RESEARCH ARTICLE OPEN ACCESS

Fragment-Block: A Novel Approach for Managing Variability in Software Product Lines

Junior Cupe Casquina[1] | Leonardo Montecchi[2]

[1]Computing Institute, University of Campinas, Campinas, Brazil | [2]Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway

Correspondence: Leonardo Montecchi (leonardo.montecchi@ntnu.no)

Received: 31 March 2024 | **Revised:** 13 November 2025 | **Accepted:** 1 January 2026

Keywords: blocks | fragments | SPL evolution | SPL tools | SPLs | VRMs

ABSTRACT

Objective: Changes in the functionality of a software product line (SPL) can generate various issues in the SPL source code. Their complexity depends primarily on the amount of changes in the SPL source code and the internal mechanisms for managing variability. Further, modern development frameworks rely on a variety of artefacts besides source code. SPL developers need tools that can consistently search, segment, and analyse the SPL source code, to minimise undesired behaviours and reduce the time to implement new functionalities.

Methods: We introduce a novel approach and tool to manage SPLs source code. This approach streamlines the search and segmentation processes during source-code changes. To demonstrate its feasibility, we apply it as a proof of concept to two concrete SPLs: an implementation of the Elevator SPL from FeatureIDE; and a new SPL that includes generic textual artefacts. Further, we position our tool with respect to other existing approaches.

Result: The results demonstrated the feasibility of our approach. Differently from other tools for variability management, our approach supports both variability in time and variability in space, and supports generic textual artefacts, offering a similar flexibility as conditional compilation.

Conclusion: We proposed a new approach and tool for managing variability in SPLs. Our proposal leverages the concepts of blocks, features, and fragments to organise variability in textual artefacts, including, but not limited to, source code, A key characteristic of the approach is that it does not rely on any specific IDE, making it broadly applicable across different development contexts.

1 | Introduction

A software product line (SPL) or software product family (SPF) is a set of software systems (SSs) built using reusable assets [1]. Over time, various approaches and artefacts have been introduced to address the issues associated with implementing and evolving an SPL in diverse contexts. From the implementation point of view, for instance, conditional compilation (CC) is one of the most common variability realisation mechanisms (VRMs) [2], which is used to implement variability in the source code using preprocessor directives (e.g., #ifndef, #define, and #endif in the C Preprocessor). On the other hand, from the conceptual point of view, variability at a higher abstraction level is typically managed through a feature model (FM) [3], which provides a view of the features of the SPL, along with restrictions and relations between them. SPL end-users, including business professionals, often prefer to use a FM, as it allows for the graphical selection of features for a product. Once selected, these features are fed into a generator to produce the product variant, circumventing the need for in-depth knowledge of the SPL's internal implementation details.

Initially, SPL research centred on reducing the effort in developing similar products; however, today's focus has expanded to encompass various issues related to the erosion and evolution of SPLs. Several studies have recognised the need for solutions that deal simultaneously with the 'variability in space' and with the 'variability in time' [4, 5]. *Variability in space* refers to the concept that an artefact, or a portion of it, can exist in multiple forms at the same time [6]. Conversely, *variability in time* involves different versions of an artefact, or its parts, being valid at different times [6]. However, despite the growing number of studies addressing SPL evolution and erosion, a definitive solution that comprehensively tackles the challenges associated with them remains elusive in the research community and industry.

Technology is constantly evolving, and nowadays SSs are considerably different from the past. Nowadays, given the constantly changing and evolving nature of technology in the software industry, to be effective, SPL approaches need to operate transparently across various contexts and programming environments. For instance, SPL approaches need to function across different programming environments. Depending on the context and the changes developers aim to integrate into the SPL, old source code written in languages like Java or C++ is now often updated with its more concise and sophisticated versions, perhaps with the aid of some modern framework, or even completely replaced with new code written in other languages (e.g., Python3). Further, in the past, a software product variant typically required only the release of source code, binaries, and a few artefacts like the requirements document. Today, depending on the situation, a product variant may require variability in a broader range of artefacts, including detailed architectural designs, security models, build scripts, and extensive documentation and configuration for each component of the product.

One of the main reasons the industry needs approaches that work across different artefacts and environments is for the SPL to be prepared for both current and future changes in technology. An SPL implemented with a restrictive approach, such as being limited to Java source code, will generate complex issues for developers when, because of new features, the SPL will need to incorporate pieces of source code written in other programming languages. For instance, many of the SPL approaches proposed in the literature cannot be applied today to recent SPL projects that internally use today's popular technology stacks, such as the MERN Stack (MongoDB, Express.js, React.js, and Node.js), which includes React,[1] a popular framework to create rich user interfaces (UIs) today using JavaScript. Such incompatibility often arises from the specific constraints and limitations of each proposed approach, which focuses on a specific programming language or development approach. In addition, there is a need to update, maintain, and document the tools supporting existing SPL approaches, as many are outdated. Few integrated development environments (IDEs) specialised for SPL exist, such as FeatureIDE and pure::variants; these tools attempt to mitigate incompatibility by maintaining and updating specific SPL approaches within their systems. However, they do not encompass all approaches or mechanisms from the literature.

In order to contribute to state-of-the-art solutions that can work with various artefacts and programming languages, we introduce a new mechanism for managing SPL variability called fragment block (FB), which incorporates ideas from the mechanisms of CC, feature-oriented programming (FOP), and interchangeable components (ICs). This new mechanism operates in tandem with a classic FM: while the FM governs feature selection through established rules, our approach addresses the internal variability aspects of the SPLs. The FB approach primarily abstracts both common and distinct parts of the SPL source code into blocks and fragments. Then, conceptual relationships between products, features, fragments, and blocks are created to manage the variability. A block is defined as a piece of source code delineated by block directives (textual markers), and a fragment constitutes a collection of blocks that form a distinct unit. Additionally, a fragment can be regarded as a unique, non-repetitive part of an SPL.

The first key aspect of the FB approach is its ability to operate on general textual content. This feature enables developers to add variability to specific executable texts, facilitating the generation of diverse artefacts (e.g., images and short videos) using external generators, such as tools, platforms, or systems, including artificial intelligence (AI) prompts. For instance, the PlantText tool[2] generates images using text in a specific format as input, while systems like TeX Live process LaTeX files to produce PDF documents. Integrating SPL variability into LaTeX documents provides a means to generate large or complex PDFs with support for variability.

In complex scenarios where manual actions are prone to errors, developers can use scripts to orchestrate the inputs and outputs between these third-party generators. For example, a script can automatically rename and organise PlantText-generated images into a specific folder, ensuring their seamless integration into LaTeX documents. In scenarios of even greater complexity, integrating the variability directly into these orchestration scripts may be necessary. More generally, supporting generic textual content allows SPL variability to be incorporated into textual domain-specific languages (DSLs). In turn, this integration aids in creating specific artefacts needed for certain products. For instance, consider a DSL-based tool that generates security models reflecting the security configuration of a software product; by adding variability to textual DSL files specifying security properties, the tool can generate a model variant tailored to different products, which can then be used to fulfil the requirements for obtaining a third-party security certification for a software product.

The second fundamental element of the FB approach is the 'projection' operator, which encompasses the 'product derivation' operation. In the FB approach, a *projection* is the source code extracted from the SPL, with or without block directives (block marks). A projection can be created for the entire SPL, for a product, or for a single feature. The "product derivation" operation is accomplished when choosing to project a product without including the block directives. The FB approach encourages developers to manage the product derivations within Git repositories. This allows third-party tools to identify modifications in these Git branches to subsequently automate operations, including tasks like running unit tests or automatic source-code analysis. We provide a console tool, written in Java, called FB Tool that implements the “projection” operator and other operators related to the FB approach.

The third core element of the FB approach is the block graph. The block graph controls all the blocks in the SPL. Thus, when developers alter any node of this graph, they consequently modify the entire SPL structure. The block graph is similar to a list of lists of blocks, so insertions or deletions of elements must re-establish or create the links of the block graph to not break its structure. Conceptually, only this graph is necessary to generate products of the SPL. Besides, enriching this block graph with timelines, we can manage the variability in time. Our supporting tool uses a Git branch to store the contents of the blocks (text content), a graph database to manage the last block graph (i.e., the last instance of the SPL), and the commit history of the changes in the blocks to manage the changes in the timeline. The FB tool provides a list of commands to interact with the block graph and SPL regarding FB concepts (features, products, fragments, and blocks).

As a proof of concept, we present three scenarios related to the evolution of a modified version of the Elevator SPL from FeatureIDE [7]: the first scenario involves adding features to the system, the second scenario involves changing the functionality of one feature, and the third scenario involves removing a feature. To validate our approach, we have measured and compared the effort developers require when evolving the SPL source code using the CC as the internal VRM and the SPL source code using the FB as the internal VRM in these three scenarios. The results show that the FB approach is similar to CC when facing issues related to our three evolution scenarios. The Elevator SPL, used for testing some SPL approaches, is presented by the non-commercial FeatureIDE tool.

This paper is organised as follows. Section 2 provides an overview of VRMs. Section 3 discusses concepts and issues related to the evolution of SPLs. The foundations of our novel approach are presented in Section 4, while the supporting tool is described in Section 5. The three scenarios pertaining to FB-Elevator SPL evolution are presented in Section 6. The related work is presented in Section 7. Finally, Section 8 presents the conclusions and future work of this paper.

2 | Variability Realisation Mechanisms

VRMs are low-level mechanisms that manage the SPL variability in artefacts that belong to the SPL (e.g., SPL source code). VRMs influence how to organise or handle the SPL's software architecture or source code. VRMs like inheritance, parameterisation, CC directives, and dynamic libraries are some of the most common [8], but they all focus on the SPL source code. This section will present the CC, FOP, and IC VRMs. These VRMs inspired the creation of the FB approach presented later in Section 4. From the CC VRM, we took the ideas of directives and preprocessing; from the FOP VRM, we took the idea of separation of concerns; and from the ICs VRM, we took the idea of interchanges.

2.1 | Conditional Compilation

CC is one of the most important and popular techniques for implementing variability [9]. This mechanism focuses on using preprocessors, allowing developers to annotate the SPL source code with directives to include or exclude code depending on feature selections. A preprocessor is a secondary programme that processes the source code before passing it to the compiler. The most representative preprocessors in the context of SPLs are the C preprocessor (CPP), Munge,[3] and Antenna.[4] The C preprocessor adds variability to the source code written in C [10], while the Munge and Antenna preprocessors add variability to the source code written in Java. To manage the variability in the SPL source code, developers create conditional statements like “if a certain feature is selected, include a specific part of code”. CPP defines Boolean directives such as `#if`, `#elif`, `#else`, and `#endif` to add variability inside the source code. In a similar way, Munge looks for conditional directives like if[tag], else[tag], if_not[tag], and end[tag] within comments to manage the variability in the source code; here, the tag label represents a specific feature. Antenna uses directives like “`#if`expression”, “`#elif` expression”, “`#else`”, “`#endif`”. Antenna even allows us to use “`#condition`expression” in the first line of a file to indicate whether to continue processing the following directives.

2.2 | Feature-Oriented Programming

FOP is a specialised form of generative programming [9], a computing paradigm that automatically creates entire software families by configuring assemblies of primary and reusable components in different arrangements [11]. To achieve separation of concerns, FOP encapsulates features into separate modules. Later, according to the configuration of a product, some of these modules are composed or mixed to generate a variant, based on some structured information about the features provided by modules (e.g., feature structure tree [FST]). During this variant creation, the sequence in which the modules are combined matters [9]. FOP uses a generic approach to manage the variability so that the FOP approach can be used in diverse languages such as Java, C#, and C. AHEAD,[5] FeatureHouse[6] and FeatureC++[7] are some of the tools that support FOP. In the case of FeatureHouse, composing is by superimposing FSTs [9] from the modules, considering their names, types, and relative position of the FST elements. A FST is a hierarchical representation of the structural elements of a module [9]. The structural elements included in the tree depend on the language; for a module written in Java, for example, the FST represents packages, classes, fields, and methods.

2.3 | Interchangeable Components

In the software industry, the term ‘component’ is employed indiscriminately to denote a ‘part of something,’ and more often to refer to a part of the architecture. On the other hand, the fundamental principle of the IC mechanism involves interchanging components, akin to playing with LEGO blocks, where one can interchange or reuse LEGO pieces from a set, such as a cityscape [12]. Within the ICs mechanism, a component can be replaced by another only if the new component implements all the external interfaces of the original. In the context of SPLs, the ICs mechanism utilises the variability produced by managing ICs to control SPL variability. The idea of interchanging components is not only used in the SPL context but also in other contexts, such as the Self-adaptive Systems and Dynamic SPLs, to manage the variability at runtime. For instance, in a specific study on self-adaptive systems (SASs), the researchers explored the use of ICs to manage runtime variability. This approach involved mapping features to a set of components based on the DyCosmos model, a model for implementing components in source code, and dynamically generating variants (ICs) using a FM at runtime [13]. Pharo Smalltalk, a tool and environment for the Smalltalk programming language, and OSGi platforms, which allow loading or unloading components on the fly [14], are both adaptable to support the ICs mechanism. Smalltalk is a programming language that supports the idea of ICs thanks to its dynamic typing and object-oriented paradigm [15], allowing the substitution of one object (or component) for another as long as they adhere to the expected interface. Eclipse can be viewed as an SPL that internally adopts the ICs VRM through the Equinox tool, particularly if one considers all the official Eclipse plugins as features and each customised Eclipse IDE as a distinct variant. Equinox is an Eclipse project that implements the OSGi core framework specification.

3 | Software Product Line Evolution

3.1 | Software Evolution

A SS evolves mainly in two different dimensions: time and space [4]. Typically, when software evolution is more focused on the time dimension than the space dimension, we refer to traditional software, which is commonly managed through version control systems (VCSs). On the other hand, when software evolution is more focused on the space dimension than the time dimension, we refer to traditional SPLs, commonly managed through an FM and one or more VRMs. Figure 1 illustrates a review of the evolution of a SS in its two dimensions. On the horizontal axis, the labels $t_0, t_1, \ldots, t_n$ represent specific instances of time over the SS evolution; t_c represents a constant time. Instances of time highlighted in red, like t_0 and t_1, emphasise moments when new products were added to the SPL. On the vertical axis, $c_0, c_1, \ldots, c_7$, represent different product configurations. Rectangles here depict new products or versions being introduced in the SPL, and lines connected by an arrow and a dot represent periods in which a product remained unchanged. The figure illustrates the software company's periodic addition of new products to the SPL. It also shows that, after t_3, the software company ceased the product maintenance related to configuration c_0.

While the evolution of software in space and time is common, it is difficult to have an effective visualisation of how such evolution takes place. Our Space-Time diagram in Figure 1 can be used as an evolution review diagram of the products of an SPL in practical circumstances, where the number of products is relatively low. However, there may be cases where it is necessary to show an evolution review diagram that shows all the possible product configurations of the SPL along with the time dimension in some way. Today, there is a need for a widely accepted, readable, and straightforward diagram or model to represent the evolution of the SPL in all the cases previously mentioned. Some authors have presented models, such as the Hyper Feature Model [16] and the Temporal Feature Model [17], to address this issue, but these have yet to gain wide acceptance. Given the substantial amount of data arising from the SPL evolution, it is challenging to represent it within a single diagram. We believe there is a need for a comprehensive model that encapsulates all this evolutionary data. However, we were unable to find or create a model that meets these criteria. This model should provide varied views that depict the SPL evolution in diverse forms. Each view should emphasise specific data subsets from the SPL evolution, tailored to the needs and context of developers or users of the SPL.

3.2 | Evolution Issues

In a software organisation, the groups responsible for developing an SPL face various issues, depending on the context in which the implementation or update of the SPL takes place. In some

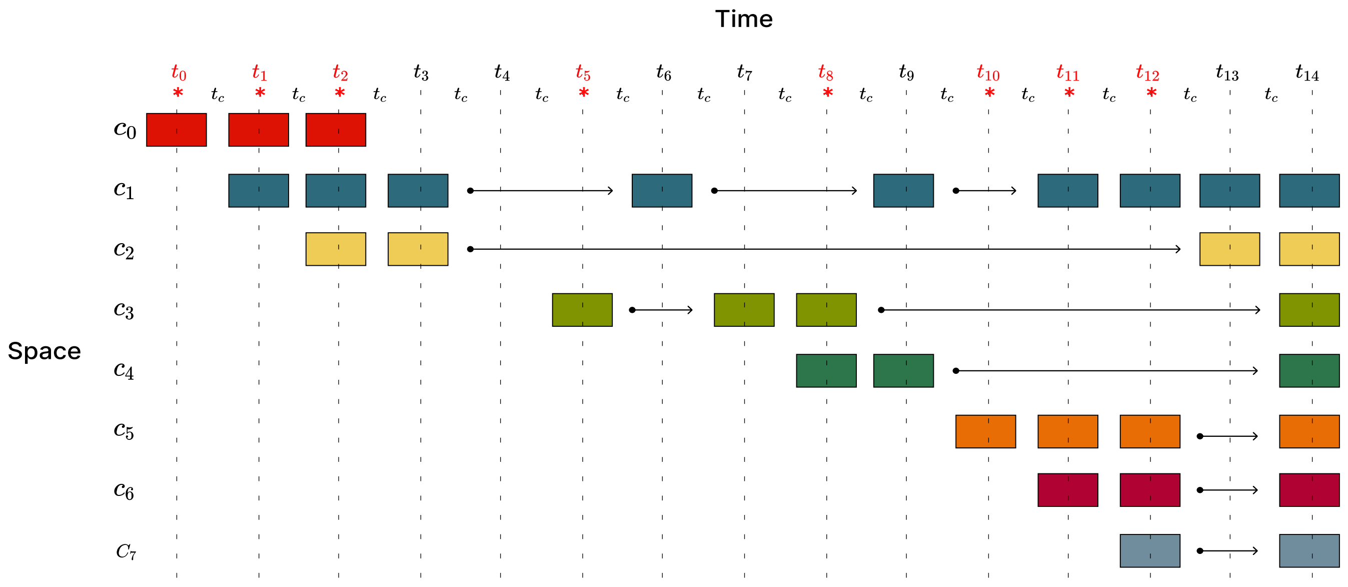


FIGURE 1 | Space-time diagram of the evolution of a SS.

cases, they have to start a new SPL from scratch, while in others, they have to deal with legacy products that need to be turned into an SPL. In addition to these scenarios, the group must decide on which set of product configurations to focus, and whether to focus on specific product configurations or on all possible configurations. This decision can significantly increase or reduce the complexity of the SPL implementation. This subsection lists two of the most common issues encountered during the implementation of an SPL.

3.2.1 | Feature Location

Feature (or concept) location [18] is an activity that aims to identify the source code's parts corresponding to specific functionality [19]. Depending on the context, this search can be applied to either a SS or a SPL source code. While developers typically perform this task manually when working on an SS, the challenge intensifies in the SPL context. In the SPL context, depending on the adopted VRM, some features are more easily located than others. Commonly, features from the FM are identified by SPL tools or by developers using the internal structures inherent to the chosen VRM, such as the conditional directives in the CC mechanism. However, as the software evolves, not all the features get integrated or referred to in the VRM structures or in the FM, and thus, locating code implementing specific functionality can become increasingly intricate. The task remains predominantly manual, as there are no widely recognised or accepted tools available for feature location in an SPL when the features are not (or not yet) represented in the FM.

The FB approach introduced in this paper, similar to how the FeatureIDE tool colours source code by features for SPL projects using the CC VRM, allows locating features of the FM in the SPL source code using the projection operator. Furthermore, the internal structure of the FB approach, like that of the CC VRM, provides the basis for third-party developers to implement similar feature-colouring views in IDE tools like FeatureIDE. Some researchers added the time parameter to the feature location activity [20], thus posing the problem of how to find a feature in the SPL source code from a specific time or past version.

3.2.2 | Maintaining Product Consistency

Changes in the SPL source code can significantly impact the entire family of products. The extent of this impact, and thus the overall consistency, depends on the type of VRM employed, since it is the VRM that dictates how to manage the SPL variability in the source code. Some approaches, such as CC, place the burden of ensuring product family consistency on the developers themselves. Conversely, other approaches, like OSGi, mitigate the consistency issues by limiting variability management. For instance, Eclipse, leveraging Equinox, allows the transparent addition or removal of plugins from different vendors, maintaining consistency. This stability is due to each plugin defining or implementing well-defined interfaces and extension points, which enhances the overall system stability. However, when a widely used extension point is modified, all the dependent plugins are automatically broken until updated by a developer.

Maintaining the consistency of the whole family of products is an essential task while evolving an SPL. Impacting released, stable products within the product family could lead to significant losses and compromise the software development company.

4 | The Fragment-Block Approach

Our Fragment-Block (FB) approach manages SPL variability using fragments and blocks. A *block* represents a string of sequential characters within the SPL source code, while a *fragment* represents a group of blocks. As a result, fragments can represent both sequential and non-sequential portions of source code, while a block could represent a sequential portion of source code, either a short number of characters or several sequential lines. The only restriction between fragments and blocks is that fragments cannot share blocks (i.e., fragments cannot overlap). From the point of view of features and products, a fragment is part of one or more features, while a feature is part of one or more products. Figure 2 shows an example of how a feature (`feature ...00`) is linked to two fragments (`frag ...101` and `frag ...011`) that, at the same time, are associated with different blocks in a source-code file.

The FB approach defines two main operators called *projection* and *syn*. The *projection* operator enables the projection (with or without block directives) of the source code for specific SPL products, features, or the entire SPL into a specific directory. When block directives are omitted, the operator generates released products. Conversely, when block directives are included, it generates an image or instance that serves as a base for evolving blocks in the source code. The *syn* operator synchronises locally altered blocks into the database.

4.1 | Blocks

Blocks mainly represent parts of the SPL source code. One restriction of the FB approach is that all the source code of an SPL has to be associated with some block. The source code that is not associated with a block will be lost. All the blocks are organised in a graph called the *block graph* of the SPL, which is composed of

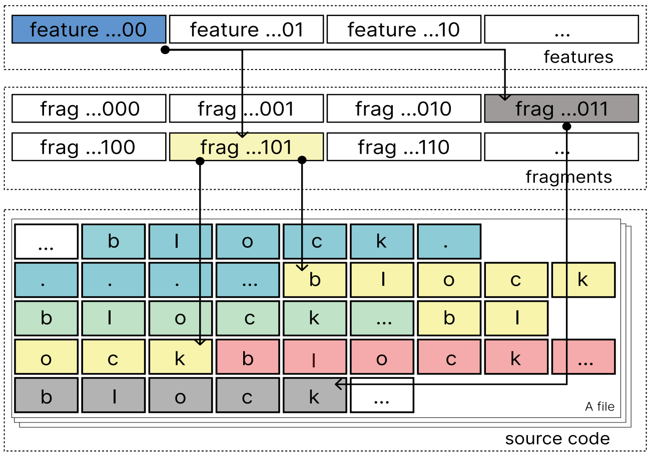


FIGURE 2 | Relationship between features, fragments, and blocks.

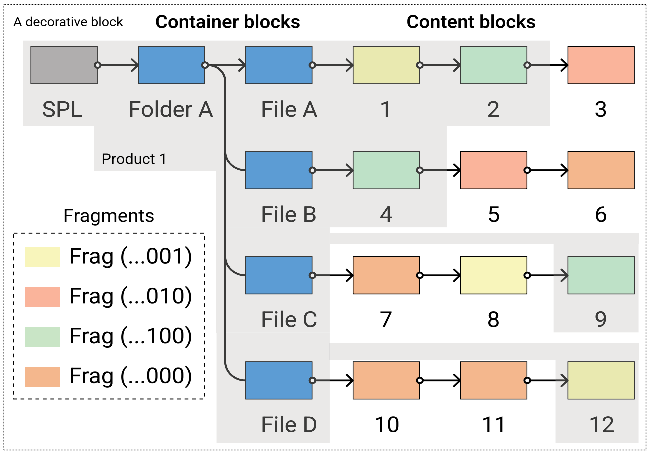


FIGURE 3 | Block graph from an illustrative SPL.

three types of blocks: Container, Content, and Decorative. *Container* blocks represent files and folders of the SPL. *Content* blocks represent sequences of characters stored in files, while *Decorative* blocks are virtual blocks used to enhance the readability and organisation of the block graph.

The block graph is used to manage the diverse scenarios related to the evolution of the SPL. Figure 3 shows an illustration of a block graph. It shows a decorative block called "SPL", several container blocks representing folders and files, and several content blocks representing the source code inside files. Each content block is associated with only one fragment. Therefore, to generate a specific product, it is necessary to identify and exclude content blocks that are not part of the fragments associated with any of that product's features.

4.1.1 | Operations Over the Block Graph

A Block Graph is a directed graph where each node represents a block, with each block having a pointer to its content, which may be stored in a different space or database. The FB approach uses some primitive operations to manage all the blocks in the block graph. Since a Block Graph is directed, its nodes and edges are used to implement some of the operations.

Add a block: New blocks are added to the block graph either before or after the selected pivot block, in a manner similar to adding a node in a linked list.

Edit a block: After navigating the block graph to locate the specific block, it is edited using its content pointer.

Remove a block: This operation involves traversing the block graph to locate the specific block for removal, akin to removing an element in a linked list. If necessary, the content of the block is removed using its pointer.

Label a block: This adds a label to a block, which specifies the fragment that owns the block. The block is identified by traversing the block graph. Each block is labelled with only one fragment.

Split a block: This operation splits a block's content into two or more blocks. First, the selected block is located, and the content to be split is identified (using some flag in the content). Then, the content is split, and the pointers of the newly created blocks are adjusted, similar to adding and removing elements in a linked list.

Change type: A block's type can be changed to modify its behaviour in the block graph. This is an internal operation, and it is often used to streamline block creation during the evolution of the block graph. For instance, if the developer deems it appropriate, a decorative block may become a content or container block.

4.1.2 | Managing Blocks

Users of the FB approach can follow different strategies to add new blocks to the block graph. One such strategy involves projecting the entire SPL's blocks into files. In this projection, a copy of the SPL's source code is stored in a folder, with the content of each file separated according to the blocks. Once the sequence of possible blocks in a file is visible, users can add the source code for new blocks and label each with a specific fragment using the primitive operations mentioned above. The concept is similar to coding in a Python Jupyter Notebook, where users can alter the source code within the cells and access additional functionalities, such as appending comments. Note that, in this case, each file consists of *all the possible* blocks of that file, depending on the features that may affect it. Another strategy is to project only a specific product. Blocks are projected to files as described in the previous strategy, but in this case, *only blocks pertaining to a single product* are visible. Developers can therefore work with a consistent view of the SPL code and focus on the implementation within a specific product.

When projecting an SPL with block directives, users of the approach have a perspective of the block graph regarding blocks, files, and folders. Thus, adding, editing, or removing a block within the block graph is similar to inserting, editing, or removing an element in a multilevel linked list, all of which is transparent to the user. To synchronise added, modified, or removed blocks from the local block graph to the master block graph, users must apply the `syn` operation. Conversely, when projecting only a specific product with block directives, the process of inserting, editing, or deleting a block differs slightly. In this case, without a local reference to the complete graph, users must rely on a pivot block as a reference for inserting new blocks. When synchronising the block with the master graph, the new block should reference the pivot block and be inserted adjacent to it in the master block graph. For removing or editing a block, only the ID of the block is needed to make changes in the master block graph. Some of these tedious activities can be automated and implemented by a specialised tool. In this paper, we introduce the FB Tool as a prototype tool supporting the proposed FB approach.

4.2 | Fragments

In our approach, we introduce the concept of *fragments* as an additional level of abstraction, to decouple features from their source code (blocks). In fact, having a direct connection between features and blocks would introduce different drawbacks.

The first concerns maintaining the no-overlap property, which is crucial for reliably projecting source code with directive blocks. Without fragments, this property would have to be applied at the

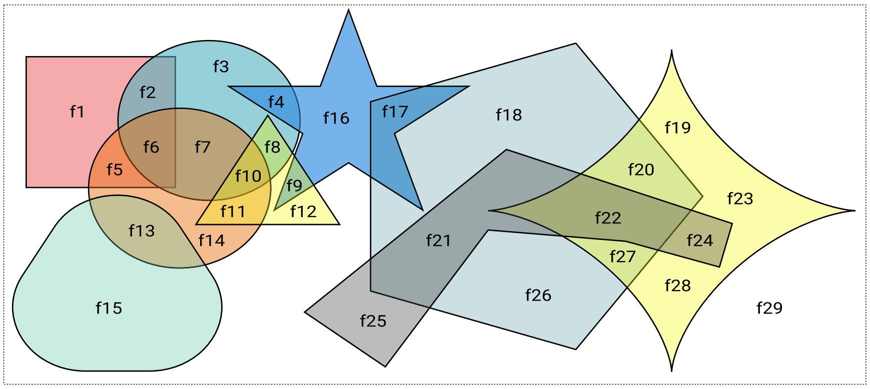


FIGURE 4 | A representation of an SPL in Fragments.

feature level, thus constraining the SPL developer to disjoint features, which is not realistic in practice. One solution to maintain this property without the fragment level is the creation of new features. However, the lack of an intermediate level leads to the generation of numerous features, excessively deforming the original FM. The fragment level hides this complexity from the user and reduces an exaggerated deformation of the FM.

In the FB approach, a fragment is a logical expression based on features, representing one of the many possible combinations of features in terms of their intersections. The FB approach assumes no-overlap at both the fragment and block levels. However, since a block is associated with at most one fragment, the user is required to ensure consistency only at the block level. In practice, most of the possible fragments are not used in an SPL, for example, because that combination of features is not possible, or because the source code associated with that combination is empty. This can happen when no specific code is needed to handle a certain combination of features (e.g., $f_1 \wedge f_2$), because the base code of the individual features is enough.

Figure 4 shows a graphical representation of an SPL using no-overlapping fragments. The union of these fragments forms various geometric figures, each representing different features of an SPL. For instance, a square feature could be created by combining the f1, f2, f5, and f6 fragments, or a star feature using the f4, f9, f16, and f17 fragments. The generation of a product is defined following the FM restrictions between features and the product configuration.

The FB approach specifies a unique fragment not associated with any product. This fragment is functional when users wish to retain a block within the master block graph without including it in any product of the family of products. By tagging a block with this unique fragment, users can ensure its exclusion from any product generated by the approach.

4.2.1 | Types of Fragments

The FB approach classifies the fragments into three types: A, B, and C. Fragments of type A represent intersections between features. Fragments of type B represent the union of features intersecting the symmetric difference of all the features (that is, they contain source code that is most specific to a certain feature only). Fragments of type C represent the difference between the universe and the union of all features. Figure 5 shows the three types of available fragments corresponding to four features: F1, F2, F3, and F4. In an SPL with four features, developers can use 16 possible fragments in the FB approach. Here, fragments with the binary codes 0001, 0010, 0100, and 1000 belong to type B; fragments with the binary codes 0011, 0110, 1100, 0111, 1110, 1010, 1111, 1010, 1101, 1011, and 1001 are of type A; and finally, the fragment with the code 0000 is categorised as type C. Note that these are the *possible* fragments that developers can use to label source code (in the form of blocks) that is only needed when a particular combination of features is included in the product.

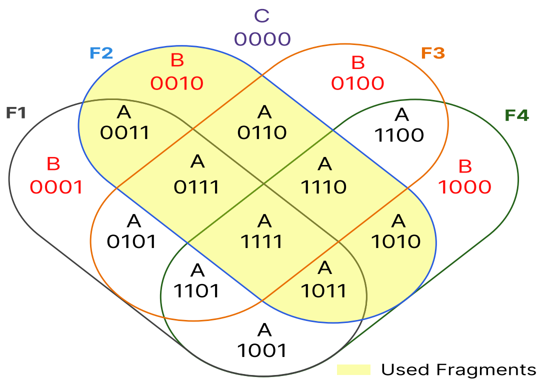


FIGURE 5 | Possible fragments for an SPL with four features.

On the other hand, the fragments filled in yellow in Figure 5 (0011, 0010, 0111, 0110, 1110, 1111, 1010, and 1011) represent the fragments currently used in the SPL by the developers. In this example, they are all related to the F2 feature; note in fact that they all have 1 in the second bit from the right (i.e., F2 is selected). However, each of them represents different associations with the other features, and they represent code that serves different configurations of the system. For example, fragment 1010 signifies an association with features F2 and F4 but not with F1 and F3, indicating it should be selected for product configurations that include F2 and F4 but exclude F1 and F3.

4.2.2 | Managing Fragments

In the FB approach, the number of features of the SPL limits the universe of fragments that can be used. Thus, new blocks, containing, for example, new source code, inserted into the SPL must be labelled with a fragment from the list of available fragments. If the new code to be added to the SPL does not fit the conditions of any of the fragments in the list, the developer must create a new SPL feature, which will enable additional fragments. Once additional fragments are enabled, the developer should label their new code with an appropriate fragment.

When users remove a feature, they generally also eliminate a set of available and used fragments, thereby reducing the SPL's total fragment count. Here, users of the approach can follow two removal methods:

i. removing the fragments only from the list of used fragments so these are still listed in the available fragments, or

ii. completely removing the fragments also from the list of available fragments, and then updating the other fragments that are related to these removed fragments. Currently, the support tool (see Section 5) implements only the first option.

So far, we have discussed how fragments are managed when developing an SPL in an FB environment. Migration of existing SPL source code organised according to other VRMs is out of the scope of this paper. Since differences exist between concrete techniques for variability management, refactoring and other reengineering techniques must be applied to bridge the gap between different implementation techniques [21].

4.3 | Projections

The projection of a product is a fundamental part of the FB approach. Three elements are needed to project a product: the list of the product's features, the mapping between features and fragments, and the mapping between fragments and blocks. Typically, constraints between features are managed at the FM level, so that when selecting a new product configuration, the possible configurations are restricted by the relations between features described in the FM. The FB approach presented in this paper is independent of how features for a specific product are selected; once the list of selected features is established, all the fragments related to them are collected. Finally, since there is a mapping between fragments and blocks, the product is generated from the source code of blocks associated with those fragments.

Our approach supports projecting source code with or without block directives. The projected source code can be related to a single feature or a set of features, encompassing all features if needed. Similarly, it can pertain to one product or a collection of products, which may include all products, as well as to one fragment or a set of fragments, and to a block or multiple blocks. The most useful projections typically relate to Products, Features, or the entire SPL, encompassing all features.

4.4 | Evolution

The FB approach defines a predetermined algorithm that users can use to handle the evolution of an SPL. However, they can use any other method or algorithm to manage this evolution. Algorithm 1 utilises SPL source code, a product list, and a deadline date as inputs to evolve the SPL. The product list in the algorithm limits the number of products to preserve consistency. The algorithm iterates the product list by applying Algorithm 2 to each product. Algorithm 2 focuses on a product and how to integrate the approved variations into it at the time of generating the product without affecting the consistency of the other products. Algorithm 2 has two primary methods: evolveSPL() and analyseVariations(). The evolveSPL() method integrates the calculated variations by comparing the initial state of a product with its varied form. In this integration, the users can use the tools provided by the selected VRM, in our case, the FB Tool of the FB approach (see Section 5). The analyseVariations() method examines the differences between the intended changes and the actual variations.

ALGORITHM 1 | Simple sequential evolution algorithm for a software product line (*Simple_Seq_EA_for_SPL*).

```
Require: An input SPL ($SPL_i$), An input
         product list ($prodList_i$), An input
         deadline date ($deadlineDate_i$)
 1: $sentinel \leftarrow$ false
 2: $SPLTeam \leftarrow$ A development team assigned to
         the project
 3: if $SPL_i$ does not exist then
 4:     $SPL_i \leftarrow createEmptySPL(prodList_i)$
 5: end if
 6: while (currentTime <= $deadlineDate_i$) &&
         ($sentinel$ == false) do
 7:     for each $p$:Product of $prodList_i$ do
 8:         $\Delta_p \leftarrow SPLTeam$ defines the variation
            for $p$
 9:         $deadlineDate_p \leftarrow SPLTeam$ define a
            deadline date for $\Delta_p$
10:         if $SPLTeam$ does not want to end the
            evolution process then
11:             Simple_Seq_EA_for_SP($SPL_i$, $p$,
                $prodList_i$, $deadlineDate_p$, $\Delta_p$)
                                   ▷ Focus on one product
12:         else
13:             $sentinel \leftarrow$ true
14:             $break$
15:         end if
16:     end for
17:     $sentinel \leftarrow$ true
                          ▷ No more family of products
18: end while
```

4.5 | Integration With Complementary Approaches

One of the keys to popularising a new development approach is how it integrates with current and popular development tools. This subsection demonstrates the integration of our approach with some of the most widely used development approaches and tools in the current industrial practice.

4.5.1 | Branching Models

VCSs are today at the core of software development. Developers typically use branches to manage the short-term changes to the source code and to synchronise the work of different developers.

ALGORITHM 2 | Simple sequential evolution algorithm for a software product (*Simple_Seq_EA_for_SP*).

```
Require: An input SPL ($SPL_i$), An input
      product ($prod_i$), An input product list
      ($prodList_i$), An input deadline date
      ($deadlineDate_i$), An input required
      variation ($\Delta_i$)
Ensure: $prod_i$ is in $prodList_i$
 1: $sentinel$ ← false
 2: while (currentTime <= $deadlineDate_i$) &&
      ($sentinel$ == false) do
                                      ▷ Evolving the SPL
 3:    $evol$ ← $defineProductEvolution(prod_i, \Delta_i)$
 4:    $prod$ ← $evolveProduct(prod_i, evol)$
 5:    $\Delta$ ← $compareProducts(prod_i, prod)$
 6:    $SPL_{new}$ ← $evolveSPL(SPL_i, \Delta, prod_i)$
                                 ▷ Testing and Evolving
 7:    for each $p$:Product of $prodList_i$ do
 8:       $tProd$ ← $generateProduct(SPL_{new}, id$ of $p)$
 9:       $test$ ← $testProduct(tProd)$
10:        if $test$ != $OK$ then
11:           $oldProd$ ← $generateProduct(SPL_i, id$ of $p)$
12:           $temp\Delta$ ← $compareProducts(oldProd,$
                $tProd)$
13:           $\Delta_{new}$ ← $analyseVariations(\Delta, temp\Delta)$
14:           if $SPLTeam$ does not want to end
                the process for $prod_i$ then
15:                Simple_Seq_EA_for_SP($SPL_{new}$, $p$,
                   $prodList_i$, $deadlineDate_i$, $\Delta_{new}$)
16:           else
17:              $sentinel$ ← true
18:              $break$
19:           end if
20:        end if
21:     end for
22:     $sentinel$ ← true
23: end while
```

A project typically follows a certain branching model,[8] that is, a way to create and manage such branches.

When projecting a product using the FB approach, the blocks from the projected product are synchronised with the database holding the master block graph. However, this synchronisation is temporary, as developers continuously evolve the product in parallel. Consequently, the blocks that are added, edited, or removed from the projection need to be periodically synchronised (pushed).

Developers usually work on separate branches and periodically merge or synchronise their changes into a main branch, often called 'master.' Upon this synchronisation (merge), FB blocks that are not updated can be synchronised as well. In this manner, they can continue working on different branches as usual, modifying the blocks and the source code within them. Furthermore, synchronisation can, in principle, be automated using continuous integration (CI) tools, such as Jenkins.

As an illustration, Figure 6 shows a GitFlow-based branching model, with common branches such as Main, Development, and Feature, where commits from any developer to the main branch require synchronisation with the block graph at some point.

4.5.2 | Other VRMs

A block in the FB approach wraps pieces of source code. The source code contained in a block, in turn, is considered as a black box that can contain any type of source code, including source code with variability (that is, implemented with some of the existing VRMs). For example, a block may contain source code from a component following the Cosmos* or DyCosmos implementation model, or source code containing CC directives. More generally, it can contain other types of textual data, such as source code for generating models or documents.

Since we can integrate other kinds of variability into the blocks, we can externally manage the projected source code to be processed with a different VRM or tool. This way, a developer could generate documents, binary programmes, or models with an external tool using some of the generated files as inputs. The order of generation depends on the context, as some inputs being processed originate from various external tools.

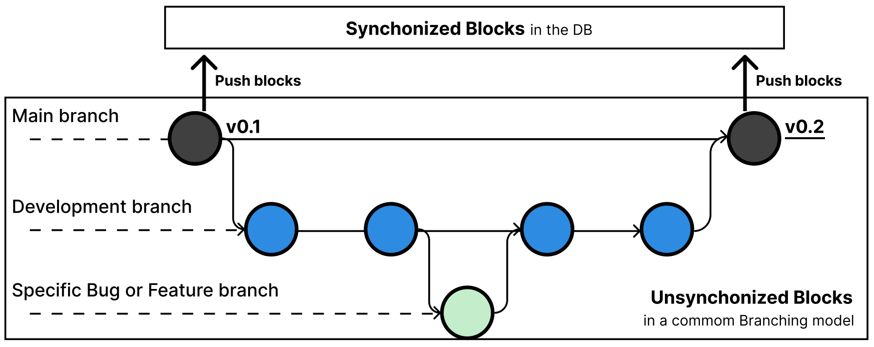

Synchonized Blocks in the DB
Push blocks
Push blocks
Main branch
v0.1
v0.2
Development branch
Specific Bug or Feature branch
Unsynchonized Blocks
in a commom Branching model

FIGURE 6 | Integration of the FB approach in a GitFlow-based branching model. Blocks can be in a synchronised state (in the *main* branch) or in an unsynchronised state (in other branches).

5 | The Fragment-Block Tool

We have implemented a prototype tool to support the application of the FB approach. In this section, we describe the tool architecture (Section 5.1), we discuss the currently available functionality (Section 5.2), and we provide an overview of its usage (Section 5.3).

5.1 | Architecture

The tool's architecture contains three main components: The *Git Storage* component, the *Block Database* component, and the *Console* component. The *Git Storage* component manages the functionality related to Git, including archiving blocks in Git. The *Block Database* component manages the functionality related to the database, including managing the block graph. The *Console* tool component manages the functionality related to the console commands to interact with the developers of SPLs, the tool users.

The current prototype uses Neo4j as the database component. The tool internally creates a Git branch to store the SPL's FB blocks after analysing the SPL's source code. The fragments and blocks of the SPL are instead maintained in the database.

5.1.1 | Database Model

The FB tool uses a Neo4j graph database to manage the products, features, blocks, and fragments, including the block graph of an SPL. The diagram in Figure 7 shows an overview of the essential part of the database schema. Resources in the SPL (e.g., source files) are organised within Container elements. A Container can serve as a parent to other Containers, or it can represent a single file. When representing a file, the Container has a pointer to the initial Block element, which allows access to the other blocks and thus the file content in its possible variants. A Block points to the next block of the file, and each Block is associated with a Fragment. A Fragment is related to a set of Indices. A Product is associated with a group of Features, and a feature has an Index. The relation between fragments and features is therefore determined by the Indices.

5.1.2 | Block Content in Git

The tool does not store the content of a block inside the database. Instead, the content of blocks is stored in files, in a separate branch used exclusively by the FB Tool. Each file in that repository corresponds to one *block* of the SPL source code. Therefore,

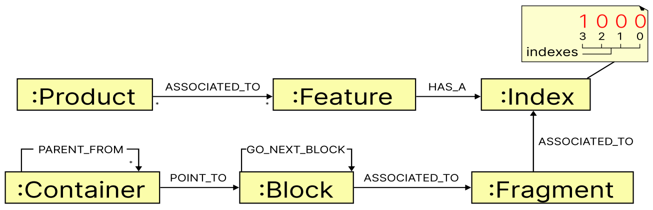


FIGURE 7 | Overview of the database schema of the prototype Fragment-Block Tool.

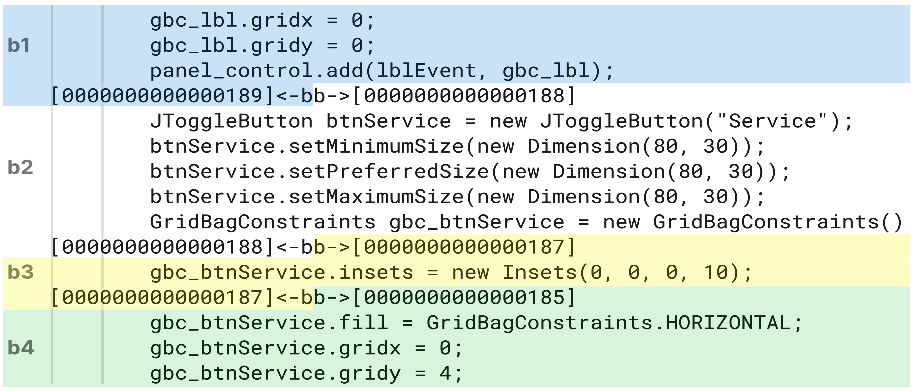
the history of each block can be tracked by exploiting the Git functionality: any change to the content of a file in the FB Tool repository represents a change to the content of the corresponding block, and therefore a change to the SPL source code. Since these changes are committed in an internal branch, Git stores a history of the evolution of blocks through commits. Thus, users can use an external tool to analyse these commits, gaining insight into the evolution of FB blocks.

5.2 | Functionality

The tool provides a list of commands that users can employ to control the block graph of an SPL and to manage other aspects of the FB approach. Table 1 shows a list of the most common commands of the tool, along with a brief description of each of them. Available commands include commands for analysing or syncing the source code target, reviewing the database contents, altering the database, and setting up the tool and managing projections.

TABLE 1 | Main FB tool commands.

Commands	Description
fb project-spl	To project all the blocks of the SPL.
fb project [product]	To project a product using block directives.
fb configure	To configure the tool, creating the FB Git branch.
fb analyse	To analyse an initial source code without directives.
fb sync	To analyse source code with directives and prepare the block updates.
fb add	To add blocks to a temporary space.
fb commit	To commit the modified blocks into the FB Git branch.
fb upsert-product --new [product] [features]	To create or update a product.
fb upsert-feature [feature] [order] [feature-name]	To create or update a feature.
fb upsert-fragment [fragment] [fragment-name]	To create or update a fragment.
fb associate-frag-to-index [fragment] [indices]	To assign indices to a fragment.
fb tag-blocks [fragment] [blocks]	To tag a fragment to a list of blocks.
fb inspect-files --all	To show a review of the blocks in each file.
fb list-products	To show the list of products.
fb list-features	To show the list of features.
fb list-fragments	To show the list of fragments.



```
        gbc_lbl.gridx = 0;
        gbc_lbl.gridy = 0;
        panel_control.add(lblEvent, gbc_lbl);
[0000000000000189]<-bb->[0000000000000188]
        JToggleButton btnService = new JToggleButton("Service");
        btnService.setMinimumSize(new Dimension(80, 30));
        btnService.setPreferredSize(new Dimension(80, 30));
        btnService.setMaximumSize(new Dimension(80, 30));
        GridBagConstraints gbc_btnService = new GridBagConstraints()
[0000000000000188]<-bb->[0000000000000187]
        gbc_btnService.insets = new Insets(0, 0, 0, 10);
[0000000000000187]<-bb->[0000000000000185]
        gbc_btnService.fill = GridBagConstraints.HORIZONTAL;
        gbc_btnService.gridx = 0;
        gbc_btnService.gridy = 4;
```

FIGURE 8 | A view of the projected blocks related to a product of the FB-Elevator SPL.

5.2.1 | Projecting Blocks

The projection operation extracts the source code of a single product from the SPL. The command `fb project [product-name]`, without any additional flags, projects a product with block directives. On the other hand, the command `fb project --clean [product-name]` projects the source code without the block directives. Users can employ the result of this latter command to test a specific product of the SPL. Figure 8 shows part of a file of source code generated after executing the `fb project [product-name]` command. It shows four blocks: b1, b2, b3, and b4, separated by block directives, that is, markers of the form `b->[blockId]` and `[blockId]<-b` that indicate the beginning and end of a block. Block directives may also appear on the same line.

Users of the FB approach can use the projections to modify blocks or test certain products according to the particularity of the SPL project. For instance, developers can run a series of unit tests on a specific product using Selenium, or manually test its functionality by deploying it on Heroku after its projection. The possibility to integrate the proposed approach with tools of CI/CD facilitates the development process of an SPL.

5.2.2 | Evolving Blocks

Like CC directives, block directives are used to separate the blocks in the source code of a projection. These directives assist developers in analysing the source code, but they are also valuable to the supporting tool, which relies on them for detecting modifications within the blocks and the block graph when users modify the SPL's source code. Once the code of the SPL is projected, users can modify, split, add, or remove the blocks of the SPL as a result of the development of the system. All these operations are performed by simply modifying the projected source code (with directives), following specific patterns.

To modify a block, the user only needs to edit the text between the block directives. To add blocks, the user only needs to edit the files, adding text outside of the boundaries of existing block directives; the tool then recognises that a new block needs to be created. To split a block, they only need to add a `b>> [cut]` marker inside a block in the position where they want it to be split. To remove a block, they only need to remove the text related to the block. To remove a block from the SPL codebase while keeping a copy, it is also possible to label the block with another fragment.

Finally, these changes are stored in the database and in the internal Git branch. To analyse and prepare the updates in the database, users of the tool can use the `fb sync` command. The `fb add` and `fb commit` commands are needed to add these changes to the internal Git branch. The `fb add` command moves files from a temporary to a permanent folder, while `fb commit` records the changes in the tool's reserved internal branch.

5.2.3 | Creating an SPL

When creating a new SPL from scratch, the developers just need to establish a directory where the FB Tool will be executed. Through the tool's interface (command-line prompt in the current version), users can project the SPL and add the folders and files from their first version of the SPL.

On the other hand, for existing source code, a preprocessing step involves moving the variability from a VRM, such as CC, to the FB variability format. At the moment, this process is done manually. To perform the validation presented in the next section, we have applied a method to transfer part of the variability in the Elevator SPL to the FB-Elevator SPL, which is the same system implemented using the FB approach. However, the translation method is currently beyond the scope of this research paper. As mentioned earlier, migration of existing SPL source code depends on the VRM used in the original implementation [21] and it requires ad-hoc refactoring techniques.

5.3 | Usage Illustration

To add variability to source code from scratch using the FB tool, the user must perform activities such as *analysis*, *cutting*, *editing*, and *labelling*. Once the reference artefacts are available (e.g., source code or text in some DSL, such as a file with PlantUML specifications), the user must undertake the analysis activity. This activity creates the initial block graph of the source code in both the database and Git.

Figure 9 shows the commands (`configure`, `analysis`, `add`, `commit`) a user must use to complete the analysis activity. The

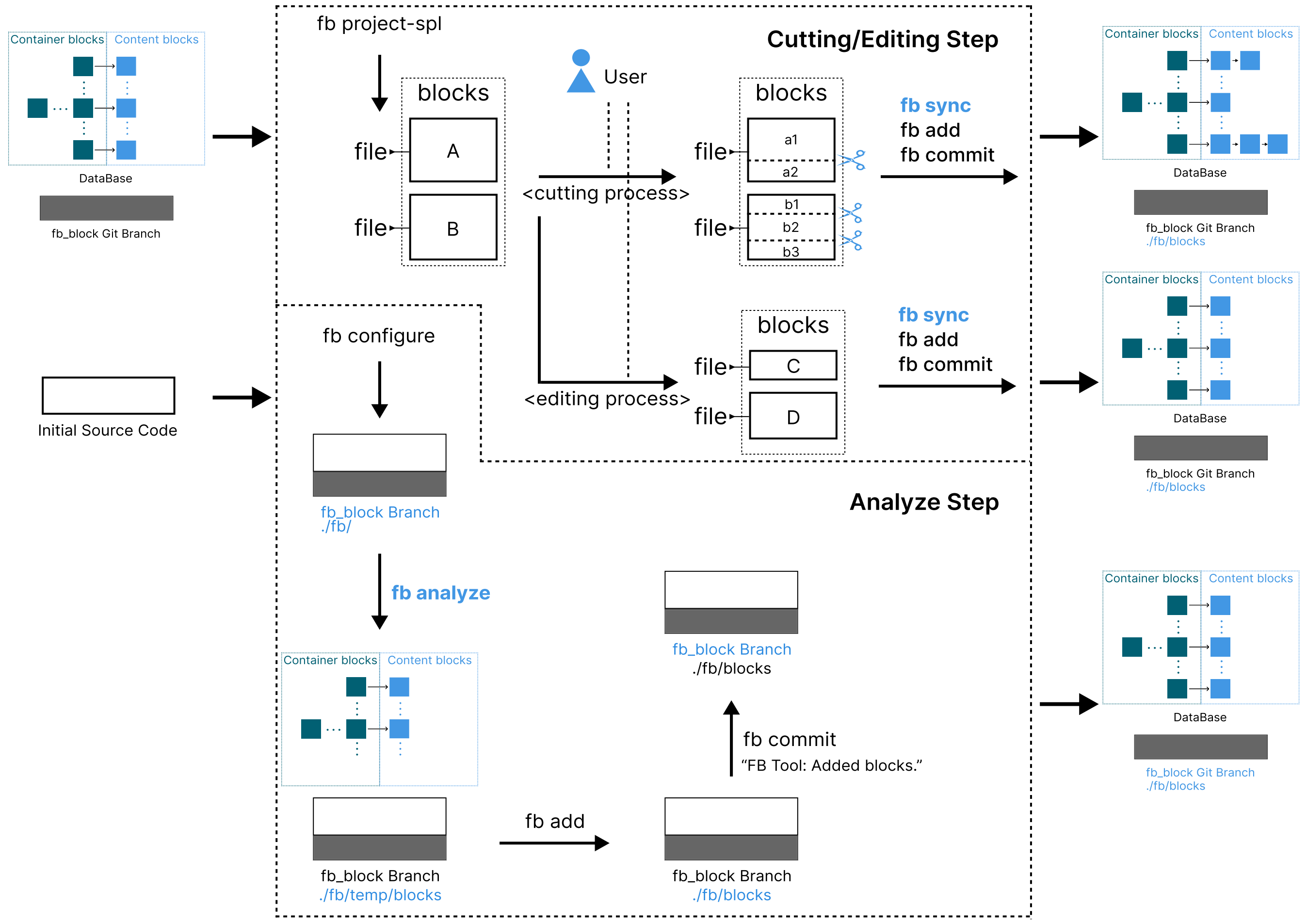


FIGURE 9 | Typical steps for developing an SPL using the FB tool.

`configure` command creates a new branch in a Git repository with a hidden folder where files representing each block will be stored. The `analysis` command will analyse the source code, creating container and content blocks according to the content in the folder that was used as input to the process. Up to this point, the content of the blocks is stored in a temporary folder; the `add` command will then move the blocks' content from the temporary folder to the folder where blocks are actually stored. Finally, the `commit` command records these changes in the branch. Once the original input artefacts have been analysed, they can be removed because their content is now stored in the FB blocks. All the artefacts of the system and its variants will then be built using the blocks from the Git branch and the relations in the database.

The activities of *cutting* and *editing*, as mentioned in Figure 9, are essential to create or update blocks managing the SPL's variability. For instance, if the tool user wants to add variability to a file, they must identify the varying parts of its source code and convert them into distinctive blocks by cutting the original block. The cutting activity internally edits and creates blocks. Sometimes, to enhance the visual presentation of blocks when projecting a product, users may need to modify the source code within these blocks through the editing activity. This may involve removing unwanted empty spaces or adjusting the formatting. Figure 9 shows the different cutting and editing results. The cutting activity modified the structure of the block graph, while the editing activity did not. The `sync`, `add`, and `commit` commands are shared between these activities. The `sync` command interprets the cutting directives and analyses the current content of the projected artefacts blocks to facilitate their creation and editing.

6 | Validation

We conducted two case studies, one using the English-Learning Books SPL and the other using the Elevator SPL [7]. Both SPLs comprise several products that share common characteristics while also exhibiting variability. The selection of the English-Learning Books SPL was determined by its suitability as a non-software case study. In contrast, the Elevator SPL was chosen for its relevance in applying and demonstrating our approach within a software context. Both SPLs were specifically developed as part of our effort to demonstrate the feasibility of the proposed approach, which constitutes the primary goal of this validation. A broader evaluation involving larger, real-world SPLs is left as future work.

The Elevator SPL is a representative SPL implementation that employs CC to manage variability. Although it includes only a limited number of features, it exhibits sufficient variability to reflect a realistic SPL scenario, and its manageable size made it feasible to reimplement using our FB approach. Furthermore, CC is, to the best of our knowledge, the only mechanism that partially supports both textual artefacts and variability in time—though only when complemented by external tools. This strengthened the rationale for selecting the Elevator SPL, as its implementation relies on CC for variability management.

To prepare the basis for the evolution scenarios in our study, we reimplemented the Elevator SPL using the FB approach, taking the original implementation, based on CC, as a reference. In addition, we developed a modified version of the Elevator SPL that also employed CC. As a result, we established two distinct branches of the same product line, each realised through a different implementation approach. These branches formed the basis for applying and comparing both approaches in the evolution scenarios presented later. We created evolution scenarios only in the Elevator SPL case, since no tool was available to support CC in the textual artefact context for the English-Learning Books SPL.

The rest of this section is structured as follows. Subsection 6.1 presents the non-software use case, while Subsection 6.2 addresses the software use case. Each subsection includes a description of its methodology (Sections 6.1.1 and 6.2.1) and provides further details on the SPLs considered in each case study (Sections 6.1.2 and 6.2.2). The software case also introduces three evolution scenarios for the Elevator SPL (Section 6.2.3). We then present a qualitative analysis of the effort required to apply the FB approach (Section 6.3), and finally conclude with a discussion of limitations (Section 6.4).

6.1 | Non-Software Use Case

6.1.1 | Methodology

To validate the applicability of the proposed approach beyond software-intensive domains, we designed a non-software case study in the context of a SPL of books. The case study focused on English-learning materials, which typically comprise four related products: the student's book, the student's workbook, the teacher's book, and the teacher's workbook. This domain was particularly appropriate for illustrating variability, since each of these products shares a set of common features—such as units, lessons, and exercises—while also introducing product-specific content and adaptations.

6.1.2 | English-Learning Books SPL

The EnglishLearningBooks (ELBooks) SPL comprises four products: the teacher's book, the student's book, the student's workbook, and the teacher's workbook. These products, corresponding to the Next Release English 1 series, were realised through separate LaTeX documents in combination with a customised LaTeX package we developed. The series could also be extended with additional products, such as a vocabulary book or a phonetics book, whose content would align with the chapters of the student's book. In our implementation, the SPL was limited to producing content up to Chapter 1, Section A in the four main books. As a potential evolution scenario, further chapters could be developed and tested, but this was not considered in the present study.

Figure 10 illustrates two complementary views of the ELBooks SPL:

i. the FM, which contains 19 features, with structure, edition, and type being the most relevant since their child features define the actual content of the book (e.g., the content of the Vocabulary feature may vary according to the chosen edition, type, and structure); and

ii. the *block graph* in the Neo4j graph database tool, which constitutes a key artefact of our approach, as it provides a complementary graph perspective on variability and commonality within the SPL at a given point in time.

At the time of this evaluation, the ELBooks SPL block graph comprised 387 relations and 224 nodes, including 146 blocks, 20 features (one more than in the FM, as generated by the tool), 28 fragments, 20 indices, 5 products, and 5 containers. The source code and database dump of the ELBooks SPL are available in our public GitHub organisation.[9]

6.2 | Software Use Case

6.2.1 | Methodology

For this case study, we used the Elevator SPL from FeatureIDE as the subject system. The original Elevator SPL was implemented using the CC approach. To adapt it for our validation, we created a new implementation, called FB-Elevator, by modifying its source code so that variability management is realised with the FB approach. Although FB-Elevator does not replicate the entire original system, it preserves all the variability aspects relevant to this validation.

Our evaluation comprises both qualitative and quantitative comparisons between the CC and FB approaches. To ensure a consistent evolution process in both approaches, we relied on the sequence of activities defined in the `Simple_Seq_EA_for_SPL` algorithm (Algorithm 1), which in turn invokes `Simple_Seq_EA_for_SP` (Algorithm 2) for evolving individual products. Using this procedure, the only difference between evolving the Elevator SPL with the CC VRM and with the FB VRM lies in how the `evolveSPL()` method is executed.

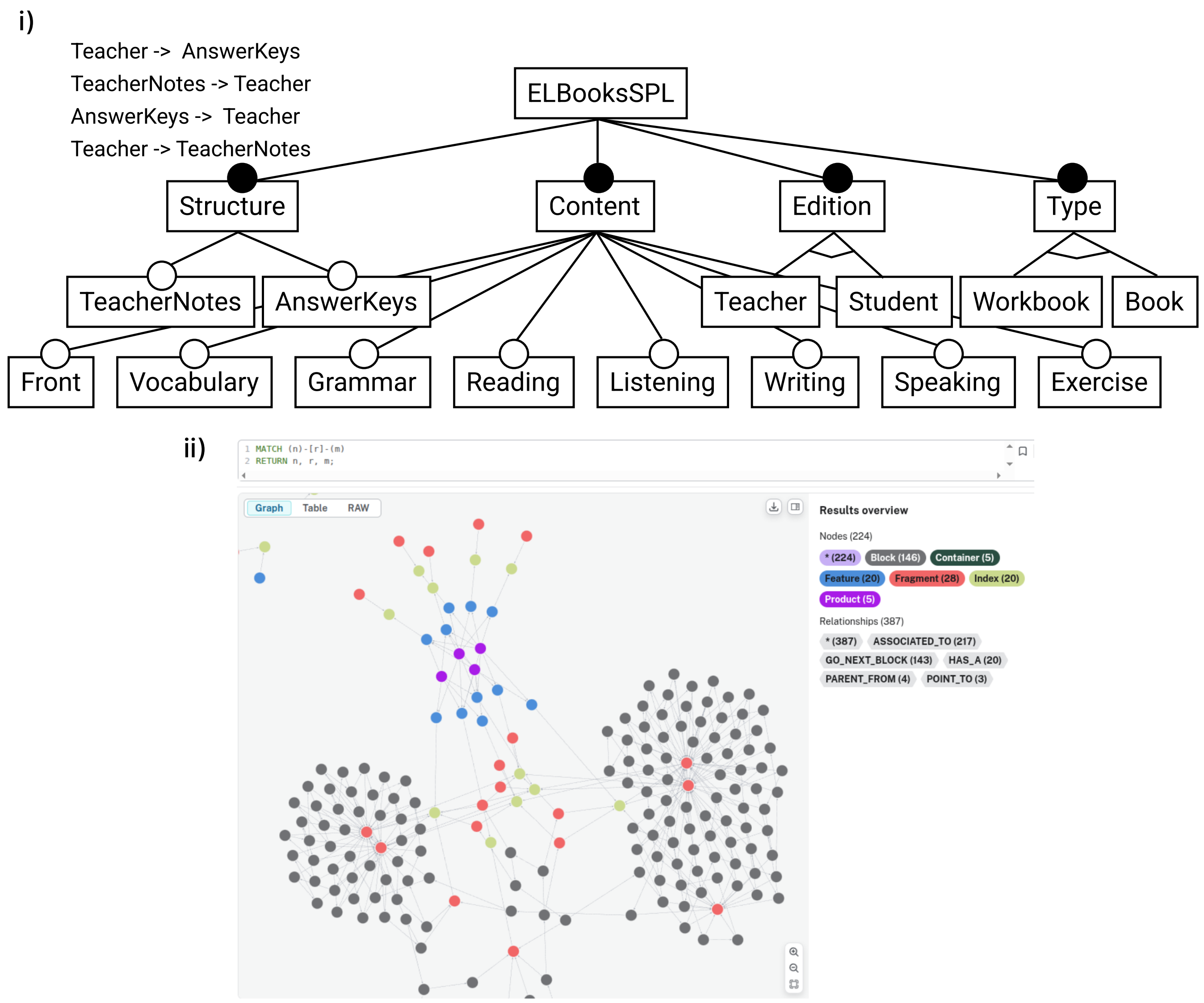
For the quantitative evaluation, we applied the metrics defined in Table 2, reporting results for each evolution scenario in separate tables. For the qualitative evaluation, we focused on the user effort required to apply each evolution scenario with the `evolveSPL()` method. The effort was assessed on a three-level scale: *manageable* (less than one hour of work), *moderate* (a few hours of work), and *challenging* (eight hours or more).

Finally, to assess the approaches in practice, we defined three evolution scenarios for the Elevator SPL. These scenarios represent realistic types of changes in SPLs:

i. adding new features,

ii. modifying the implementation of existing features, and

iii. removing features.



```
MATCH (n)-[r]-(m)
RETURN n, r, m;
```

Results overview

Nodes (224)

* (224) Block (146) Container (5) Feature (20) Fragment (28) Index (20) Product (5)

Relationships (387)

* (387) ASSOCIATED_TO (217) GO_NEXT_BLOCK (143) HAS_A (20) PARENT_FROM (4) POINT_TO (3)

FIGURE 10 | (i) FM of the English-Learning Books SPL; (ii) Block Graph representation of the English-Learning Books SPL.

Both the quantitative metrics and the qualitative effort scale were applied systematically across these three scenarios, following the evolution procedure as described earlier. The detailed description and evaluation of each case are presented in Section 6.2.3.

In our evaluation, the computation of RLOCM differs depending on the approach. For the FB approach, we project the entire SPL (`project-spl` command) from the initial state into the block format and then compare it with the projection of the target state in the same format. For the CC approach, we directly compare the source code of the initial version against the target version. In both cases, RLOCM excludes non-semantic modifications such as whitespace edits or formatting changes, as well as code edits unrelated to the defined evolution scenarios, including automatically generated files or IDE-related artefacts.

Finally, the history of changes (evolution scenarios) in the SPL is recorded in separate repositories, one for the CC VRM and one for the FB VRM.

6.2.2 | Elevator SPL

The Elevator SPL is an SPL of products that simulates elevator control systems. It features a configurable control panel that manages the elevator's movement between floors, mirroring how a real elevator operates. By selecting a different set of features in the product configuration, the SPL generates an elevator control in Java with the specifications and characteristics defined by the selected configuration. The original Elevator SPL exists in various versions. The one used in this evaluation is the Elevator-Antenna version 1.4 [7]. This version has a FM with 21 features and 3 restrictions, as shown in Figure 11. On the other hand, the first version of the FB-Elevator SPL has more features than the Elevator SPL since the conversion method introduced additional features when translating the Elevator variability. These 31 additional features, termed 'unassigned features,' need to be allocated within the new FM of the new SPL; this allocation is typically undertaken during the tuning phase of projecting SPL products.

The source code of the FB-Elevator SPL [23] and the Elevator SPL version 1.4 [24] are shown in our public GitHub organisation. A Docker version of the database used in the FB-Elevator SPL is shared in Docker Hub.[10]

6.2.3 | Evolution Scenarios on the Elevator SPL

6.2.3.1 | Scenario 1: Adding More Features to the SPL.

Description: In the first evolution scenario, four new features are added to the SPL: Stop, UI, Enterprise, and Common (Figure 12). The Stop feature introduces a button to halt the elevator's up-and-down movement and was incorporated as an optional child of the behaviour feature. In contrast, the UI feature separates the presentation mode of the resulting application into two alternatives: Enterprise UI and Common UI. In addition, this UI feature was added as an optional child of the ElevatorSPL feature.

As indicated in the evolution procedure described in the methodology, after defining the changes in the SPL, the evolution algorithm requires selecting an SPL product and applying the specified variability. Finally, once the evolution is incorporated into a product, the modifications must also be manually incorporated into the SPL itself—the complexity of this step depends significantly on the type of VRM to be employed.

Representative actions in the evolution process: Figure 13 shows a portion of the actions applied during the evolution of the SPL using the CC approach (i and ii) and the FB approach (iii, iv, and v). (i) represents the initial source code from the original Elevator SPL, while (ii) represents the changes added by the user after analysing the source code. (iii) represents the initial source code from the FB-Elevator SPL. (iv) represents the changes added by the user after analysing the source code. Finally, (v) represents some of the blocks in the database (DB) that belong to that specific file after applying the commands detailed in the previous chapters using the changes in (iv). Although we applied (iv), we could also apply a different way to evolve the source code using the cut tag in block 101 at the end of the block content, followed by the source code we want to add, and then process these changes with the same commands applied above.

Evolution and Results: In the case of the FB approach, we begin with the FB-Elevator SPL. This initial version maintains its own database and a dedicated Git branch for blocks. As the algorithm for evolution specifies to work first on a specific product, we worked on the Enterprise version, adding the planned new features to this product. After verifying the evolution, we committed the changes and used the Git diff tool to track this variation. Then,

TABLE 2 | Quantitative Metrics.

Metric	Acronym	Description
Number of modified files	NMF	Counts the source files affected in the transition from the initial SPL state to the target state specified by the scenario.
Number of annotations or markers	NAM	Number of variability-related annotations/markers (e.g., `#ifdef` directives in CC or block markers such as `b» [cut]` in FB) that are added, modified, or removed in each scenario.
Number of Git commits	NGC	Number of commits used to evolve the SPL from its initial state to the target state defined by the scenario.
Relevant Lines of Code Modified	RLOCM	Number of source code lines added, removed, or changed when evolving the SPL from its initial state to the target state defined by the scenario, excluding non-semantic modifications such as whitespace changes, formatting adjustments, or edits unrelated to the focus of the evaluation.

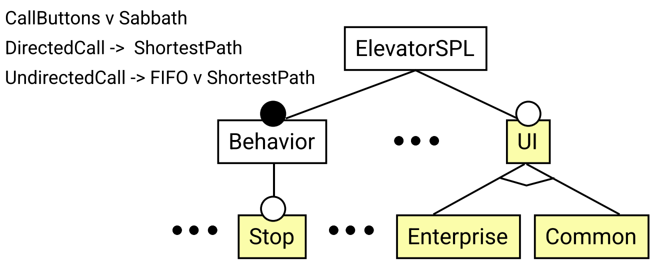

CallButtons v Sabbath
DirectedCall -> ShortestPath
UndirectedCall -> FIFO v ShortestPath
ElevatorSPL
Behavior
UI
Stop
Enterprise
Common

FIGURE 12 | Features added to the SPL in the first evolution scenario.

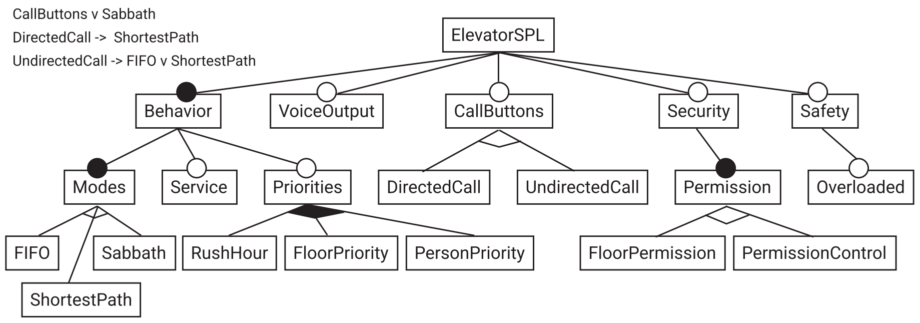
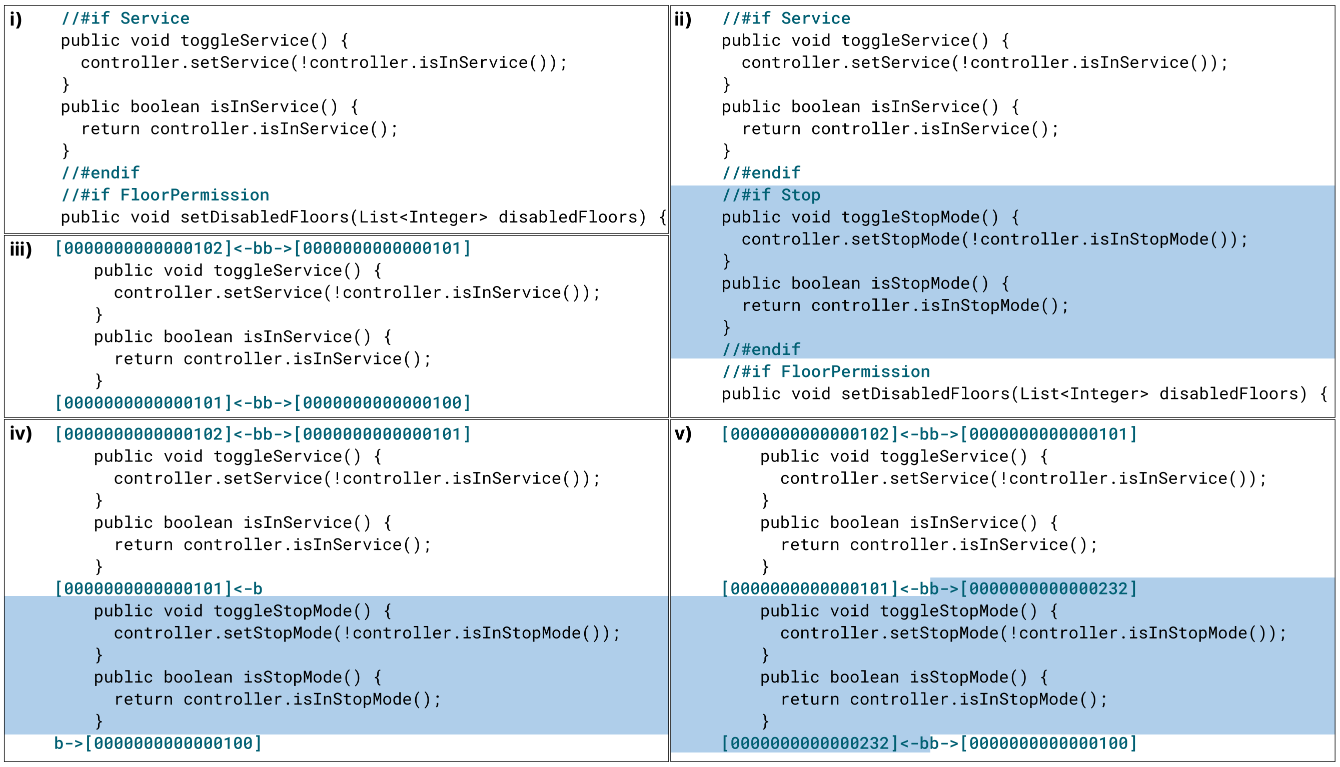


FIGURE 11 | FM of the Elevator SPL [22].



i)
```
//#if Service
public void toggleService() {
  controller.setService(!controller.isInService());
}
public boolean isInService() {
  return controller.isInService();
}
//#endif
//#if FloorPermission
public void setDisabledFloors(List<Integer> disabledFloors) {
```

ii)
```
//#if Service
public void toggleService() {
  controller.setService(!controller.isInService());
}
public boolean isInService() {
  return controller.isInService();
}
//#endif
//#if Stop
public void toggleStopMode() {
  controller.setStopMode(!controller.isInStopMode());
}
public boolean isStopMode() {
  return controller.isInStopMode();
}
//#endif
//#if FloorPermission
public void setDisabledFloors(List<Integer> disabledFloors) {
```

iii)
```
[0000000000000102]<-bb->[0000000000000101]
    public void toggleService() {
      controller.setService(!controller.isInService());
    }
    public boolean isInService() {
      return controller.isInService();
    }
[0000000000000101]<-bb->[0000000000000100]
```

iv)
```
[0000000000000102]<-bb->[0000000000000101]
    public void toggleService() {
      controller.setService(!controller.isInService());
    }
    public boolean isInService() {
      return controller.isInService();
    }
[0000000000000101]<-b
    public void toggleStopMode() {
      controller.setStopMode(!controller.isInStopMode());
    }
    public boolean isStopMode() {
      return controller.isInStopMode();
    }
b->[0000000000000100]
```

v)
```
[0000000000000102]<-bb->[0000000000000101]
    public void toggleService() {
      controller.setService(!controller.isInService());
    }
    public boolean isInService() {
      return controller.isInService();
    }
[0000000000000101]<-bb->[0000000000000232]
    public void toggleStopMode() {
      controller.setStopMode(!controller.isInStopMode());
    }
    public boolean isStopMode() {
      return controller.isInStopMode();
    }
[0000000000000232]<-bb->[0000000000000100]
```

FIGURE 13 | Representative evolutionary steps in Scenario 1.

following this variation between versions, we created the new blocks in the SPL using the cutting and editing activities mentioned in the above sections. The process to add the variability in the SPL took less than one hour, while adding the new features to a single product took more than one hour, but combined, it still took less than eight hours.

In the case of the CC approach, we obtained a similar result when generating the "Enterprise" product. Additionally, the effort to evolve the SPL (via the $evolveSPL()$ method) using the CC approach took a similar amount of time, not more than eight hours. We have to highlight that we have chosen the enterprise configuration as the default product configuration in FeatureIDE and used the original Elevator SPL as the initial version.

The qualitative analysis revealed that, in this scenario, both approaches required a similar amount of time—a few hours of work (moderate).

The quantitative analysis, presented in Table 3, shows that both approaches yielded broadly comparable results. NMF, NAM, and NGC were essentially the same in both cases. For RLOCM, the values were of the same order of magnitude, but slightly lower for the FB approach (108 lines) compared to the CC approach (149 lines), suggesting that the FB approach is somewhat more compact.

6.2.3.2 | Scenario 2: Changing the Source Code of Features.

Description: In the second evolution scenario, we consider a change in the visual presentation of the Elevator, specifically modifying the arrangement of buttons from left-to-right to right-to-left. In addition, the functionality for floor selection is altered: instead of disabling certain floors, it now enables specific floors.

TABLE 3 | Quantitative metrics – scenario 1 (adding features).

Metric	CC approach	FB approach
NMF	5	5
NAM	51	52
NGC	1	1
RLOCM	149	108

Representative actions in the evolution process: Figure 14 shows a portion of the actions applied during the evolution of the SPL using the CC approach (i and ii) and the FB approach (iii and iv). (i) represents the initial source code from the original Elevator SPL, while (ii) represents the updates added by the user after analysing the source code. (iii) represents the initial source code from the FB-Elevator SPL. (iv) represents the updates added by the user after analysing the source code. Updates in blocks are very similar to updates using the conditions of CC, except that, in the current prototype, the FB approach requires explicit commands, whereas, in CC, the FeatureIDE does all the inner work.

Evolution and results: In both the FB and CC VRM cases, we begin this evolution scenario from the SPL resulting from the changes in the first scenario. We first generated the "Enterprise" product and then applied the planned modifications to its features. As in Scenario 1, we committed the changes to a repository and captured the variation using the Git diff tool. Based on this variation, we edited some blocks of the *block graph* using the editing

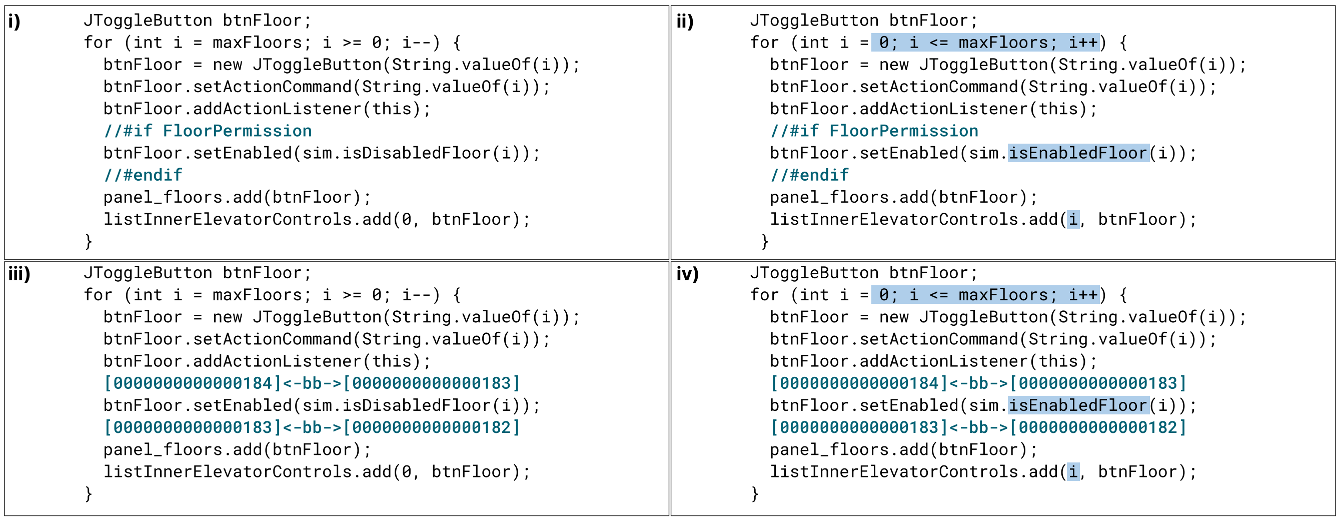

i)
```
JToggleButton btnFloor;
for (int i = maxFloors; i >= 0; i--) {
  btnFloor = new JToggleButton(String.valueOf(i));
  btnFloor.setActionCommand(String.valueOf(i));
  btnFloor.addActionListener(this);
  //#if FloorPermission
  btnFloor.setEnabled(sim.isDisabledFloor(i));
  //#endif
  panel_floors.add(btnFloor);
  listInnerElevatorControls.add(0, btnFloor);
}
```
ii)
```
JToggleButton btnFloor;
for (int i = 0; i <= maxFloors; i++) {
  btnFloor = new JToggleButton(String.valueOf(i));
  btnFloor.setActionCommand(String.valueOf(i));
  btnFloor.addActionListener(this);
  //#if FloorPermission
  btnFloor.setEnabled(sim.isEnabledFloor(i));
  //#endif
  panel_floors.add(btnFloor);
  listInnerElevatorControls.add(i, btnFloor);
 }
```
iii)
```
JToggleButton btnFloor;
for (int i = maxFloors; i >= 0; i--) {
  btnFloor = new JToggleButton(String.valueOf(i));
  btnFloor.setActionCommand(String.valueOf(i));
  btnFloor.addActionListener(this);
  [0000000000000184]<-bb->[0000000000000183]
  btnFloor.setEnabled(sim.isDisabledFloor(i));
  [0000000000000183]<-bb->[0000000000000182]
  panel_floors.add(btnFloor);
  listInnerElevatorControls.add(0, btnFloor);
}
```
iv)
```
JToggleButton btnFloor;
for (int i = 0; i <= maxFloors; i++) {
  btnFloor = new JToggleButton(String.valueOf(i));
  btnFloor.setActionCommand(String.valueOf(i));
  btnFloor.addActionListener(this);
  [0000000000000184]<-bb->[0000000000000183]
  btnFloor.setEnabled(sim.isEnabledFloor(i));
  [0000000000000183]<-bb->[0000000000000182]
  panel_floors.add(btnFloor);
  listInnerElevatorControls.add(i, btnFloor);
}
```

FIGURE 14 | Representative evolutionary steps in Scenario 2.

activity described above. Editing the variability in the SPL took less than one hour, while applying the changes to a single product took more than one hour but less than two.

Overall, the results were consistent across both approaches. In the qualitative analysis, the effort to evolve the SPL required between one and three hours in total, which we classified as *moderate* effort. The quantitative analysis for Scenario 2, presented in Table 4, shows that both approaches modified the same number of files. Neither approach required annotations/markers, and the evolution was captured with a single Git commit in each case. The RLOCM values differed, with 41 lines for the CC approach and 21 lines for the FB approach, indicating that the FB approach required fewer relevant code modifications in this scenario.

6.2.3.3 | Scenario 3: Removing a Feature From the SPL.

Description: In the third evolution scenario, we consider the removal of the Service feature (Figure 15), which is positioned as a child of the behaviour feature. The Service feature consists of adding a button to the interface, allowing users to signal a move to floor number zero, which is a special floor used to perform maintenance on the elevator.

Representative actions in the evolution process: Figure 16 shows a portion of the actions applied during the evolution of the SPL using the CC approach (i and ii) and the FB approach (iii). (i) represents the initial source code from the original Elevator SPL, while (ii) represents the removals added by the user after analysing the source code. (iii) represents the initial source code from the FB-Elevator SPL. Since fragments and blocks manage the SPL source code, the blocks can be removed from the product generation by simply labelling them with another fragment not related to the product (most of the time, the default fragment).

Evolution and results: As in the previous scenarios, we begin this evolution scenario from the SPL resulting from the changes in the second scenario, both for the FB and CC VRM cases. We first generated the "Enterprise" product and then incorporated the planned removal of the Service feature. As in the earlier

TABLE 4 | Quantitative metrics – scenario 2 (editing features).

Metric	CC approach	FB approach
NMF	5	5
NAM	0	0
NGC	1	1
RLOCM	41	21

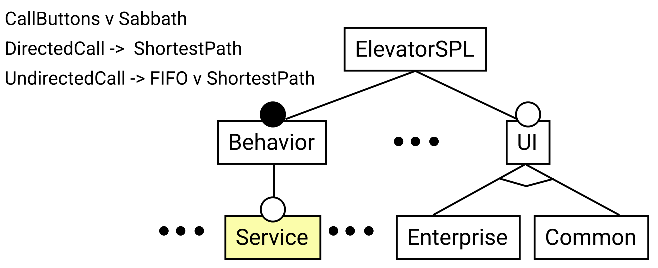
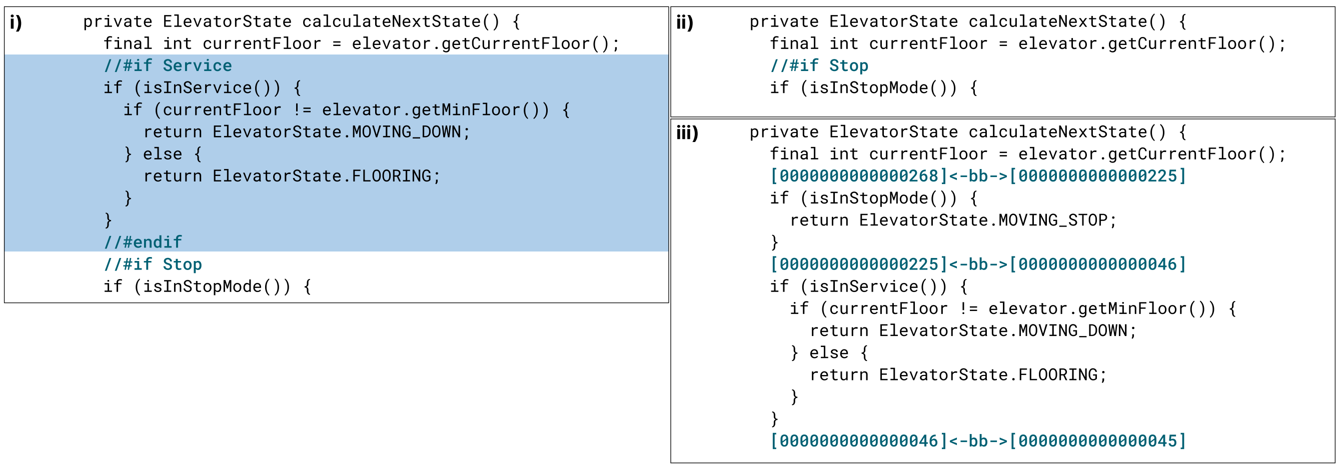

CallButtons v Sabbath
DirectedCall -> ShortestPath
UndirectedCall -> FIFO v ShortestPath
ElevatorSPL
Behavior
UI
Service
Enterprise
Common

FIGURE 15 | Chosen feature to be removed in the third evolution scenario.

scenarios, we committed the changes to a repository and captured the variation using the Git diff tool. Based on this variation, we edited some blocks using the editing activity described above in order to reflect the removal of the feature.

The results were consistent across both approaches. In the qualitative analysis, the effort to evolve the SPL required between one and two hours in each case, which we classified as *moderate* effort.

The quantitative analysis for Scenario 3, presented in Table 5, shows that both approaches modified the same number of files and required only a single Git commit. However, the CC approach involved 42 annotations/markers, whereas the FB approach required none. The RLOCM values also differed considerably, with 112 lines for the CC approach and 48 lines for the FB approach, indicating that the FB approach required fewer code modifications overall.



i)
```
    private ElevatorState calculateNextState() {
      final int currentFloor = elevator.getCurrentFloor();
      //#if Service
      if (isInService()) {
        if (currentFloor != elevator.getMinFloor()) {
          return ElevatorState.MOVING_DOWN;
        } else {
          return ElevatorState.FLOORING;
        }
      }
      //#endif
      //#if Stop
      if (isInStopMode()) {
```

ii)
```
    private ElevatorState calculateNextState() {
      final int currentFloor = elevator.getCurrentFloor();
      //#if Stop
      if (isInStopMode()) {
```

iii)
```
    private ElevatorState calculateNextState() {
      final int currentFloor = elevator.getCurrentFloor();
      [0000000000000268]<-bb->[0000000000000225]
      if (isInStopMode()) {
        return ElevatorState.MOVING_STOP;
      }
      [0000000000000225]<-bb->[0000000000000046]
      if (isInService()) {
        if (currentFloor != elevator.getMinFloor()) {
          return ElevatorState.MOVING_DOWN;
        } else {
          return ElevatorState.FLOORING;
        }
      }
      [0000000000000046]<-bb->[0000000000000045]
```

FIGURE 16 | Some evolutionary steps in Scenario 3.

TABLE 5 | Quantitative Metrics – Scenario 3 (Removing Features).

Metric	**CC approach**	**FB approach**
Number of modified files	4	4
Number of annotations/markers	42	0
Number of Git commits	1	1
RLOCM	112	48

6.3 | Effort Analysis and Observations

Building on the experience gained from both case studies (ELBooks and Elevator) and the evolution scenarios applied to the Elevator SPL, we present a qualitative comparison of the FB and CC approaches. The analysis considers four aspects that directly influence the practical maintainability and usability of an SPL during evolution: *Change Localizability*, *Feature Traceability*, *Developer Comprehension Effort*, and *Toolchain Integration*. Each aspect is evaluated using three qualitative levels to capture finer distinctions: *High/Medium/Low* for change localizability, *Clear/Partial/Unclear* for feature traceability, *Easy/Moderate/Hard* for developer comprehension effort, and *Flexible/Semi-flexible/Restricted* for toolchain integration.

Change Localizability
This aspect refers to how easily a developer can identify and isolate the exact location of changes during evolution. FB leverages Git to track modifications at the block level, which improves visibility compared to CC, but still requires developers to interpret the semantics of the modified blocks to fully understand the impact of the change. For this reason, we classify it as *Medium*. In contrast, CC requires scanning entire source files and manually interpreting conditional directives, which obscures differences and results in a *Low* classification.

Feature Traceability
This aspect measures how clearly code sections are associated with specific features. In FB, each block is linked to a fragment that corresponds to a feature in the model. However, making these associations visible in practice requires queries and projections in the block graph, which we classify as *Partial* traceability. In CC, by contrast, variability is expressed through non-standardised `#ifdef` annotations that may overlap or be nested, making it difficult to map code to specific features. For this reason, CC is categorised as *Unclear*. The highest level, *Clear*, would apply to approaches where feature associations are always directly visible without the need for additional queries or conventions.

Developer Comprehension Effort
This aspect reflects the cognitive effort required for a developer to read, understand, and modify the code when working on a particular feature. FB provides a projected view that shows only the relevant blocks for the selected configuration, reducing distractions and simplifying comprehension compared to CC. Developers can also rely on Neo4j queries to navigate the block graph and better understand the variability. Nevertheless, some analysis effort is still required, and for this reason, we classify FB as *Moderate*. By contrast, CC requires developers to mentally filter through nested conditions and multiple configurations, which significantly increases cognitive load and makes comprehension *Hard*.

Toolchain Integration
This aspect evaluates how well each approach can be integrated into different development environments and external tools. The FB approach requires its own tool to operate, but since it works on text-based artefacts, it can be combined with common development environments and toolchains. For this reason, we classify FB as *Semi-flexible*. By contrast, the CC approach relies on language-specific preprocessors, whose availability and reliability vary considerably across ecosystems. In some modern software projects (e.g., React projects), suitable preprocessors may not even exist, which makes integration with external IDEs or toolchains particularly challenging. Therefore, CC is classified as *Restricted*. The highest level, *Flexible*, would correspond to approaches that integrate smoothly with diverse tools and environments without relying on a dedicated infrastructure (Table 6).

These observations, although qualitative, highlight essential differences between the two approaches when applied to real SPL evolution tasks. They emerge from our experience with both case

TABLE 6 | Qualitative comparison between FB and CC.

Aspect	FB approach	CC approach
Change localizability	Medium	Low
Feature traceability	Partial	Unclear
Developer comprehension effort	Moderate	Hard
Toolchain integration	Semi-flexible	Restricted

studies and the comparative evaluation of evolution scenarios. Overall, they illustrate practical concerns related to maintainability, developer effort, and integration with development ecosystems—issues that are not fully captured by quantitative metrics alone.

6.4 | Limitations and Scope of Evaluation

Our evaluation was based on two case studies: the ELBooks SPL (non-software) and the Elevator SPL (software). The ELBooks SPL was designed to illustrate the applicability of our approach in a non-software domain, but its implementation was limited to producing content up to "Chapter 1, Section A" of the book. The Elevator SPL, while relatively small, is a well-established benchmark in SPL research and was used to evaluate three representative evolution scenarios: feature addition, feature modification, and feature removal. Both case studies, therefore, provide complementary perspectives but remain constrained in scope.

A broader empirical evaluation across domains and larger SPLs would be valuable but lies outside the scope of this paper. A further limitation is the lack of publicly available, product-line-style SPLs; as a result, researchers often rely either on small benchmarks (such as Elevator) or on purpose-built SPLs (such as ELBooks). Nevertheless, we believe that our scenario-based validation—combining both qualitative and quantitative analyses of effort—offers an appropriate and meaningful first step toward understanding the practical impact of the FB approach. Additionally, all evaluation artefacts are publicly available to facilitate reproducibility and further experimentation by the research community.

These limitations also point to opportunities for future research.

7 | Related Work

7.1 | Tools and Approaches for Variability Management

Different tools have been developed for managing some aspect of variability in SSs. In this subsection, we discuss the most popular approaches in the literature that also have at least a prototype implementation. Unfortunately, most of the tools listed in this section are either proprietary, not publicly available, or not actively maintained.

Variation control systems (VarCSs) allow working on one or multiple variants by providing views (or projections) that filter irrelevant details of configurable artefacts to facilitate their comprehension and lower the cognitive complexity when editing them [25]. However, most VarCSs depend on or have a particular restrictive style of use (e.g., outdated or not maintained tools), so they become unattractive to most developers [26]. The VarCSs workflow defines concepts such as internal representation, external representation, externalisation operation, and internalisation operation. The "internal representation" stores the current source code, usually not visible to the users, while the "external representation" is the part that the user can interact with and modify. In VarCSs, to generate an "external representation," an "externalisation operation" with an "externalisation expression" is used. On the other hand, VarCSs uses the "internalisation operation" with an "internalisation expression" to generate the "internal representation." Ideally, when updating the external representation in a VarCS, the internal representation has to be updated automatically.

Leviathan [27] employs a filesystem-centric strategy to surpass the conventional, toolchain-dependent approaches. By forging an abstraction layer over the filesystem, it manages to orchestrate the SPL variability without blocking the user tools. As a result, developers can handle multiple variants simultaneously without directly involving specific tools that hinder the development of configurable SSs. Leviathan, nevertheless, does not support changes in conditional blocks when working on the mounted view, although it does support changes in the content of those blocks. This limitation is due to the heuristic algorithm used to merge changes in the code. Leviathan mentioned four use cases to consider when working with configurable systems: the first centres on analysing the performance of a specific variant; the second measures the effort required to comprehend the source code with variability; the third focuses on refactoring tools related to a variant; the fourth concentrates on the maintenance aspects of the configurable SS. As indicated in the study, the latter two scenarios—refactoring and maintenance—are beyond the scope of Leviathan's support. Consequently, the research does not account for changes over time, an aspect of variability in time.

DeltaEcore [28] is a framework tailored to generating Delta Languages and Delta Dialects. Delta languages concretely delineate variabilities, detailing the specific transformations applicable to a base system (e.g., an instance of a model). In contrast, a Delta dialect generically outlines the permissible variabilities or changes. Approaches that apply the Delta Modelling as mechanisms to manage the SPL variability can use this framework to facilitate the creation of Delta dialects and then, by using these dialects, facilitate the creation of the Delta languages. The general idea of Delta Modelling is to transform one valid variant of the family into another variant by adding, modifying, or removing elements of the first variant. Delta Modelling emphasises the importance of a systematic application of changes, ensuring that each transformation is executed in a defined sequence to maintain the integrity of the software variant. The authors of DeltaEcore showed four artefacts related to safety-critical software that were created using this framework. However, the discussion on the evolutionary implications of applying the delta modelling approach is not addressed.

SuperMod [29] is an Eclipse-integrated tool for versioning the "variability in space" of a model-driven SPL. While primarily designed for model contexts, it can also function as a straightforward VCS in other scenarios such as Web Development. Within a local environment, SuperMod offers three core operations. The first is the checkout command, which, upon a specified configuration called Choice, retrieves a workspace from the repository for the SPL. The second is the modify command, used to alter the workspace of the SPL. The third is the commit command, which saves the changes under a specified configuration called Ambition. In a distributed environment, push and pull commands are also available. These additional distributed operations, pull and push, resemble Git commands. At the moment of pushing the changes, the tool merges prior changes. The document does not address resolving potential conflicts arising from user changes during merging. Although they say it is easily integrated with user tools, SuperMod is based on Eclipse, which has lost much of its popularity as an IDE today, according to the 2025 Stack Overflow Developer Survey.[11] The integration of SuperMod with other environments is not straightforward, and developers may be reluctant to use such an approach if it involves adopting an IDE they perceive as outdated. Besides, this tool is only applicable when developers decide to use the model-driven approach to deal with the SPL variability.

DarwinSPL [30] is a collection of tools designed to simultaneously manage the variability in space and time. The authors divide these variabilities into spatial, contextual, and temporal variability. DarwinSPL mainly uses the temporal FMs, HyVarRec (a reconfiguration engine for SPLs), and DeltaEcore to manage the variabilities. The feasibility of DarwinSPL is demonstrated through its application in managing electronic control units. These units capture geolocation and set up emergency calls in case of accidents. The integration of diverse ideas and tools from various authors in DarwinSPL adds complexity to new development projects that adopt this approach. Furthermore, since it depends on DeltaEcore, it also depends on the Delta Modelling approach.

ECCO [31, 32] began as a tool mainly focused on variability in space. ECCO concepts such as extraction, composition, and completion were tested in five case studies. Subsequently, ECCO extended its capabilities to include aspects of variability in time, introducing checkout and commit operations based on specific configurations. The commit operation, which computes traces from features to their implementation artefacts, compares the repository's content with the configuration and its implementation. In this process, the repository does not store each variant separately; instead, it stores these traces. The checkout operation selects the necessary implementation artefacts from the repository required for a given configuration and recomposes them correctly. This tool faces a challenge similar to many others reliant on the Eclipse platform, which has diminished in popularity compared to ten years ago.

SiPL [33] is a sophisticated Delta-based modelling framework designed for the implementation of SPLs using the delta modelling approach. Key functionalities of the framework include: (i) facilitating the creation and visualisation of Delta Modules; (ii) analysing Delta-Module Relations to identify conflicts, dependencies, and potential duplicates; (iii) restructuring the Delta-Module Set with operations such as intersect, concat, minus, purge, and merge; and (iv) enabling product generation and (v) the debugging of Edit Scripts.

Although SiPL streamlines the development and maintenance of model-focused SPLs, it lacks an explicit explanation of how it handles variability over time. Moreover, its application is limited as it excludes non-model-based projects.

P-Edit [34] is an old tool that implemented two mechanisms to manage the variability of SPLs: Internal Boolean expressions (conditional assembly or compilation) and Deltas. Deltas are instructions on modifying a previous version to produce the next version. A Delta strategy typically has a base file and a sequence of delta files. P-Edit combines these approaches and implements them simultaneously. However, the work [34] does not present any case study of the tool.

Given an SPL implemented using the CC VRM, the projection-based variation control system (PBVCS) prototype [35] allows (i) the generation of a simplified view of it, (ii) the edition of the code using this view, and (iii) the generation of the ambition which specifies how the developer's changes should be integrated into the repository.

The PBVCS prototype underwent testing with partial Git patches in the Marlin SPL [36], comparing the outcomes of changes made using the prototype versus the original method. Additionally, the study classified the types of changes made by Marlin's developers during the SPL development. The prototype relies solely on CC and lacks a mention of how it manages variability over time.

The recent work by Ananieva et al. [37] proposes a common taxonomy for the operations supported by tools for managing variability. In particular, the work focuses on providing an overview of the differences and similarities among variability management tools, most of which have been mentioned earlier in this section. The analysis reported in [37] shows that none of the existing tools supports all the operations needed to completely address variability both in the space and in the time perspectives. Our FB Tool supports the concepts of externalising the product and internalising changes; however, it is not a strict implementation of these concepts.

7.2 | Comparison With FB Approach

To complement the previous review of tools and approaches for variability management, Table 7 presents a comparative analysis between our proposed FB approach and representative existing solutions. The comparison is structured along six criteria that are particularly relevant for assessing the practical applicability and flexibility of variability management tools:

- *Variability in Space*: Support for managing the set of possible product configurations that can be derived from an SPL at a given point in time.
- *Variability in Time*: Support for capturing and managing how the SPL and its products evolve across versions.

TABLE 7 | Comparison of FB with other variability management approaches and tools.

	Approach/ tool	Variability in space	Variability in time	Change tracking	Feature editing	Variant editing	Textual artefacts
Specific	Leviathan	✓	✗	✗	Partial	✓	✗
	DeltaEcore	✓	✗	✗	✓(rules)	✗	✗
	SuperMod	✓	✗	✗	✓	✓	✗
	DarwinSPL	✓	✓	✗	✓(via DeltaEcore)	✗	✗
	ECCO	✓	Partial	✗	✓	✓	✗
	SiPL	✓	✗	✗	✓(deltas)	✗	✗
	PBVCS	✓	✗	✗	✓(views)	✓	✗
	FB (ours)	✓	✓	Partial (via Git)	✓(blocks)	✓	✓
General	Conditional compilation	✓	Partial (via Git)	Partial (via Git)	✓	✗	Partial
	Feature programming	✓	✗	✗	✓	✗	✗
	Java Reflection	✓	Partial (via Git)	Partial (via Git)	✓	✗	✗
	Aspect-Oriented Programming	✓	✗	✗	✓	✗	✗

- *Change Tracking*: Mechanisms for recording and tracking changes (e.g., commits, deltas, version histories).
- *Feature Editing*: Mechanisms that allow modifying features so that changes are reflected in the SPL as a whole.
- *Variant Editing*: Ability to edit or refine individual product variants and support the reintegration of these changes into the SPL.
- *Textual Artefacts*: Applicability to textual artefacts such as source code, configuration files, or documentation.

The table uses the following symbols and labels: ✓= supported; ×= not supported; *Partial* = support with significant limitations or requiring external tools (e.g., Git).

As shown in Table 7, the FB approach supports both variability in space and variability in time, a combination that only a few existing tools address simultaneously. In addition, FB operates directly on textual artefacts and integrates naturally into Git-based development workflows, without depending on tools that are no longer maintained. This makes it both accessible and practical for adoption across a wide range of development environments.

8 | Conclusion

In this paper, we present a new pragmatic approach to managing SPLs, intended as an alternative to the widely used CC approach. A key characteristic of the approach is that it does not rely on any specific IDE, making it broadly applicable across different development contexts.

We validated the approach through two case studies: a non-software case using the ELBooks SPL and a software case using the Elevator SPL. In the latter case, we applied our approach to an adapted version of the SPL and simulated three evolution scenarios. The results suggest that our approach requires an effort comparable to, but not greater than, that of the CC approach. We also presented a supporting tool, implemented in Java, that enables developers to adopt the approach in practice.

In general, this work represents a first step towards more advanced variability management techniques that exploit the concepts of fragments and blocks. Future work will focus on exploring richer graph-based capabilities for organising variability at the fragment and block level and on conducting larger-scale empirical evaluations. To facilitate reproducibility, all artefacts used in our validation are publicly available in Git repositories, enabling other researchers to replicate and extend our experiments.

Author Contributions

All authors contributed to the conception of the study, analysis of the results, and writing of the manuscript.

Acknowledgments

The Article Processing Charge for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614).

Funding

This work was supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Grant Number: 88887.466672/2019-00.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data supporting the findings of this study are openly available in the FBTechSolutions repository at: https://github.com/FBTechSolutions.

Endnotes

[1]https://react.dev/.

[2]https://www.planttext.com/.

[3]http://sonatype.github.io/munge-maven-plugin/.

[4]http://antenna.sourceforge.net/.

[5]http://www.cs.utexas.edu/users/schwartz/ATS.html.

[6]http://www.fosd.de/fh.

[7]http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/.

[8]https://www.atlassian.com/git/tutorials/comparing-workflows.

[9]https://github.com/FBTechSolutions.

[10]https://hub.docker.com/r/fbtechsolutions/fbelevatordb.

[11]https://survey.stackoverflow.co/2025/technology#1-dev-id-es.

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice: Software Architect Practice_c3* (Addison-Wesley, 2012).

2. B. Zhang, "Extraction and Improvement of Conditionally Compiled Product Line Code," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)* (ICPC, 2012), 257–258.

3. D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems* 35, no. 6 (2010): 615–636.

4. S. Ananieva, S. Greiner, T. Kehrer, et al., "A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications," *Empirical Software Engineering* 27, no. 5 (2022): 101.

5. B. Westfechtel, B. P. Munch, and R. Conradi, "A Layered Architecture for Uniform Version Management," *IEEE Transactions on Software Engineering* 27, no. 12 (2001): 1111–1133.

6. D. Nestor, L. O'Malley, A. J. Quigley, E. Sikora, and S. Thiel, "Visualisation of Variability in Software Product Line Engineering," in *VaMoS* Lero Technical Report (VaMoS, 2007).

7. FeatureIDE, "Elevator SPL," 2024, https://github.com/FeatureIDE/FeatureIDE/.

8. J. Bosch and R. Capilla, "Variability Implementation," in *Systems and Software Variability Management: Concepts, Tools and Experiences* (Springer, 2013), 75–86.

9. J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability With FeatureIDE* (Springer International Publishing, 2017).

10. D. Baum, C. Sixtus, L. Vogelsberg, and U. Eisenecker, "Understanding Conditional Compilation Through Integrated Representation of Variability and Source Code," in *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B* (Association for Computing Machinery (ACM), 2019), 21–24.

11. E. Vacchi and W. Cazzola, "Neverlang: A Framework for Feature-Oriented Language Development," *Computer Languages, Systems & Structures* 43 (2015): 1–40.

12. J. Sametinger, *Software Engineering With Reusable Components* (Springer Science & Business Media, 1997).

13. J. C. Casquina, J. D. S. Eleuterio, and C. M. Rubira, "Adaptive Deployment Infrastructure for Android Applications," in *2016 12th European Dependable Computing Conference (EDCC)* (EDCC, 2016), 218–228.

14. E. S. Almeida, E. C. Santos, A. Alvaro, et al., "Domain Implementation in Software Product Lines Using OSGi," in *Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)* (ICCBSS, 2008), 72–81.

15. S. Goldsack and S. Kent, *Static Typing for Object-Oriented Languages* (Springer, 1996), 262–286.

16. C. Seidl, I. Schaefer, and U. Aßmann, "Capturing Variability in Space and Time With Hyper Feature Models," in *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems* (VaMoS. ACM, 2014), 1–8.

17. D. Hinterreiter, M. Nieke, L. Linsbauer, C. Seidl, H. Prähofer, and P. Grünbacher, "Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution," in *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (ACM SIGPLAN, 2019), 115–128.

18. Z. Xing, Y. Xue, and S. Jarzabek, "A Large Scale Linux-Kernel Based Benchmark for Feature Location Research," in *2013 35th International Conference on Software Engineering (ICSE)* (ICSE, 2013), 1311–1314.

19. M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," in *2010 IEEE 18th International Conference on Program Comprehension* (ICPC, 2010), 14–23.

20. G. K. Michelon, D. Obermann, W. K. Assunção, L. Linsbauer, P. Grünbacher, and A. Egyed, "Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A, SPLC (A)* (Association for Computing Machinery (ACM), 2021), 75–80.

21. W. Fenske, T. Thüm, and G. Saake, "A Taxonomy of Software Product Line Reengineering," in *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '14. VaMoS* (ACM, 2014).

22. J. C. Casquina and L. Montecchi, "A Proposal for Organizing Source Code Variability in the Git Version Control System," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A* (SPLC (A), 2021), 82–88.

23. FBTechSolutions, "FB Elevator SPL," 2024, https://github.com/FBTechSolutions/Elevator-SPL-FB-Blocks.

24. FBTechSolutions, "Elevator SPL," 2024, https://github.com/FBTechSolutions/Elevator-SPL-CC.

25. L. Linsbauer, T. Berger, and P. Grünbacher, "A Classification of Variation Control Systems," in *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Association for Computing Machinery (ACM), 2017).

26. G. K. Michelon, "Evolving System Families in Space and Time," in *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B* (Association for Computing Machinery (ACM), 2020).

27. W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, and D. Lohmann, "Toolchain-Independent Variant Management With the Leviathan Filesystem," in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development* (FOSD, 2010), 18–24.

28. C. Seidl, I. Schaefer, and U. Aßmann, "DeltaEcore-A Model-Based Delta Language Generation Framework," in *Modellierung 2014* (Gesellschaft für Informatik e.V., 2014), 81–96.

29. F. Schwägerl and B. Westfechtel, "SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE, 2016), 822–827.

30. M. Nieke, G. Engel, and C. Seidl, "DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines," in *Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems* (VaMoS, 2017), 92–99.

31. S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ECCO Tool: Extraction and Composition for Clone-And-Own," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2 (ICSE, 2015), 665–668.

32. L. Linsbauer, “A Variability Aware Configuration Management and Revision Control Platform,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)* (IEEE, 2016), 803–806.

33. C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf, “SiPL–A Delta-Based Modeling Framework for Software Product Line Engineering,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (ASE, 2015), 852–857.

34. V. Kruskal, “Managing Multi-Version Programs With an Editor,” *IBM Journal of Research and Development* 28, no. 1 (1984): 74–81.

35. S. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wasowski, “Concepts, Operations, and Feasibility of a Projection-Based Variation Control System,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (ICSME, 2016), 323–333.

36. S. Stanciulescu, “Marlin SPL,” 2024, https://bitbucket.org/modelsteam/2016-vcs-marlin/src/master/.

37. S. Ananieva, S. Greiner, J. Krüger, et al., “Unified Operations for Variability in Space and Time,” in *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems* (VaMoS, 2022), 1–10.