Feature agnostic feature isolation

Yannis Hübenthal

November 16, 2025

Abstract

Reengineering software product lines (SPLs) from clone-and-own portfolios often relies on language-specific tooling and substantial manual effort to isolate, relate, and reconstruct features across variants. This paper presents a *programming-language independent* and *feature-agnostic* method that supports full-circle reengineering by recovering a unified representation from multiple variants and regenerating the originals from it. The approach treats all artefacts uniformly as text, covering polyglot source code, configuration files, build scripts, domain-specific languages, and other textual resources. Artefacts are tokenized along character-class boundaries and processed by a stratified similarity pipeline that collapses identical files, identifies matching and non-matching lines, and applies token-level alignment via the Longest Common Subsequence to obtain fine-grained blocks characteristic of commonalities and differences. Similarity-level pruning reduces redundant comparisons and keeps the analysis tractable for small to medium-sized variant sets. A prototype implementation demonstrates feasibility in a non-trivial synthetic SPL and the ArgoUML benchmark, showing that the recovered representation is precise enough to regenerate variants while avoiding dependence on parsers, abstract syntax trees, or pretty printers. The work contributes a parser-free extraction pipeline, a proof-of-concept tool, and initial empirical evidence that programming-language independent text processing can support SPL reengineering across heterogeneous artefacts. We spell out current assumptions (aligned paths, text-level equivalence) and outline future steps, including validation on SPLs from industrial practice and extensions towards additional resources such as string tables, help files, and other non-code artefacts.

1 Introduction

Organizations increasingly deliver families of related products rather than single monolithic systems. A common pragmatic strategy is clone-and-own: copy an existing product and adapt it to new requirements. In practice, such variant portfolios are often polyglot: they span multiple programming languages and a variety of non-code textual artefacts, including configuration files, build scripts, domain-specific languages (DSLs), and resource texts. Over time, the accumulation of customer-specific adaptations causes variants to diverge, while the relationships among variants remain implicit.

The resulting maintenance burden is considerable. First, propagating bug fixes and feature changes across multiple forks is labour-intensive and error-prone. Second, traceability degrades: it becomes difficult to determine which variant contains which feature or fix. Third, divergence increases cost and risk when onboarding new customers or composing new configurations from existing assets. These issues are a common motivation for moving from ad hoc clone-and-own development to software product line (SPL) engineering, where reusable assets and variability are managed explicitly (Clements and Northrop, 2001; Pohl and Metzger, 2018).

SPL reengineering and variability mining aim to recover reusable assets and variability models from existing variants. Many established techniques rely on language-specific parsers, abstract syntax trees, or feature structure trees for each programming language and artefact type they analyse, often complemented by type-system reasoning or model transformations. These approaches have shown strong results in settings where suitable front-ends are available, but they are costly to deploy in polyglot, DSL-rich environments: front-ends may not exist for in-house DSLs or bespoke configuration formats, and maintaining multiple parsers and pretty printers requires substantial effort. This creates a gap between the heterogeneous artefacts found in industrial portfolios and the assumptions of parser-dependent reengineering pipelines.

We address the following question: how can one extract a reusable, generative representation from sets of related variants in a way that is programming-language independent and applicable to heterogeneous textual artefacts, while still supporting regeneration of the original variants? Our answer is a parser-free, token-based approach that treats all artefacts uniformly as text and does not assume predefined feature names or models, making the analysis *feature-agnostic* in scope.

The method performs a stratified similarity analysis. Identical files are collapsed at the file level, matching and non-matching material is identified at the line level, and residual differences are aligned at the token level using a Longest Common Subsequence procedure. The resulting common and variant-specific blocks together constitute a recovered SPL representation from which the original variants can be regenerated. The reasoning is programming-language independent because the pipeline only requires a lightweight tokenizer for each artefact family, rather than full parsers or grammars.

We ground the work in an industrially motivated scenario: a vendor of control systems maintains between five and twenty customer-specific variants as full project directories created via clone-and-own. Our prototype focuses on such small-to-medium portfolios and assumes aligned relative paths across variants; regenerated products aim at textual equivalence to the originals. Within this scope, we report a feasibility study and a first partial empirical evaluation on a deliberately constructed SPL and on the ArgoUML benchmark, assessing regeneration fidelity and practical performance.

Contributions. This paper makes the following contributions:

- A lightweight, programming-language independent pipeline for extracting common and variant-specific blocks from polyglot variant sets using parser-free, token-based similarity analysis.
- A feature-agnostic prototype implementation that recovers a generative representation and regenerates the original variants from it.
- A feasibility study and first partial empirical evaluation on a synthetic SPL and the ArgoUML benchmark, demonstrating end-to-end regeneration and discussing assumptions and limitations of the approach.

Paper structure. Section 3 introduces background and formalizes the problem setting. Section **??** presents the extraction approach. Section **??** describes the prototype. Section **??** reports the evaluation. Section **??** discusses implications and limitations. Section **??** reviews related work, and Section **??** concludes with directions for future research.

2 Related Work

Research on extractive software product lines (SPLs) and variant management spans parser- and model-based reengineering, variability mining, clone-based analyses, and software transplantation. We group the most relevant strands and position our contribution accordingly.

Parser- and FST-based SPL reengineering ExtractorPL abstracts variants into feature structure trees (FSTs) and synthesizes an SPL implementation capable of regenerating the originals (Ziadi et al., 2014). Although the core representation is language-independent, the approach depends on programming-language specific parsers and pretty printers to translate between code and FSTs. FeatureHouse generalizes FST-based, superimposition-style composition across multiple implementation languages (Apel et al., 2013); each supported language again requires a dedicated frontend. Our work shares the goal of programming-language independent reasoning but deliberately avoids parsers, abstract syntax trees (ASTs), and pretty printers by operating on tokenized text only.

Preprocessor- and annotation-based extraction Annotation-centric approaches and variability-aware frontends assume explicit feature annotations or disciplined conditional compilation. Variability-aware parsing (TypeChef) builds a unified variational AST for all `#ifdef` configurations (Kästner, Giarrusso, Rendel, et al., 2011), while partial preprocessing produces conditional token streams suitable for subsequent analyses (Kästner, Giarrusso, and Ostermann, 2011). These techniques provide strong guarantees for annotated code bases but tie the pipeline to specific languages and preprocessing ecosystems. In contrast, we neither rely on annotations nor conditional compilation and remain parser-free.

Variability mining and feature location Variability mining combines structural and textual signals to detect features and their implementations. Kästner et al. present a semiautomatic process that integrates multiple sources of evidence to consistently detect product-line features (Kästner, Dreiling, et al., 2014). Outlier- and cluster-based analysis has been shown to improve mining quality in industrial settings (Wille, Babur, et al., 2018). Linsbauer et al. address variability extraction and modeling for product variants,

with a focus on building explicit variability models and reusable artefacts from existing portfolios (Linsbauer et al., 2017). Bottom-up frameworks such as BUT4Reuse provide an extensible process and intermediate representations into which concrete mining and model-synthesis techniques can be plugged (Martinez et al., 2015). Model-based delta module generation leverages mined relations to construct deltas and regenerate variants (Wille, Runge, et al., 2017). Our method operates at a lower abstraction level: we compute programming-language independent, token-based block correspondences across artefacts that can serve as a similarity backbone within such frameworks.

Clone-based and text-based analyses Token- and text-based clone detectors have been studied in SPL settings to understand and refactor feature implementations. Schulze et al. analyse code clones in feature-oriented SPLs to investigate their impact on maintenance and evolution (Schulze et al., 2010). We adopt a related low-level perspective (tokenization and alignment) but target an explicit, generative SPL representation that supports regeneration of variants, rather than clone reporting alone.

Formal concept analysis and data mining Formal Concept Analysis (FCA) has been proposed as a structural framework for representing commonality and variability by building lattices over artefact presence and configuration relations, which can inform feature grouping and model synthesis (Galasso, 2025). Our current pipeline stops at block presence across variants; FCA-style structuring is a complementary step on top of our extracted relations and is left as future work.

Software transplantation and slice-based reengineering Software transplantation transfers feature "organs" between systems via slicing and search (Barr et al., 2015). Building on this idea, recent work explores SPL engineering via transplantation, extracting feature slices and integrating them into a base product line (Souza et al., 2025). These methods typically rely on language-specific analyses and focus on moving features across systems. We instead mine a coherent representation from a set of closely related clone-and-own variants and regenerate those variants from the mined representation.

Positioning Taken together, prior work demonstrates effective SPL extraction via parser-dependent analyses (ExtractorPL, FeatureHouse, TypeChef), framework-based variability mining (BUT4Reuse, FCA, variability extraction), clone analysis, and transplantation. Our contribution is a deliberately lightweight alternative: a parser-free, token-based pipeline that treats heterogeneous textual artefacts uniformly and yields a generative representation capable of regenerating the original variants, which makes it suited to polyglot, DSL-rich portfolios where programming-language specific frontends are impractical or unavailable.

3 Background and Problem Setting

3.1 Basic Concepts

We adopt the standard understanding of a software product line (SPL) as a *set of software-intensive systems that share a common, managed set of features for a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*, as formulated by Clements and Northrop in their seminal work on SPL engineering (Clements and Northrop, 2001). This view has been reiterated and refined in subsequent overviews of SPL practice and research (Pohl and Metzger, 2018). The emphasis on a managed portfolio of systems and on systematic reuse of core assets distinguishes SPL engineering from ad hoc reuse strategies.

Within this context, a *feature* denotes a system property that is relevant to some stakeholder and is used to capture commonalities or to discriminate among systems in a family. This notion of features as stakeholder-relevant units of functionality or quality attributes is well established in the generative programming and SPL literature (Czarnecki and Eisenecker, 2000; Czarnecki, Helsen, et al., 2005). It provides a unifying abstraction across heterogeneous artifacts and implementation techniques: a single feature may have manifestations in source code, configuration files, build scripts, or domain-specific languages.

A *variant* (or product) is one concrete member of the SPL, i.e., one program that realizes a particular combination of features (Thüm et al., 2011; Linsbauer et al., 2017). From this perspective, an SPL can be understood as a set of related program variants, together with a feature space and a (possibly partial) mapping from feature selections to variants. Variability extraction and reengineering techniques aim at recovering such a mapping and the underlying reusable assets from existing variant sets.

3.2 Clone-and-Own in Industrial Practice

A widely observed practice for developing and maintaining product variants in industry is *clone-and-own*. In clone-and-own, new variants are created by copying (cloning) an existing system and then modifying the copy to satisfy the requirements of a particular customer or use case. Linsbauer et al. provide a precise characterization of this practice in the context of variability extraction and modeling for product variants (Linsbauer et al., 2017). While

clone-and-own offers an attractive, low-barrier reuse mechanism, it lacks the disciplined variability management and systematic reuse of SPL engineering.

The consequences of clone-and-own for long-term maintenance and evolution have been documented in several empirical and industrial reports. For example, Kuiter et al. describe the migration of a temperature monitoring system from clone-and-own development to an SPL and report on the maintenance burdens associated with the original clone-and-own portfolio (Kuiter et al., 2018). Fischer et al. propose tool support to enhance clone-and-own with more systematic reuse, again motivated by the challenges of managing diverging clones over time (Fischer et al., 2014). Commonly cited problems include redundant bug fixing across multiple forks, inconsistent propagation of changes, and the absence of an explicit variability model.

In the scenario that motivates this work, the domain is an industrial control system. A vendor delivers customized control systems to multiple customers. New customer solutions are routinely created via clone-and-own from an existing system. Over time, between roughly five and twenty variants evolve for different customers. Each variant is organized as a full project directory, comprising code and related artifacts. As the number of variants and the extent of customer-specific changes grow, maintenance and evolution become increasingly costly: engineers must manually identify and propagate relevant changes across variants, and the relationships between variants are only implicitly encoded in the clone history.

3.3 Polyglot Variant Portfolios and Artifacts

The variants in our setting are *polyglot*. The primary implementation languages of interest are Java and C++, but the systems also involve additional artifacts such as configuration files, build scripts, textual requirements, and potentially dedicated domain-specific languages (DSLs). In principle, each of these artifact types may carry variability information and feature implementations.

In practice, the polyglot nature of such portfolios poses a challenge for SPL reengineering. While structure may exist in many artifacts (e.g., syntactic structure in source code or DSLs, schema-constrained configuration formats), exploiting this structure typically requires language-specific parsers, grammars, or meta-models. For in-house DSLs and ad hoc configuration formats, such tooling may be unavailable, only partially implemented, or costly to maintain. One of the core design decisions of the present work

is therefore to treat all artifacts uniformly as text. Our extraction method deliberately refrains from using language-specific structure and instead operates on a generic, token-based representation that can be applied to arbitrary text-based artifacts.

3.4 Problem Statement

We consider an industrial control system that is maintained via clone-and-own across multiple customers. Starting from a common ancestor system, customer-specific changes and bug fixes are applied separately to each cloned variant. Over time, the variants diverge. This leads to several interrelated problems. First, the cost of maintaining multiple forks grows, as the same defect may need to be fixed independently in many variants. Second, it becomes difficult to track which variants contain which bug fixes or feature implementations. Third, propagating a new bug fix or feature to all relevant variants consistently is hard, especially when changes interact or when customer-specific constraints must be respected.

Existing SPL reengineering approaches aim to address such problems by deriving reusable assets and variability models from existing variants. However, many of these approaches rely on language-specific parsers, abstract syntax trees, or feature structure trees for each programming language and artifact type they analyze. In polyglot and DSL-heavy settings, this dependence on language-specific tooling is problematic: tools must be developed or adapted for each language and DSL, and the reengineering process breaks down when adequate frontends are not available. This motivates the following goal: to automatically extract a structured SPL representation from existing variants without relying on language-specific tooling and to enable the regeneration of all original variants from this representation.

3.5 Input Model

The input to our approach is a finite set of product variants. Each variant is modeled as a project directory tree whose contents comprise source code files, DSL files, textual requirements, build scripts, and other text-based artifacts. At the granularity relevant for our method, each file is represented by its path within the directory tree and its textual content.

The current prototype assumes that file names and directory structure are aligned across variants, i.e., corresponding files share the same relative

path. Under this assumption, there is no built-in support for tracking renamed or moved files between variants; such evolution is treated as deletion and addition of files. The intended scale of application ranges from a handful up to roughly a dozen or slightly more variants. Runtime and memory consumption increase with the number of variants and the size and number of files but are expected to remain acceptable for small to medium-sized portfolios in the industrial control system context.

3.6 Output Model

The output of the extraction process is a representation of the software product line that is inferred from the given variants. Conceptually, this representation separates *common blocks*, which are present in multiple variants, from *unique blocks*, which are specific to subsets of variants. In the current prototype, these blocks are recorded in files of the form `blocks__S1-2-3-...-n.tsv`, which indicate which systems (variants) share which blocks.

The target vision extends this representation towards one logical artifact per feature, capturing all associated blocks across files and languages. Together, these feature artifacts form a *template* for the reconstructed SPL, which serves as a generative base. For each variant, a per-variant configuration specifies the set of features (or blocks) selected for that variant. A generator then takes the template, the mapping between system names and file names, and the per-variant configurations as input and produces regenerated variants as output. Ideally, these regenerated variants textually match the original inputs (modulo known limitations, such as handling of renamed files or non-deterministic ordering of independent blocks). Future work targets a richer representation for multi-file inputs that more explicitly encodes directory structure and file-level grouping.

3.7 Assumptions and Constraints

Our approach rests on several assumptions and constraints. We assume that all variants originate from a common ancestor via clone-and-own and that divergence has occurred through subsequent, mostly localized changes. Artifacts of interest are treated as ordinary text, including source code, DSLs, build files, and configuration files. The method does not rely on language-specific parsers, grammars, or meta-models and deliberately abstains from

exploiting syntactic or semantic structure beyond what is visible at the token level.

The extraction and regeneration operate at the level of textual blocks. Consequently, there is no guarantee of semantic or behavioral equivalence beyond textual reconstruction: regenerated variants are intended to reproduce the original text, but no additional behavioral analysis or verification is performed. Likewise, the current stage of the work does not infer feature names or an explicit feature model. Instead, it focuses on isolating and grouping code and text fragments that vary across variants and on enabling their regeneration.

3.8 Solution Requirements

Given this setting, we derive a set of requirements for a suitable SPL reengineering solution. First, processing must be *language agnostic* and must not depend on per-language frontends, so that the approach can be applied uniformly to arbitrary text-based artifacts, including in-house DSLs and semi-structured configuration files. Second, the approach should support *arbitrary text-based artifacts*, including code, DSLs, configuration files, and requirements documents, without making strong assumptions about their structure.

Third, the method must recover enough structure to regenerate all original variants from a single SPL representation, thus enabling full-circle reengineering. Fourth, the process should require only minimal manual effort and be primarily fully automatic, relying on default settings rather than extensive user guidance. Fifth, runtime must be acceptable for small to medium-sized variant sets of the size observed in the industrial control system context. Finally, the approach should be reasonably robust to moderate differences in formatting and layout, such as whitespace or comment changes, so that irrelevant syntactic variation does not dominate the analysis.

3.9 Intuitive Example

To provide an intuition for the kind of variability that our miner aims to capture, consider two variants of a component that support different combinations of features. In one variant, a check is implemented recursively and augmented with logging, whereas in the other variant the same check is implemented iteratively without logging. At a textual level, parts of the checking logic appear in both variants, while the recursive implementation

and the logging code are specific to the first variant and the iterative implementation is specific to the second. A token-based similarity analysis can isolate the shared checking logic as a common block, while treating the recursive, logging, and iterative parts as unique blocks associated with the respective variants. This illustrates how common and unique blocks can be identified without language-specific knowledge and how they can serve as building blocks for an extracted SPL representation.

References

Apel, Sven, Christian Kästner, and Christian Lengauer (2013). "Language-Independent and Automated Software Composition: The FeatureHouse Experience". In: *IEEE Transactions on Software Engineering* 39.1, pp. 63–79. DOI: 10.1109/TSE.2011.120.

Barr, Earl T. et al. (2015). "Automated Software Transplantation". In: *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, pp. 257–269. DOI: 10.1145/2771783.2771796.

Clements, Paul and Linda Northrop (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Czarnecki, Krzysztof and Ulrich W. Eisenecker (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Czarnecki, Krzysztof, Simon Helsen, and Ulrich W. Eisenecker (2005). "Formalizing Cardinality-Based Feature Models and Their Specialization". In: *Software Process: Improvement and Practice* 10.1, pp. 7–29. DOI: 10.1002/spip.213.

Fischer, Stefan et al. (2014). "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants". In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 391–400. DOI: 10.1109/ICSME.2014.61.

Galasso, Jessie (2025). "Formal Concept Analysis: A Structural Framework for Variability Extraction and Analysis". In: *CoRR* abs/2508.06668. arXiv preprint.

Kästner, Christian, Alexander Dreiling, and Klaus Ostermann (2014). "Variability Mining: Consistent Semiautomatic Detection of Product-Line Features". In: *IEEE Transactions on Software Engineering* 40.1, pp. 67–82. DOI: 10.1109/TSE.2013.45.

Kästner, Christian, Paolo G. Giarrusso, and Klaus Ostermann (2011). "Partial Preprocessing C Code for Variability Analysis". In: *Proceedings of the 5th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, pp. 127–136.

Kästner, Christian, Paolo G. Giarrusso, Tillmann Rendel, et al. (2011). "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, pp. 805–824. DOI: 10.1145/2048066.2048128.

Kuiter, Elias et al. (2018). "Getting rid of clone-and-own: moving to a software product line for temperature monitoring". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 179–189. ISBN: 9781450364645. DOI: 10.1145/3233027.3233050. URL: https://doi.org/10.1145/3233027.3233050.

Linsbauer, Lukas, Roberto Erick López-Herrejon, and Alexander Egyed (2017). "Variability Extraction and Modeling for Product Variants". In: *Software and Systems Modeling* 16.4, pp. 1179–1199. DOI: 10.1007/s10270-015-0512-y.

Martinez, Jabier et al. (2015). "Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach". In: *Proceedings of the 19th International Conference on Software Product Lines (SPLC)*. ACM, pp. 101–110. DOI: 10.1145/2791060.2791086.

Pohl, Klaus and Andreas Metzger (2018). "Software Product Lines". In: *The Essence of Software Engineering*. Ed. by Volker Gruhn and Rüdiger Striemer. Springer, pp. 185–201.

Schulze, Sandro, Sven Apel, and Christian Kästner (2010). "Code Clones in Feature-Oriented Software Product Lines". In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, pp. 103–112. DOI: 10.1145/1868294.1868310.

Souza, Leandro O. et al. (2025). "Software Product Line Engineering via Software Transplantation". In: *ACM Transactions on Software Engineering and Methodology* 34.2, 31:1–31:27.

Thüm, Thomas et al. (2011). "Abstract Features in Feature Modeling". In: *Proceedings of the 15th International Software Product Line Conference (SPLC)*. IEEE, pp. 191–200.

Wille, David, Önder Babur, et al. (2018). "Improving Custom-Tailored Variability Mining Using Outlier and Cluster Detection". In: *Science of Computer Programming* 163, pp. 62–84. DOI: 10.1016/j.scico.2018.04.002.

Wille, David, Tobias Runge, et al. (2017). "Extractive Software Product Line Engineering Using Model-Based Delta Module Generation". In: *Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, pp. 36–43.

Ziadi, Tewfik et al. (2014). "Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines". In: *Proceedings of the*

29th ACM Symposium on Applied Computing (SAC). ACM, pp. 1064–1071. DOI: 10.1145/2554850.2554931.