ICSE*. ACM.

[69] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[70] Sana Ben Nasr, Guillaume Bécan, Mathieu Acher, João Bosco Ferreira Filho, Nicolas Sannier, Benoit Baudry, and Jean-Marc Davril. 2017. Automated Extraction of Product Comparison Matrices from Informal Product Descriptions. *Journal of Systems and Software* 124 (2017), 82–103.

[71] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. 2008. Applying Visualisation Techniques in Software Product Lines. In *SoftVis*. ACM.

[72] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. 2011. Investigating the Safe Evolution of Software Product Lines. *ACM SIGPLAN Notices* 47, 3 (2011), 33–42.

[73] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *MODULARITY*. ACM.

[74] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2018. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* (2018).

[75] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems* 24, 3 (2007), 45–77.

[76] Juliana Alves Pereira, Jabier Martinez, Hari Kumar Gurudu, Sebastian Krieter, and Gunter Saake. 2018. Visual Guidance for Product Line Configuration using Recommendations and Non-Functional Properties. In *SAC*. ACM.

[77] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *SPLC*. ACM.

[78] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *SPLC*. ACM.

[79] Andrey Sadovykh, Tewfik Ziadi, Jacques Robin, Elena Gallego, Jan-Philipp Steghoefer, Thorsten Berger, Alessandra Bagnato, and Raul Mazo. 2019. REVAMP2 Project: Towards Round-Trip Engineering of Software Product Lines – Approach, Intermediate Results and Challenges. In *TOOLS 50 + 1*.

[80] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting Model Transformations to Product Lines. In *ICSE*. ACM.

[81] Ana B. Sánchez, Sergio Segura, José Antonio Parejo, and Antonio Ruiz Cortés. 2017. Variability Testing in the Wild: The Drupal Case Study. *Software and System Modeling* 16, 1 (2017), 173–194.

[82] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. 2012. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *VaMoS*. ACM.

[83] Sergio Segura, José A. Galindo, David Benavides, José Antonio Parejo, and Antonio Ruiz Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *VaMoS*. ACM.

[84] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari Sahraoui. 2017. Recovering Software Product Line Architecture of a Family of Object-Oriented Product Variants. *Journal of Systems and Software* 131 (2017), 325–346.

[85] Steven She and Thorsten Berger. 2010. *Formal Semantics of the Kconfig Language*. Technical Report. Electrical and Computer Engineering, University of Waterloo, Canada.

[86] Steven She, Krzysztof Czarnecki, and Andrzej Wąsowski. 2012. Usage Scenarios for Feature Model Synthesis. In *VARY*. ACM.

[87] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *ICSE*. ACM.

[88] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2014. Efficient Synthesis of Feature Models. *Information and Software Technology* 56, 9 (2014), 1122–1143.

[89] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *ESEC/FSE*. ACM.

[90] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal* 20, 3-4 (2012), 487–517.

[91] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *ICSE*. IEEE.

[92] Zipani Tom Sinkala, Martin Blom, and Sebastian Herold. 2018. A Mapping Study of Software Architecture Recovery for Software Product Lines. In *ECSA*. ACM.

[93] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. 2016. Scalability of Model Transformations: Position Paper and Benchmark Set. In *BigMDE*.

[94] Ştefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *ICSME*. IEEE.

[95] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. 2008. FAMA Framework. In *SPLC*. IEEE.

[96] Rachel Tzoref-Brill and Shahar Maoz. 2018. Modify, Enhance, Select: Co-Evolution of Combinatorial Models and Test Plans. In *ESEC/FSE*. ACM.

[97] Gergely Varró, Andy Schürr, and Dániel Varró. 2005. Benchmarking for Graph Transformation. In *VL/HCC*. IEEE.

[98] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2013. A Large Scale Linux-Kernel based Benchmark for Feature Location Research. In *ICSE*. ACM.

[99] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *WCRE*. IEEE.

[100] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *ICSE*. IEEE.

[101] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2017. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *MSR*. IEEE.