

Received 26 December 2024, accepted 4 February 2025, date of publication 7 February 2025, date of current version 12 February 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3539868

RESEARCH ARTICLE

Fully Automated Software Product Line Evolution With Diverse Artifacts

JAKUB PERDEK¹ AND VALENTINO VRANIĆ²

¹Institute of Informatics, Information Systems, and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, 812 43 Bratislava, Slovakia

²Faculty of Informatics, Pan-European University, 820 09 Bratislava, Slovakia

Corresponding author: Jakub Perdek (jakub.perdek@stuba.sk)

This work was supported by the Slovak Research and Development Agency under Grant APVV-23-0408.

ABSTRACT Existing approaches in software product lines usually neglect knowledge modeling and simulation of the interaction between features capable of bringing dynamism and automation. Consequently, these solutions miss opportunities to resolve associated and emerging problems, including defect detection or quality assurance, which can be solved by effectively extracting and utilizing knowledge from data based on the differences between variants. We bring capabilities to seize them in introducing a fully automated and minimalistic approach to software product line evolution that strictly focuses on handling variability at low-level code fragments. It incorporates the autonomous modeling of emerging knowledge across preconfigured simulations. Specifically, fully automated knowledge-driven software product line evolution provides various views on an existing software product line, its variants, and their evolution through semantic and structural information accompanied by the time and order of the performed changes. We initially developed our approach for software product line evolution, followed by its successful application to the evolution of fractal scripts. We present it accordingly. Fractal products are much simpler because an application state does not span out of recursive behavior, making it manageable within the drawing. Each change is propagated into repetitive phases, causing the visual performance of the implemented feature to be infinitely detailed. Even minor changes in low code fragments tend to manifest as user-visible features owing to recursive behavior, allowing one to massively introduce new features and/or configure existing features and manage variability observable as infinitely detailed shapes. Future applications propose automated observation of more comprehensive in-code representation of feature trees.

INDEX TERMS Aspect-oriented, configuration expressions, knowledge-driven, software product line evolution, variability modeling.

I. INTRODUCTION

Knowledge modeling, along with the simulation of interactions between features, is neglected in most software product lines [1]. Consequently, such solutions miss opportunities to resolve associated and emerging problems, including defect detection or quality assurance, by handling variability in an automated fashion. Knowledge modeling helps organize information from software families that open space to manage reuse among various platforms, each belonging to an independent software product line as in a known

case in the automobile industry [2]. Additionally, handling feature interactions can benefit from the proposed variety of simulations. It can guide the dynamic reconfiguration of a software product line [3], [4] and its testing in an autonomous fashion. Accordingly, they are important in adaptive and competitive software systems, especially software product lines [5], [6], which must be adapted in a particular context and even compete to provide better outcomes.

Many works [7], [8] mined artifacts from repositories beyond source code, including logs, documentation, images, communication, and even built system files. These sources are usually widespread, which gives them a non-negligible role [9]. In contrast, documentation should describe the

The associate editor coordinating the review of this manuscript and approving it for publication was Xueqin Jiang¹.

functionality of a particular code fragment, which usually covers its dynamic nature. Additionally, it is impossible to cover all aspects of automatically expanding code in a software family with artifacts that are typically provided manually. Consequently, a representation directly originating from a code that can abstract from some details to represent aspects of products, primarily in the form of perceived value for the user, is required. We perceive to incorporate the support of this into our product line and then into their evolution.

Full automation has not been proposed in many software product line evolution approaches, especially among those with their evolution specialized [10] to variability [11] so far. Information about “What the system is” encompassing the domain knowledge is expected as the input to evolve DocGen using FDL [12] and reconfiguration to best fit interrelation between components are left unaware. Software is rather decomposed into components that are composed into a single source tree to benefit from a centralized configuration and build [12]. Consequently, no new domain knowledge is emerged from the organization of variability manifesting in its configuration. We perceive such an approach to be problematic for combining and composing small scattered code fragments. Similarly, existing domain knowledge established in the form of a variability configuration is provided as an input to another approach [11]. It optimizes variability by supporting the derivation of restructuring strategies with visualizations of the usage of base products. On the contrary, we avoid applying the restructuring changes in iterative development to predict it with the respective data.

Various specific emerging works tend to justify these demands, such as making configuration expressions, according to which variable functionality is selected into final products, more concise [13], and providing quality assurance, especially repeatable techniques that exceed single-system development [14]. These cases are manually resolved through the participation of fluctuating domain experts and developers. Furthermore, these problems require further automation [10], [11] to reach states when manual management is ineffective and conflicts cannot be easily resolved. On the contrary, related technologies such as conditional compilation [15] or those based on annotations applied in annotation-based software product lines [16], such as frame technology [17], cannot handle these tasks without much effort. Additionally, the development processes must significantly change if aspect-oriented programming extends both techniques in the form of framed aspects [17] owing to aspect orientation and introduced dependencies.

For this reason, this paper introduces a new approach to fully automatically and iteratively evolve annotation-based software product lines by deciding on adding, removing, and updating commonality and variability taken by related strategies and possibly supported by their data-driven extension. Variability is expressed in code using annotations as representative of an annotation-based [16] software product

line. Additionally, this approach can drive the evolution of code, even from the early beginning, when variability is not identified or explicitly marked.

The approach proposed in this paper targets the management of a large number of features, which is one of the main challenges in variability management [10]. Additionally, the required data to perform simulations that primarily enable observation, testing, and optimizing feature interactions are missing. Leaving simulations and modeling the software product line evolution unsupported prevents utilization of its outcomes. Consequently, to tackle these challenges, our approach is initially developed, successfully applied and presented in the case of evolving fractal scripts. They are much simpler than any other entities because the state in the program is held only through functionality in fractals calling itself and the composition of smaller code fragments as constructs to set up constituents. The application of even minor code changes tends to manifest as user-visible features, owing to the recursive behavior responsible for composing constructs into constituents that repeatedly occur in differently configured series. Consequently, constructs can be set up in a large number of ways; however, symmetrical patterns lead to the production of good-looking shapes. These patterns became easily perceived in infinite detail as a feature, thanks to self-dependency in the recursively applied composition of these constructs into their respective constituents. Introducing new constructs can be performed easily and directly into the proposed functionality, which repeatedly calls the self to satisfy the introduced dependencies on previous versions. Accordingly, these dependencies are forced to be reused. We can continuously benefit from reuse, which is propagated further in their iterative evolution, by establishing the software product line. During each iteration, commonalities and variability are managed, along with an optional generation of new assets, which are applied at the end of the evolution iteration phase to derive the final products.

The reduced complexity helps perform various simulations on many product lines and their products, such as optimizing the in-code representation of feature trees to improve their comprehension. Additionally, the approach brings some views of product families through their diverse representations, leading to extensive modeling of diverse aspects of variability through its related representations reflecting the provided code sample.

We bring full automation into our approach and support it with a configurable framework, which is available in GitHub.¹

The remainder of this paper is organized as follows. Section II summarizes the capabilities of various tools towards flexible and automated software product line evolution. Section III presents the adapted concepts and steps of our minimalistic model of automated aspect-oriented

¹<https://github.com/jperdek/automatedSPLEvolutionFramework>

software product line evolution. Section IV demonstrates the configuration, flow, and steps of our approach applied to software product line evolution for drawing fractals. Section V describes in detail the flow and algorithms to fully automate the core of the software product line evolution applied to the evolution of fractals. Section VI provides a containerized version of a fully automated framework with services for scaling and producing diverse representations. Section VII evaluates the capabilities gained from diverse representations for automated and minimalistic evolution and perspectives to extend its application. In Section VIII, the extensibility of the proposed approach to support evolution algorithms, the role of aspects and decorators, and further possibilities and challenges are discussed. Section IX compares our results to what others have achieved. Section X concludes the paper and introduces ideas for future work.

II. IN-CODE VARIABILITY MANAGEMENT: TOWARDS FLEXIBLE AND AUTOMATED SOFTWARE PRODUCT LINE EVOLUTION

Software product lines increase the effectiveness of product delivery by reusing existing assets, including code fragments, tests, and software artifacts. Variability management is necessary to select and integrate these assets as the main component of a software product line. The most commonly used implementation at the code level in the area of software product lines is conditional compilation [15]. Alternatively, a more comprehensive configuration and independence from the used programming language can be achieved with frame technology; specifically, marks/tags are used inside the source code, which becomes a template [17].

The difference in the object-oriented code displayed in Listing 1 from a similar piece of code written in frame technology is demonstrated in Listing 2, where both are taken from the original paper [17]. As can be seen, the frame technology requires a third-party tool to be used. Accordingly, the code must be treated as a template. If aspects are integrated with this technology, it allows for flexibility, reusability, and evolvability as the selection of the prevalent benefits from this combination [18]. Despite these benefits, aspects make solutions dependent on the support and compatibility of third-party libraries or a particular compiler to weave them. Additionally, run-time weaving can cause performance issues [19]. Possible trade-offs can be reached with comments used to annotate variable code fragments [13], [20].

Examples of variability annotations used to annotate some variability related code fragments are shown in Listings 3 and 4, followed by associated configuration expressions in Listings 5 and 6. One of these is our lightweight aspect-oriented software product line method. A demonstration of our lightweight method without aspects in products based on a caching solution written in Java is presented in Listing 7. Here, aspect orientation is used marginally in the background only for variability management to allow the instantiation of a particular product according to

```

1  class Editor extends JEditorPane
    implements HyperlinkListener {
2      private Network network; private
        Hashtable cache = new Hashtable()
        ;
3      // .. methods for adding and
        retrieving data to/from cache
4      //... constructor and editor
        initialization
5      public void hyperlinkUpdate(
        HyperlinkEvent e) {
6          if (e.getEventType() ==
            HyperlinkEvent.EventType.
                ACTIVATED) {
7              String url = e.getURL().toString
                ();
8              Document cachedPage = (Document)
                getFromCache(url);
9              if (cachedPage == null) {
10                 network.requestInfo(this, url)
                    ; addToCache(url, this.
                        getDocument());
11             } else { // get record from
                cache and display it
12                 this.setDocument((Document)
                    cachedPage.getContent());
13             }
14         }
15     }
16 }

```

LISTING 1. Object-oriented implementation of the cache feature (taken from [18]).

configuration and without its derivation at run-time. The code is modularized, even without aspects, and only two annotations are required. The annotation in the caching method is redundant; however, it will be required if this caching feature is decomposed further into other subfeatures. The aspects incorporated into software product line feature management help execute commented functionality in predefined places where associated configuration expressions are evaluated positively. By contrast, aspects are used directly to separate crosscutting concerns in frame technology from aspects and the original lightweight method.

The approach concerning variability management with the introduction of superimposed variants [21] is similar to our lightweight aspect-oriented software product line method [13], but rather in more abstract levels, does not directly concern how entities or structures are expressed for the purpose of variability handling. These can be transformed back into code. Many of the aforementioned representations of variability handling in code use variables to express variation points. Information about variability from these variation points can be composed of configuration expressions, direct conditions for code fragment inclusion/exclusion into derived product instance in the form of conditional compilation, or differently, such as with the feature-oriented domain analysis (FODA) notation [22], [23]. The type of knowledge included in place of the variation points inside the source code has an exclusive configuration character.

```

1  class Editor extends JEditorPane
2      implements HyperlinkListener {
3      <option cache>
4          private Network network; private
5              Hashtable cache = new
6                  Hashtable();
7          // .. methods for adding and
8              retrieving data to/from
9              cache
10         </option>
11         //.. constructor and editor
12             initialisation
13         public void hyperlinkUpdate(
14             HyperlinkEvent e) {
15             if (e.getEventType() ==
16                 HyperlinkEvent.EventType.
17                     ACTIVATED) {
18                 String url = e.getURL().
19                     toString();
20             <option cache>
21                 Document cachedPage = (
22                     Document) getFromCache(
23                         url);
24                 If (cachedPage == null) {
25             </option>
26                 network.requestInfo(this,
27                     url);
28             <option cache>
29                 addToCache(url, this.
30                     getDocument());
31             </option>
32             } else { // get record from
33                 cache and display it
34                 this.setDocument((
35                     Document) cachedPage.
36                         getContent());
37             }
38         </option>
39     }
40 }

```

LISTING 2. Using frame option tags to identify caching code (taken from [18]).

```

1  //PV:IFCOND(pv:hasFeature('HazardWarning
2      '))
3  // get/set value for warning_lights
4  static int warning_lights_value;
5  void set_warning_lights(int state) {
6      warning_lights_value = state;
7  }
8  int get_warning_lights() {
9      return warning_lights.value;
10 }
11 //PV:ENDCOND

```

LISTING 3. The variability-annotated code using pure::variants (taken from [20]).

Specifically, no information associated with developing related code fragments, their previous versions, or their implicit relations to the hierarchical representation of used models is preserved. Consequently, problems related to each approach or technology complicate the evolution of software

```

1  //{"playerNames": "true", "
2      computerOpponent": "true"}
3  import battleship.ComputerPlayer;
4  ...
5  public aspect PlayerName {
6      private String AbstractPlayer.name;
7      ...
8      //{"playerNames": "true"}
9      Player around(): call (Player.new(..)
10         ) && if(Configuration.
11             playerNames){
12         ...
13     }

```

LISTING 4. variability-annotated code using the lightweight aspect-oriented method (taken from [13]).

```
1  pv:hasFeature("HazardWarning")
```

LISTING 5. Configuration expressions identified according to pure::variants (taken from [20]).

```

1  {"playerNames": "true", "
2      computerOpponent": "true"}
3  {"playerNames": "true"}

```

LISTING 6. Configuration expressions identified according to the lightweight aspect-oriented method (taken from [13]).

product lines and affect the resulting derived products. In conclusion, the resulting solutions should not be dependent on a particular technology for variability management, are easy to use for automatization and change realization, support modular development, easy integration and compatibility with old software versions, especially independent of the execution of a particular programming language, and also applicable to different assets.

Extensive studies focused on software product line evolution have evaluated the quality of software product lines across object-oriented and aspect-oriented paradigms [15], but also include the use of conditional compilation [24]. Frequently referenced systems in these studies are the web information system HealthWatcher [15] and the embedded application Mobilemedia [15], [24], [25]. The effective separation of crosscutting concerns, as the main benefit of applying aspects, was observed during the refactoring of HelthWatcher in the evolution process. Specifically, it is efficiently applied to separate ACID principles for persistence and exception handling [26]. Aspects should be preferred in the accomplishment of the open-closed principle according to the increased number of fulfilled requirements for change realization with respect to conditional compilation, which results in a decreased number of modifications applied to the original code [15]. Conditional compilation is one of the most commonly used techniques for implementing software product lines, especially for variability management at the code level. The transformation of the mentioned solutions and their effects on the software product line evolution process, especially for change realization, has probably not been realized for frame technology and its associated


```

1  //@wholeClass({"cache": true})
2  class Cache {
3      private static Hashtable cache = new
        Hashtable();
4      // .. methods for adding and
        retrieving data to/from cache
5
6      //@wholeMethod({"cache": true})
7      public void availableCache(String
        url, Editor editor) {
8          Document cachedPage = (Document)
        Cache.getFromCache(url);
9          if (cachedPage == null) {
10             editor.requestInfo(url); //
                solving previous task-
                visitor pattern should be
                used for various
                functionalities
11             addToCache(url, this.
                getDocument());
12         } else { // get record from cache
                and display it
13             this.setDocument((Document)
                cachedPage.getContent());
14         }
15     }
16 }
17
18 class Editor extends JEditorPane
        implements HyperlinkListener {
19     private Network network;
20     //.. constructor and editor
        initialisation
21     public void hyperlinkUpdate(
        HyperlinkEvent e) {
22         if (e.getEventType() ==
        HyperlinkEvent.EventType.
        ACTIVATED) {
23             String url = e.getURL().
                toString();
24             //@skipLine({"cache": false}},
                "[NOT=Cache.availableCache(
                url, this);]")
25             this.requestInfo(url);
26         }
27     }
28     public void requestInfo(String url)
        {
29         network.requestInfo(this, url);
30     }
31 }

```

LISTING 7. The caching example based on our modification of the lightweight aspect-oriented method [13] with the derivation of aspect-free products.

aspect-oriented extensions known as framed aspects [17]. In this case, problems with templates and the need for a particular software tool likely outweighed the benefits of potentially improving extendability and those perceived by some possible combinations of these technologies. Conversely, significant problems in aspect-oriented programming were identified during the refactoring of the database system [27] and calculator [28]. The inability to capture

and extend some places in a given source code, especially inside methods, is required to apply hooks to hang aspects on them. Similarly, for realized use cases, their application is problematic for implementing variability management.

Introducing a new functionality can cause conflicts among aspects that contain effectively modularized crosscutting concerns. These conflicts are usually handled by applying other aspects on separated layers [29] in proposed layered architecture. These are the consequences of quantification and obliviousness, which, on the contrary, belong to the most influential advanced aspect-oriented techniques and are necessary to achieve the majority of benefits [30]. For the first one, it is possible to affect various places in a code, and the second refers to the impossibility of identifying how a given code is affected only by itself. Despite conditional compilation and templates, the code is maintained in particular aspects. The use of aspects requires specifying all relevant pointcuts that are affected. Consequently, new side effects are introduced, including additional code complexity and aggravated traceability of the affected code fragments, which do not directly show signs of how they are affected. The mentioned conflicts between aspects that are handled on separate layers require the application of advanced aspect-oriented techniques, which usually manifest with complex specifications of pointcuts. Their complexity issues can be resolved using simpler, restricted, and widely spread aspect-oriented techniques such as decorators or strictly specified pointcuts bound to a specific type or attributes/functionalities. Additionally, they can be applied as independent concerns associated with modular code fragments and are separable from business functionality. Their removal is performed through an abstract syntax tree (AST) representation filtered according to naming conventions.

Variability management can be introduced in several ways. Universality, independence from the specific programming language, lower complexity of variability management constructs, flexibility of change realization, compilability, independence from practices and management practices associated with software product lines, simplicity, and interoperability can be achieved using the lightweight aspect-oriented method [13] for feature management. The lower in-code complexity of variability management constructs in favor of the lightweight method over wrappers used in pure::variants, frame technology with all its variants, and other approaches is proven by converting them into an in-code version and evaluating their complexity in the context of each variability-annotated file [31]. Similarly, using aspects for change realization is far easier owing to the non-invasive join points [18]. A specific product can be directly tested simply by launching a program as the proof of compilability and execution. Its lightweight nature prescribes only three types of annotations to copy whole files, methods, and one-line code fragments, respectively, where fulfilling such work is done with an additional tool that can be customized

for any language [13]. On the contrary, considering its disadvantages, such as relying on aspects, quantification and obliviousness, the problem of overall separation of variability management from constructions in code, and the inability to evaluate the complexity of used annotations and configuration expressions, it is beneficial to remove aspects and possibly thanks to the use of decorators and additional transformations as an extension of the mentioned approach. The application of aspects that guarantee the full removal of aspects during product derivation, as well as the independence of products on aspects, can better help tune the existing functionality of the specific product that is instantiated owing to aspects during development.

The domain itself significantly influences the variability management and configuration process of individual assets, primarily with efforts towards extendable and maintainable solutions [15]. Their outcome focuses on how given software fragments are organized, including modeling common and variable features for specific situations and contexts, usually in the form of a software product line architecture [32]. Despite the direct focus of this architecture on variability management [33], which guarantees the simultaneous capture of various products [34] with an explicit focus on the software product family, including overall support by defining all parts of these products [35], which are significant differences in comparison with a reference architecture, the principles of variability modeling are not rigorously applied in general by scientific publications or papers, nor in practice [1]. The identified lack of rigor during the simulation of interactions between features is bypassed by directly applying domain knowledge to solve various tasks, especially applying decision-making at the design level by proposing a variability model. The limitations of overly abstract models complicate the automation of software product line evolution processes. Consequently, an integrated approach based on linear algebra [36] is focused on extending the related information to the decision-making of architects through instantiated views according to the synthesized feature models in this process. Each is based on a structural analysis combined and balanced with semantics.

Knowledge modeling, simulation of interactions between features, and application of principles of variability modeling are processes in which proper incorporation into solutions requires a large number of potentially demanded products on the output. The integration of various feature selections can create unwanted products or cause defects that can be handled during optimization [10]. The search to identify defects usually starts from more abstract levels, particularly with the analysis of feature models [37]. Semantic networks allow the modeling of diverse artifacts of resulting products, various representations of products, and associated data extracted from software, together with the advantage of deriving new facts [38]. The automatic creation of fast-instantiated outputs at the code level, which are also direct products and for simplicity, without other dependencies associated with

interacting modules, is possible by applying the lightweight aspect-oriented method for variability management to fractal drawings (in the form of environments) [13]. Similarly, such an application provides opportunities to effectively create various representations from the associated metadata of each product, especially how its code is organized, and thus jointly contributes to the knowledge of their software product family. These capabilities make the evolution process bound to the original code and directly to software product line families.

III. A MINIMALISTIC MODEL FOR AUTOMATED SOFTWARE PRODUCT LINE EVOLUTION

Most software product lines neglect knowledge modeling and simulation of the interactions between features [1], which are essential for supporting software product line dynamics and extracting domain knowledge. Consequently, we designed an approach to aspect-oriented software product line evolution based on our lightweight method [13], which handles variability in a minimalistic and automated manner.

Its primary capability is to fully automatically and iteratively evolve annotation-based software product lines [16] by deciding on adding, removing, and updating commonality and variability taken by related strategies and possibly supported by their data-driven extension. Variability is expressed in code using annotations to represent the annotation-based [16] software product line. The mechanism is designed to manage variability in a selected script. The script is optionally extended with the provided software artifacts, such as code fragments from similar software product line scripts.

Minimalism lies in managing and accommodating the evolution of software product lines, relying only on decisions regarding commonalities/variability and introducing new variable functionality. Consequently, the reusable assets are redefined, and variability predefines the specifics of the final products while continuing to evolve. Each combination of these brings the establishment of a new evolved software product line differentiated by new way how its chosen common assets are reused and which functionality will be selectively propagated into final products, followed by updates on particular variation points about new exported functionality. Specifically, the performed operations consist only of effects related to variability handling, which are captured iteratively in its iteration into the newly evolved software product line. The difference between the evolved software product line and its predecessors is restricted to variability handling. Decisions are made through preconfigured strategies used exclusively for handling variability. Additionally, enabling or disabling specific functionalities, such as injecting supporting functionality to incorporate a data extraction mechanism into a new version of an evolved software product line, is configurable through a few global settings.

The approach is presented on fractal scripts of low complexity capable of observing configurations of variability management annotations where visible features are

recognizable into infinite detail owing to the introduced minimalism. Handling semantics is initially omitted to be handled later with data sources as a transition from less complex to samples with higher complexity.

Additionally, the lack of specialized models with bound domain knowledge or insufficient detail to resolve the incorporation of new features from high abstraction levels complicates the measurement of the correctness and quality of a particular interaction between features. We tackled and resolved such problems with the aforementioned minimalistic mechanism, managing the incorporation of new code-level features with the full support of diverse data representations extracted directly from instantiated code structures and semantics. In the case of our software product line for drawing fractals, these diverse representations consist of code (commands to draw shapes), graphs (methods or data structures as nodes and calls between them as connections which are dynamically created during drawing of shapes), rasters (a visual image of shapes drawn on output), and vectors (markup language to draw shapes), which are intended to be integrated together in the form of ontology (triples).

The lightweight nature of our approach is preserved in variability handling ensured in the sequence of respective strategies through TypeScript decorator technology [39]. We refer to this sequence as a variability handling pipeline. It consists of groups of strategies organized in a sequence (see Fig. 1), which are selectively employed to leverage complexity. Most of these strategies handle injections of existing functionality into selected places that are responsible for introducing new features. We applied it to alter recursive behavior during the drawing of fractals, lowering possible modifications and increasing effects perceived as dependencies of the functionality on itself caused by repeatedly calling oneself. Fractal drawing is performed by updating each constituent through the update of constructs occurring in place of recursively called functionality, which results in the introduction of a separate feature in each iteration. Consequently, the pipeline for variability handling as part of the software product line evolution is executed per iteration to directly track these important changes. One or more strategies are selected from each group in a particular iteration. The pipeline for variability handling is responsible for decisions or reconsidering inclusions of processed code fragments into variability and commonality, injecting new code based on contextual information into the processed code, and providing and organizing various representations of particular software product lines or even product variants. The full range of diverse related data representations captures the evolved stages of a particular product family. Diversity is integrated and organized into a unique dataset or ontology.

Managing the variability in each iteration of the software product line evolution begins by processing annotated or a totally variability-unaware script. In the pipeline for variability handling, this processed script is divided into

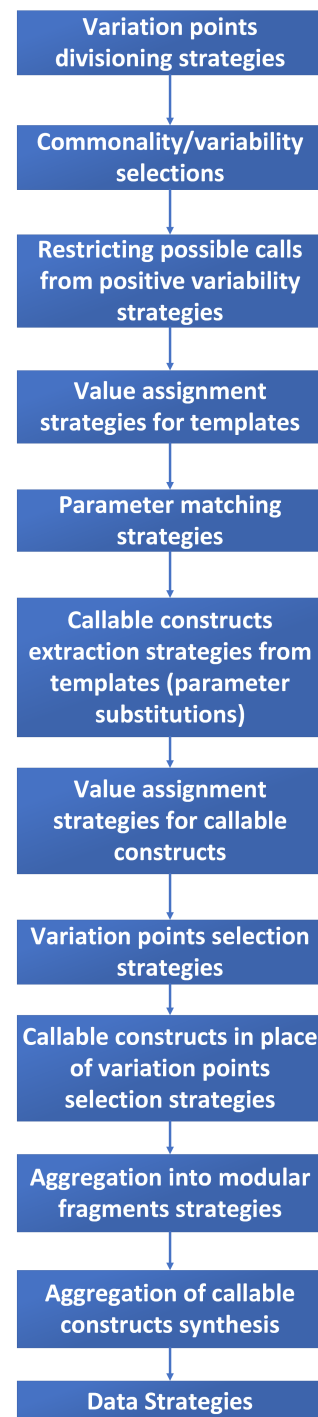


FIGURE 1. Sequence of variability decision strategies.

variation points, what is performed by *Variation points divisioning strategies* (see Fig. 1). The strategies designed for this purpose determine which if not all, the variation points in the script should be considered. The resulting variation points are processed using the next group of strategies (called *Commonality/variability selections*). Particular

strategies re-evaluate their affiliation to the common and variable if appropriate annotations are available; otherwise, they are divided into these two categories. Consequently, the script is enriched with information about variability, or this information is reconfigured.

Several strategies exist for managing and restricting the selection of substitution candidates to the observed variation points in the previous phases of the variability handling pipeline.

The first one restricts the possible calls from positive variability. It filters possible calls of a particular functionality that can be injected at a specific variation point position. Such calls are transformed into templates to prepare them for substitution, followed by assessing their code complexity or applying custom metrics. The available parameters or variables are substituted into templates to extend the existing functionality while avoiding the majority of defects due to undefined or inaccessible variables/parameters. Substitution occurs according to the availability of variables/parameters at specific positions in the source code of a particular programming language.

Recommendations for parameter substitution in the form of regular expressions can be obtained from parameter decorators and used in parameter-matching strategies. Currently, the parameter name must be a substring of the name extracted from the template or vice versa. The aggregated admissible parameters and variables are checked against each template associated with a particular variation point. If a substitution is successful, the injection of such a call introduces no defects. Extraction of executable calls from templates across all considered variation points and their further preparation are the responsibility of callable construct extraction strategies from templates. Their preparation includes the aforementioned substitution of parameters/variables.

The selected callable constructs from the previous variability handling pipeline phase are assessed according to a particular metric or custom strategy. We incorporated and initially applied a strategy based on code complexity metrics [31]. Other metrics can be used to consider the relevance of callable constructs, especially on a semantic basis.

The preparation of the calls that can be substituted is complete at this point. The pipeline for handling variability continues with the creation of features. Firstly, the destination variation points are selected from among all those found at the beginning of this pipeline for variability. The aggregation of calls associated with these points is optionally used to consider the number and quality of available constructs. Secondly, one or more callable constructs are selected from these variation points according to the configuration and strategy used. These constructs may be aggregated into larger entities such as methods or classes. This aggregation demonstrates the introduction of more complex and modular entities in the code. Thirdly, aggregation among all selected calls associated with the selected variation points is created to cover features containing scattered concerns and ensure

their incorporation in one iteration. Finally, the variability in the evolved software product line should be optionally documented using various views, considering the code, its dynamics, and the resulting product according to visual appearance, or last but not least, as a list of drawn entities. The incorporation of functionality to create diverse representations in an evolved software product line is the responsibility of a group of data strategies. Decisions on which and how diverse representations will be integrated and used are left to bridge the following new evolution iteration. The process is incorporated into software product line evolution to dynamically extend and analyze software product lines, particularly to gather domain knowledge and optimize configuration expressions. Concatenating the aforementioned evolution iterations or the entire evolution leads to an organized and configurable evolution of the entire software product line. Our approach allows us to adapt and evolve a single-system software tool based on our evolutionary process/model as a software product line.

IV. A DEMONSTRATION OF FULLY-AUTOMATED VARIABILITY HANDLING: FIVE-SIDED FRACTAL EVOLUTION

Variability can be studied on families of fractals. Each representative has a function where, after the application of multiple changes, it remains a fractal. Fractals are geometric objects created in a recursive or iterative process that can show infinite detail [40]. They consist of at least two shapes called the base and generator, where the generator changes parts of the base by applying a given mechanism, such as scaling, rotation, or translation [40]. Additionally, the generated samples can be processed using third-party models that evaluate their quality or generate fractals on their own. The original software product line can evolve using the most appropriate ones. A decision on which variation points should be removed or added should also be made.

We created six different fractals with variations, including various programming styles restricted by the functionality provided in a particular programming language. TypeScript programming language is used in the presented case. The expected outcome of fractal evolution is to find shapes or fractals with high aesthetic value. Transferring such fractal evolution into a field of software product lines primarily requires variability handling and quality enhancement. Our demonstration focuses on the latter, contributing to precise analysis, modeling, and support of variability handling. Consequently, we are extracting and adapting diverse representations across fractal evolution that capture all these aspects (aesthetics, commonalities and variability in a software product line, quality of a software product line, etc.), and open space to introduce, incorporate, and tune appropriate models into this evolution. We demonstrate our approach, its capabilities, and the results from evolving a five-sided shape; however, the previously applied configuration is applied almost similarly to the evolution of other shapes.

Consider a simple script written in TypeScript, as presented in Listing 8.

```
1 var newVariable = "Hallo";
2 function a(param1, param2) {
3   @wholeClass({ "outsideGallery": "true"
4   })
5   class GG {
6     callMe() { /* ... callMe function
7       logic ... */ }
8   }
9 }
10 /* ... other content ... */
```

LISTING 8. A simple TypeScript code.

The parser reads the entire file and transforms it into an abstract syntax tree. Abstract syntax trees make processing code easier, especially in recognizing and harvesting each large code entity, such as a class or function. Obtaining their beginning and terminating positions is not complicated because of the occurrence of these values in the processed abstract syntax tree. Other smaller code constructs are associated with these entities, according to the semantics of a particular programming language. For example, a class variable, even declared later, can be used in each non-static class function. After these steps, the specific entity is placed into the context hierarchy according to its position in the abstract syntax tree. An example based on the previous Listing is shown in Fig. 2. Global variables from the program are stored separately outside the hierarchy.

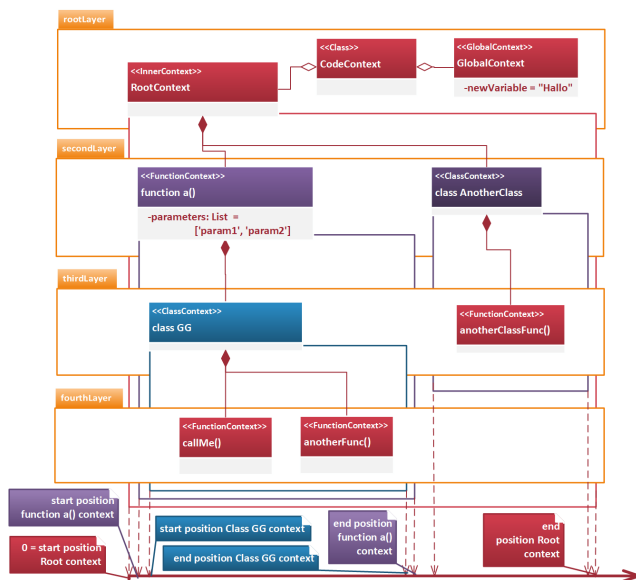


FIGURE 2. The hierarchy of entities with preserving semantics.

Three additional layers are created because of the declaration of the class inside the function and the class functions. This hierarchical data structure gathers information about particular entities used to specify possible constructs and injects it in place of a specified variation point. For positive

variability, new variables are declared in place of possible variation points with such an interoperable JSON expression encompassing possible calls that can be instantiated or even available parameters or variables. Additionally, all possible places that belong to negative variability, including existing entities, are annotated using system annotations. The variability in the program is incorporated into the processed abstract syntax tree and then into the code according to the following points:

Marking positive variability

markerVP[id] Variability is expressed using detachable decorators

Annotating negative variability

AnnotationVP[id] Variability is expressed using detachable decorators with variability configuration expressions

Configuring parameters

paramVP[id] The rules expressed using regular expressions are applied to control the substitution of annotated parameters while converting code templates into callable constructs.

The result after this phase using an implementation capable of finding all the variation points is shown in Listing 9.

```
1 var markerVP1 = { "global": {}, "inner":
2   { "p": 699 } };
3 @AnnotationVP1.variableVP() var
4   newVariable = "Hallo";
5 var markerVP2 = { /*...*/ };
6 @AnnotationVP2.functionVP() function a(
7   param1, param2) {
8   var markerVP3 = { /*...*/ };
9   @AnnotationVP3.classVP()
10   class GG {
11     @AnnotationVP10.classVariableVP()
12     markerVP6 = { /*...*/ };
13     @AnnotationVP4.classFunctionVP()
14     callMe() {
15       var markerVP4 = { /*...*/ };
16     }
17     @AnnotationVP11.classVariableVP()
18     markerVP7 = { /*...*/ };
19   }
20 var markerVP15 = { /*...*/ };
21 /* ... other content ... */
```

LISTING 9. Incorporating variability information into processed script.

The algorithm used to find variation points can be replaced with its version, which is restricted to processing only particular places, such as classes. A unique identifier is assigned to each annotation or marker. In this phase, configuration expressions can be checked or generated in place of annotations to manage the variability of the existing entities. In-code complexity metrics for these negative variability annotations and their configuration expressions were assessed using an approach to measure in-code variability based on decorators [31]. The measured complexities can

be used to select, compare, and optimize these configuration expressions, especially in an automated fashion. Finally, all found and expressed points with markers or annotations are persisted in a JSON file to aggregate the harvested variation points in one place. The use of the JSON format supports interoperability. The captured data represent information regarding commonalities and variability in the processed script and the state of the processed sample in a particular iteration of evolution. They are used for two reasons related to variability. The first is to select places to inject new content, leading to the introduction of new changes in the evolving software product line. The second is to swap some of the selected negative variability variation points with those not chosen. This leads to the derivation of different products due to the selected variable features incorporated into those responsible for achieved reuse.

Persisting the variation points enables the removal of negative-variability system annotations from the annotated variation points. Only positive variability markers, each containing code contexts, serve as the next optional injection of a particular content into their position. The file now appears as shown in Listing 10.

One or more such markers can be replaced with a functionality that is feasible to call. Such a call must be part of the marker configuration and the substituted parameters should be valid in a particular context. The substitution process consists of selecting some of the positive variation points, obtaining constructs that can be substituted, creating templates where parameters are substituted according to their type, substituting these parameters if their name is contained in the parameter name in the template and types match, and finally creating a limited number of new updated software product lines for this content. Each step is handled by a particular strategy to decrease processing demands or support diversity by easily exchanging it. For example, a strategy that selects recursively executed functions is applied instead of the original one, returning all functions.

The changes are applied to an abstract syntax tree with incorporated system markers from the beginning of the iteration. Such an abstract syntax tree is larger and the positions of previously observed entities have changed. To properly find parameters and variables for substitution in the created hierarchy, it is necessary to find mappings between abstract syntax trees, especially their larger entities, such as functions or classes representing nodes. For this purpose, we obtain the identifiers of these entities along with their type, and organize them in sequences created according to the path from the root. After processing both trees, we mapped the blocks of the abstract syntax tree with the information regarding the captured entities to the identifiers of these entities. Using markers, we propagated the correct positions from the original abstract syntax tree to the updated version, which is much larger owing to variability annotations and markers. After deriving five-sided fractal

```

1  var markerVP1 = { "global": {}, "inner":
    { "p": 0 } };
2  var newVariable = "Hallo";
3  var markerVP2 = { "global": { "
    globalVariables": [{ "name": "
      newVariable", "type": "string" }] }, "
    inner": { "p": 0 } };
4  function a(param1, param2) {
5    var markerVP3 = { "allAvailableCalls":
      ["a([%[param1: any]%, [%[param2:
        any]%)"]], "global": {
6      "globalVariables": [{ "name": "
        newVariable", "type": "string
        " }] }, "inner": { "p": 709,
        "allAvailableCalls": ["a([%[
        param1: any]%, [%[param2:
        any]%)"]], "callable": "a([%[
        param1: any]%, [%[param2:
        any]%)"]], "
        availableParameters": [{ "
        name": "param1", "type": "any
        " }, { "name": "param2", "
        type": "any" }], "
        functionName": "a", "
        contextType": "Function", "
        returnType": "any" } }];
7    @wholeClass({ "outsideGallery": "true"
        })
8    class GG {
9      markerVP6 = { "allAvailableCalls":
        ["a([%[param1: any]%, [%[param2:
        any]%)"]], "global": { "
        globalVariables": [{ "name": "
        newVariable", "type": "string"
        }] }, "inner": { "p": 327, "
        allAvailableCalls": ["a([%[
        param1: any]%, [%[param2: any
        ]%)"]], "callable": "", "
        contextType": "Class", "
        className": "GG", "functionality
        ": ["callMe()"] } }];
10     @wholeClassMethod({ "outsideGallery
        ": "true" })
11     callMe() {
12       var markerVP4 = { "
        allAvailableCalls": ["a([%[
        param1: any]%, [%[param2:
        any]%)"]], "global": {
13       "globalVariables": [{ "
        name": "newVariable", "
        type": "string" }] }, "
        inner": { "p": 323, "
        allAvailableCalls": ["a
        ([%[param1: any]%, [%[
        param2: any]%)"]], "
        callable": "callMe()",
        "functionName": "callMe
        ", "contextType": "
        Function", "returnType"
        : "any" } }];
14       /* ... callMe function logic ...
        */
15     }
16     markerVP7 = { /* ... */ };

```

LISTING 10. Illustration of left positive variability in processed script.

```

17     }
18     var markerVP8 = { /* ... */ };
19 }
20 var markerVP15 = { /* ... */ };
21 /* ... other content ... */

```

LISTING 10. (Continued.) Illustration of left positive variability in processed script.

products, their successful immediate rendering in the browser proved that the mapping was correct and no wrongly used variables or parameters occurred.

A. ESTABLISHING A PRODUCT LINE BASED ON A FIVE-SIDED FRACTAL

We make an extendable implementation of a five-sided fractal with included information about the coordinates of the larger and six smaller fractals using knowledge from planimetry.

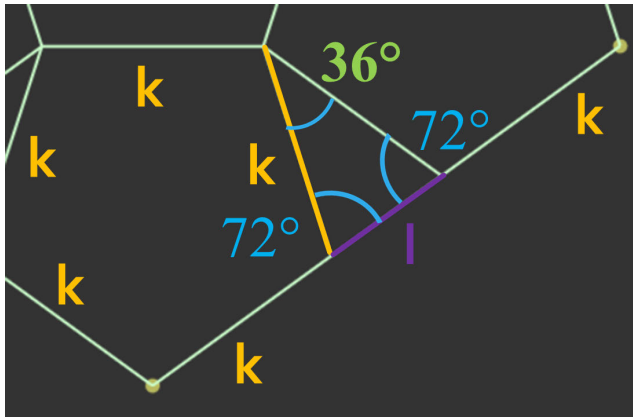


FIGURE 3. Calculating subpart of fractal.

Initially, we calculated the length of the smaller five-sided side based on the size (length) of the larger five-sided side. We used the following information drawn from Fig. 3. Accordingly, we used the sine theorem based on the middle triangle:

$$\frac{k}{\sin(72)} = \frac{l}{\sin(36)} \quad (1)$$

where angle 72 is obtained by creating a five-sided inside circle as $360/5 = 72$ and 36 is obtained from the triangle as $180 - 72 - 72 = 36$. Length of part between two smaller sides (k) on the large side (the side of a large five-sided shape):

$$l = \frac{k * \sin(36)}{\sin(72)} \quad (2)$$

The length of the larger side (the side of a large five-sided shape) is:

$$n = 2 * k + l \quad (3)$$

Putting them together results in getting a smaller side from a larger one:

$$k = \frac{n * \sin(72)}{2 * \sin(72) + \sin(36)} \quad (4)$$

Then, to draw a fractal, we must obtain the coordinates of points A_1 and A_2 inside the line $|BA|$ that delimits the large side into smaller sides as shown in Fig. 4. Mentioned delimitation is possible by using linear interpolation:

$$A_i = [(1 - g_i) * A_{ix} + g_i * B_{ix}, (1 - g_i) * A_{iy} + g_i * B_{iy}] \quad (5)$$

where parameter g_1 is used to determine point A_1 as $g_1 = |k|/|n|$ and parameter g_2 is used to observe point A_2 as $g_2 = |l + k|/|n|$.

Subsequently, it is necessary to determine point T , where two smaller five-sided shapes touch each other. Firstly, we obtain the length of the middle tangent line crossing this point and the center point S . We obtained a triangle with a smaller side equal to half of side l , side of small five-sided shape k and unknown side r depicted in Fig. 5. The Pythagorean theorem is used to obtain the following expression:

$$r^2 = k^2 - (l/2)^2 \quad (6)$$

Secondly, we get the coordinates of point S as:

$$S[A_1x + (A_2x - A_1x)/2, A_1y + (A_2y - A_1y)/2] \quad (7)$$

Thirdly, we used interpolation to find coordinates with parameters evaluated based on the ratio that is calculated as $g_r = r/w$ where r is $|ST|$ length and w is the length of the large fractal side, including points A and B as $|BA|$:

$$M = [(1 - g_r) * A_x + g_r * B_x, (1 - g_r) * A_y + g_r * B_y] \quad (8)$$

Finally, we observed two possible tangent points according to the rule expressed by the equation $|ST| = |AM| = T - S = M - A$. These tangent points T can then be found by refining previous equation into $T = (M - A) + S$, which leads to:

$$T[(M_x - A_x) + S_x, (M_y - A_y) + S_y] \quad (9)$$

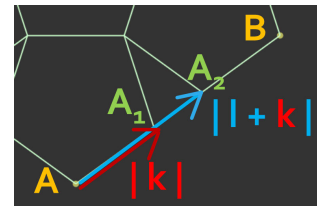


FIGURE 4. Points delimiting the side of a larger five-sided fractal into two sides of smaller fractals.

followed by switching the coordinates to get resulting perpendicular points to line AB :

$$T[-y, x] \text{ as } T[-(M_y - A_y) + S_x, (M_x - A_x) + S_y] \quad (10)$$

$$T[y, -x] \text{ as } T[(M_y - A_y) + S_x, -(M_x - A_x) + S_y] \quad (11)$$

We select a point whose distance from the opposite point (D in this example, according to Fig. 6) is smaller.

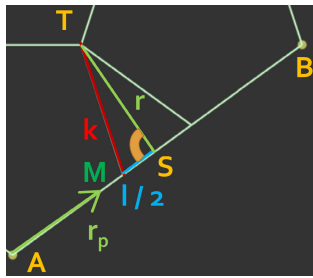


FIGURE 5. Getting upper point T: distance r and tangent line from r_p .

We draw a line between every pair of points from $A1$, $A2$, and T . Analogically, we evaluated and connected each of the remaining inner points: $T2$, $T3$, $T4$, and $T5$, with two particular delimiting points to draw six similar, smaller five-sided shapes. The resulting fractal is shown twice in Fig. 6. The first one highlights found T points and the second all known points used to made constituent out of constructs (to ensure the creation of fractals [41]) and which can be modified to further update this process.

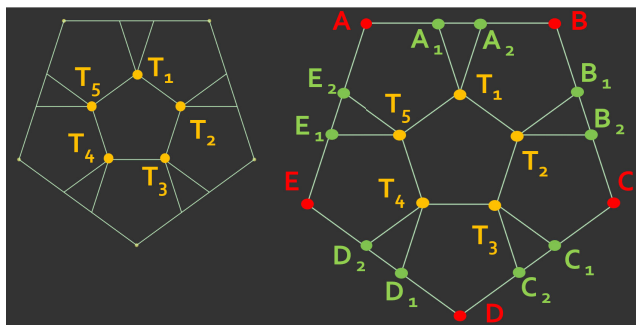


FIGURE 6. Inner and connected points of recursively divided shape.

The mathematical proof based on planimetry demonstrates recursively evaluated equations within the program. The transition from a larger five-sided fractal into each of the six smaller fractals recursively is the key to the propagating similarity between the construct (smaller five-sided shapes contained in the large five-sided shape) and constituent (large five-sided shape) during the creation of this shape. Changes in the evaluated core functionality reflected in the equations can possibly modify how the respective positions are calculated and new patterns are prescribed. Compared to drawings made by one stroke, this additionally provides the coordinates of larger (constituent) and all of its inner smaller five-sided fractals (constructs). Even L-systems based on recursively replaced substrings are shown to be more complex because they directly rely on context-sensitive grammar instead of the context-free variant used in our five-sided implementation. Their limitation lies in the interpretation of particular symbols and the need to draw them sequentially without differentiation of layers created as a consequence when the program dives deeper in recursion during drawing. Consequently, extendability in the presented planimetry

solution is increased by more possibilities for applying particular patterns in the context of recurrently expanding the five-sided shape performed as direct management of constituents created from respective constructs inside a recursive function (usually one). Accordingly, other fractals are created in a similar manner.

The stub code abstracted from some calculations is based on the following equations that appear in Listing 11.

B. INITIALIZING EVOLUTION OF PRODUCT LINE BASED ON FIVE-SIDED FRACTAL SCRIPT

The framework must be configured to introduce a new functionality into recursively executed locations. Each change applied inside the recursion has a high probability of being symmetrically applied quantitatively in an observable manner to form a feature and possibly to preserve its fractal nature. The configuration of the evolution depth and changes in its parameters have various effects during reuse inside recursively executed functionality. Such changes must be observed in the resulting data or code provided in the final phases of each evolution iteration. We can assume that we are creating one feature in each evolution iteration and only from the structural information on the input, owing to recursion. This assumption helps configure the strategies used to inject new functionality within the iteration. For this purpose, each evolution iteration is configured independently and connected to the previous iteration in the sequence.

Various switches from canvas contexts must be applied to increase the drawing possibilities and reduce the number of parameters of certain methods. Therefore, our software product line requires no state-preserving variables or modules. For this purpose, we must only allow and use the `canto-js` [42] library. This library provides wrappers under all HTML canvas methods to save previously used coordinates, thereby allowing turtle graphics and method-chaining. This simplifies future adaptations and narrows the possibility of extending previous fractal versions by omitting the use of parameters of integrated modules and similarity measures to find correct variables with the responsibility of propagating coordinates through the module for further drawing. The library is required to insert specific script files, is initialized as a wrapper to the previous 2D context of the HTML canvas object, and provide exported functions that will be integrated. Reducing the number of methods, their arguments, and the possibilities to insert them helped optimize operations (especially those with exponential complexity) inside the framework and flexibly evolve the solution. Consequently, exports of the input scripts should be excluded and batched according to the focus on the momentary construction of the positive variability increment.

Another important setting is to plan which code fragments can be injected and in what numbers. The provided setting is responsible for managing the injection of a particular functionality in the evolution iteration only once or again


```

1  @DecoratorTypesService.wholeClass({"
    fiveSide": "true"})
2  export class FiveSideFractal {
3
4    getFiveSideShapes(
5      iteration: number, sideLength:
        number, x1: number, y1: number,
        x2: number,
6      y2: number, x3: number, y3: number,
        x4: number, y4: number, x5:
        number,
7      y5: number, fiveSide: FiveSide,
        context:
        CanvasRenderingContext2D): void
      {
8        /** VARIABLE INITIALIZATION **/
9        /** EVALUATING LENGTHS AND PERFORMING
            NECESSARY SIZES **/
10
11        if(iteration > 0){
12          /** GETTING REMAINING 5 INNER POINTS
13
14          // MAKE FINAL LINES
15          if(iteration == 1){
16            /* DRAWING 5 OUTER LINES */
17            /* DRAWING 5 TRIANGLES FROM EACH LINE
              - CONNECTING FOUND INNER POINTS
              WITH 2 POINTS ON PERPENDICULAR LINE
              (TAKEN FROM THE REMAINING PART IN THE
              MIDDLE AFTER TAKING TWO SIDES OF
              SMALLER FIVE SIDE FROM IT) */
18            /** CONNECTING INNER 5 POINTS TO FORM
              FIVE SIDE In THE MIDDLE */
19            /** EXTEND RECURSION TO 6 FIVE SIDES
20            } else {
21
22            // LEFT UP
23            this.getFiveSideShapes(iteration -
24              1, smallerSideLength, right12x,
25              right12y,
26              x2, y2, left23x, left23y, new23x,
27              new23y, new12x, new12y, fiveSide,
28              context);
29            // LEFT DOWN
30            this.getFiveSideShapes(iteration -
31              1, smallerSideLength, right23x,
32              right23y,
33              x3, y3, left34x, left34y, new34x,
34              new34y, new23x, new23y, fiveSide,
35              context);
36            // RIGHT DOWN
37            this.getFiveSideShapes(iteration -
38              1, smallerSideLength, right34x,
39              right34y,
40              x4, y4, left45x, left45y, new45x,
41              new45y, new34x, new34y, fiveSide,
42              context);
43            // RIGHT UP
44            this.getFiveSideShapes(iteration -
45              1, smallerSideLength, right45x,
46              right45y,
47              x5, y5, left51x, left51y, new51x,
48              new51y, new45x, new45y, fiveSide,
49              context);
50            // UP

```

LISTING 11. Stub code from the script to draw a five-sided fractal.

```

36      this.getFiveSideShapes(iteration -
37        1, smallerSideLength, right51x,
38        right51y,
39        x1, y1, left12x, left12y, new12x,
40        new12y, new51x, new51y, fiveSide,
41        context);
42
43      // MIDDLE
44      this.getFiveSideShapes(iteration -
45        1, smallerSideLength, new12x,
46        new12y,
47        new23x, new23y, new34x, new34y,
48        new45x, new45y, new51x, new51y,
49        fiveSide, context);
50    }
51  }
52
53  drawFiveStar(context:
54    CanvasRenderingContext2D, radius:
55    number, iterations: number,
56    thickness: number) {
57    const fiveSideMapInfo = new
58      FiveSideMapInfo(context, 5, 300,
59      300, 400);
60    const fiveSide = new FiveSide(
61      fiveSideMapInfo, iterations,
62      thickness);
63
64    this.drawShapeAndStoreVertices(
65      context, fiveSideMapInfo,
66      fiveSideMapInfo.ruleString,
67      fiveSideMapInfo.angleForPoint,
68      fiveSideMapInfo.sideLength,
69      radius, fiveSide); //draws
70
71    outhere five side
72    this.getFiveSideShapes(fiveSide.
73      iterations, fiveSide.
74      sideLengthInitial, fiveSide.
75      vertices[0].x, fiveSide.vertices
76      [0].y,
77      fiveSide.vertices[1].x,
78      fiveSide.vertices[1].y,
79      fiveSide.vertices[2].x,
80      fiveSide.vertices[2].y,
81      fiveSide.vertices[3].x,
82      fiveSide.vertices[3].y,
83      fiveSide.vertices[4].x,
84      fiveSide.vertices[4].y,
85      fiveSide, context) //
86      recursively draws five side
87  }
88 }
89
90 function drawAnkletModMain(contextMain:
91   CanvasRenderingContext2D, radius:
92   number, iterations: number, thickness:
93   number): void {
94   let fiveSideFractal = new
95     FiveSideFractal();
96   fiveSideFractal.drawFiveStar(
97     contextMain, radius, iterations,
98     thickness);
99 }

```

LISTING 11. (Continued.) Stub code from the script to draw a five-sided fractal.

after a predefined number of iterations. The application of the setting (configuring a particular evolution iteration) to choose and inject some functionality only once results in a lack of such functionality in the derived software product lines, where another functionality was selected. In addition, the preconfiguration of some variables that can be directly substituted, such as the line length or diameter of the radius, is beneficial. We provide these in the form of global declarations in a separate file and link the file system path inside the configuration. Despite this configuration, substitution of the found parameters is performed in place of a particular variation point based on the available type. For semantic reasons, the parameter name should be included in the substitution candidate variable to narrow the range of possibilities.

C. RECURSIVELY DRIVEN FIVE-SIDED FRACTAL-SOFTWARE PRODUCT LINE EVOLUTION

We ran an automated aspect-oriented knowledge-driven evolution process on a five-sided fractal script without configuring internal variability. A high-level view of the entire process is shown in Fig. 7. In its application to the evolution of fractals, shown in Fig. 8, especially five-sided fractals, the focus is on creating multiple representations as a basis for knowledge aggregation and further processing. The first two steps involve variability handling and product derivation.

Consequently, the code and its structure are perceived as characteristics of the software product line family. Accordingly, the script is converted into an abstract syntax tree to manage all associated functionalities. In parallel, scripts with supporting code fragments are parsed, followed by the extraction of exported code fragments in a similar fashion. The next step is to handle negative and positive variability. Firstly, the existing entities in place of the variation points are chosen and annotated as variable. Accordingly, managing the negative variability is a fully automated step. Secondly, the exported functionality must be placed inside the particular variation points between existing entities from the previous step. The observed context must be adhered to to prevent defects by substituting unavailable parameters or calling functionality into a forbidden place. The injection of the new functionality is performed directly on the abstract syntax tree of the processed script. Additionally, variability markers and annotations are removed from the tree.

The project that evolved in the current iteration is copied for each intended extension. Each updated abstract syntax tree is converted back to code and its old version is replaced from the previous iteration in the copied project. The evolution cycle is closed by continuing the next iteration in an already project from this evolution iteration phase. The product of the evolved five-sided fractal software product line can also be derived. Optionally, multiple representations are extracted and organized in a dataset or ontology, as described in detail in Section IV-D.

When the derivation is completed, it is feasible to manually evolve some of the derived solutions from tailored functionality.

Mixing various fractals will not easily bring aesthetic value that usually originates from recursion and repetitions and prevents even roughly associating this value. This complication especially holds when code constructs are incorporated as impenetrable exported modules with their own interfaces, based on the parameters provided to them. In contrast to traditional tasks for synthesizing various code fragments, similarity or quality metrics (the complexity metrics are measured using the TyphonJS-ESComplex service [43]) do not help in this process because they are not associated with recursion. Consequently, observing recursion patterns according to the relationship between elements of a different order (constructs/constituents) and more adapted self-similarity [44] in parallel with additional metrics are necessary to drive incremental changes in each phase. These rules are fulfilled by preserving continuous lines or fitting shapes on the path of the base fractal. Accordingly, the drawing of the new shape starts at the point on the canvas where the drawing of the previous shape stops. For this purpose, we extend the functionality only inside recursive functions.

To achieve this, we incorporated a recursion-detection mechanism. The original annotated abstract syntax tree containing inserted positive variability markers and the position of the class and method declarations is provided as the input. The declared function and class names are stored in the hash map, pointing to a set of names from all called functionalities inside them. Similarly, the start and end positions of the declared functionality from the abstract syntax tree are harvested to detect if a given marker or variation point is inside, and then executed inside recursion. The detection process iterates over all stored keys and, for each of them, attempts to check whether some inner calls point to its name. If this holds, the cycle is found and verifies the recursively evaluated code fragment; otherwise, no recursion is detected. The last step is to include this information for both types of variation, namely positive and negative points, and to persist on their list. The start and end positions of these annotations and markers are selected because of the unavailability of their names in the map of the detection mechanism. Finally, checking whether the variation point is inside recursion is performed by finding any function, class function, or class name that was previously evaluated as recursive. The analyzed point is located inside them. In the evolution process of fractals, this information is valuable for selecting such points, repeating patterns, and whole fractals at different levels.

More complex information about the number of repeatedly called functions among each other can be evaluated, and the direct process of applied repetitive patterns can be split into this recursively evaluated sequence. The evolution process of the five-edge fractals in the first evolution phase is shown

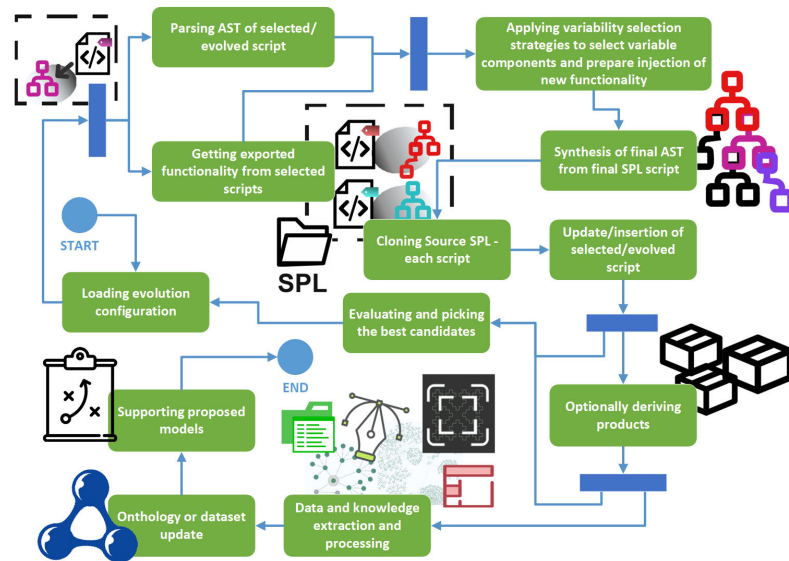


FIGURE 7. High-level view on software product line evolution process.

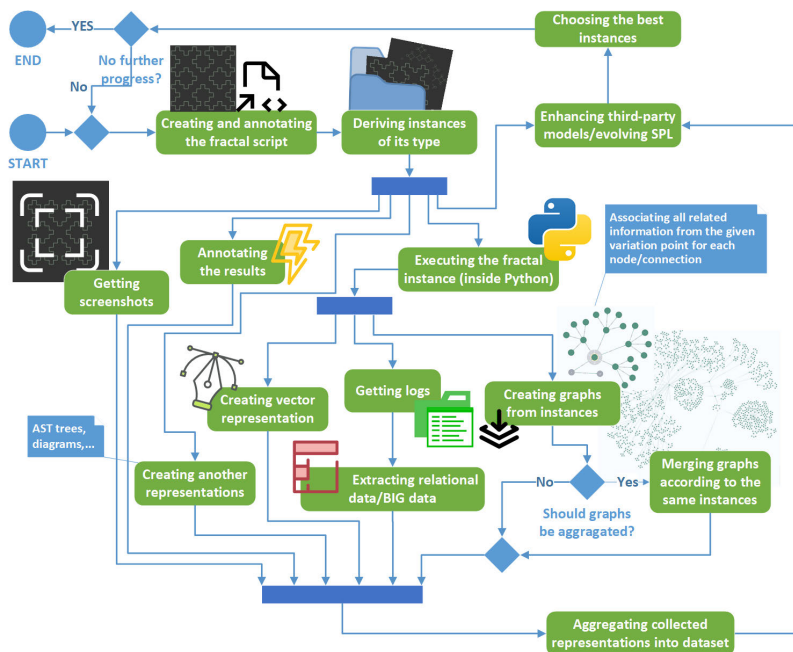


FIGURE 8. The process of getting various representations from the fractal script and using it for the software product line evolution.

in Fig. 9. The first fractal displayed is the base. Each of the remaining ones emerged after inserting a call into a recursively evaluated function responsible for creating a filled circle with a different radius in the center under a previously used position.

Domain knowledge was thus shown to be important in directing the evolution phase and the overall evolution process towards fulfilling requirements. In the comparison-driven evolution process, according to the similarities between the base and integrated code fragments, the rules

derived from the requirements are not too strict to precisely formulate suitable similarity values. However, deriving new annotated software product lines that lost some of their variable functionality can theoretically lead to correct adjustments made with the help of knowledge gained from previous phases. Positive variability without a proper domain-driven mechanism is difficult to orchestrate. Verification of the entire software product line is handled directly inside the browser by loading pages and resources. The product is functional if no exception is thrown and the script is

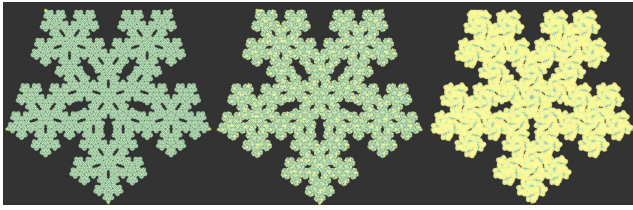


FIGURE 9. Deriving various fractals in recursion-oriented software product line evolution phase.

executed entirely. Testing products for aesthetic value or taking various measures requires extracting appropriate data such as screenshots and using them in advanced decision models. These products are derived during the derivation process and are accompanied by an associated version of the evolved software product line.

D. IN-PRODUCT VIEWS: DIVERSE DATA AND SOFTWARE KNOWLEDGE REPRESENTATIONS

When the core of our automated variability handling mechanism has processed the five-sided fractal, it emanates samples of a similar five-sided fractal with a preconfiguration of its inner variability extended with a new feature. The changes in the variability configuration allowed us to derive slightly different products. The emerging instance of the five-sided fractal software product line and its possibly derived products must be analyzed and abstracted from particular information in multiple ways, each being treated as a product-representative view. Each view expresses different in-product characteristics, particularly in the case of each diverse data representation. For example, a vector representation can capture every drawn object in its respective order. In contrast, a raster representation is characterized by a frame filled with pixels from objects already drawn on the canvas. A semi-structured representation captures documents created from instantiated objects or is more impoverished than other representations by preserving only some relations obtained from the program execution. Complementary to the aforementioned representations is graph construction, which is capable of capturing created entities, such as classes or functions, during program execution and incorporating them into a graph. This is the most significant representation capable in its visualization to provide a view of the entire software family if such product graphs are merged according to similar nodes from the root node, and information from such incremental aggregations of each merged node is properly represented directly inside the found similar node. Such information can consist of a counter or a list of deviations from the similarity between the compared nodes. Additionally, integrating all of these representations under a flexible scheme in semantic ontology improves further manipulations with such diverse data, especially satisfying a variety of queries. As part of the evolution of the fractal software product line, time-series data that capture incremental and iterative additions are associated

with a particular state of an evolved (five-sided) fractal. Diverse representations can be generalized to capture similar information from any other executable code, not only from a fractal. In this case, the five-sided fractal is captured as:

Vector

A sequence consisting of the described objects that will be drawn. Elements containing information on the geometric shapes occurring in fractals and optionally including animation elements sufficiently cover everything from the rendered outcome of the final product. Accordingly, it is a descriptive representation of the final product containing a description of the value-bringing elements/behavior of such a product abstracted from the code.

Raster

Drawn fractal itself as a raster image. This key representation captures the value of the entire product for end users, which is an aesthetic value in case of drawing fractals. This representation can be generalized to screenshots displaying key features, especially user interfaces.

Semi-structured

The logs used to capture the relationship between parameters and their values are harvested from repetitive diving into recursion during fractal creation. This representation is easily obtainable from any executable if custom logging is incorporated.

Graph

Sequence of created or called functionality while executing a five-sided fractal script. Similarly, as in the previous case, it is feasible to construct for any executable software product line script.

Relational

Additionally evaluated quality/complexity metrics, user assessment, or associated labels. Generalizable for any executable.

Code

Executable five-sided script itself (or any script).

Ontology/Triples

Integration of all representations of a five-sided script together (or any script).

Time-Series

The sequence of historical and future changes is in accordance with the evolution of a particular software product line.

The whole fractal is effectively captured in the respective views presented as puzzles that are combined into a whole using ontology, as shown in Fig. 10. Consequently, with their help, some simulations can help detect various problems and direct software product line evolution towards multiple goals. To the best of our knowledge, the available studies have ignored these aspects related to software quality and/or feature management solutions incorporated into the provided views and prepared to be integrated as ontology about a particular fractal product (five-sided in this case). Our

approach enables simulations based on automated software product line evolution and its further adapted extensions.

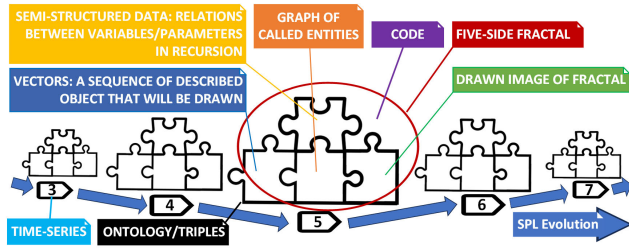


FIGURE 10. Relations among diverse representations for presented five-sided fractal in software product line evolution.

Knowledge modeling helps organize and derive new existing knowledge according to a suitable schema. In contrast, ontology creation must be adapted to the available methods for specific problems to achieve the highest accuracy [45]. Automated knowledge modeling methods are preferred, but require more data. Consequently, the derivation and evolution of products from the fractal software product family should satisfy this requirement. Domain knowledge accumulated and contained within the evolved five-sided fractal software product line lying in its preconfigured variable features must be extracted and used further to derive new knowledge and for automatic and autonomous reasoning.

Helper functionality elicits raw data to craft these representations, which are optionally embedded in a particular software product line and propagated into individual derived products. Its essential capabilities include simulating the stack, dynamically capturing the created objects after execution, and storing them in graphs. They require special code to be optionally injected before and after the body of the called method or executable code block. After executing a particular code, a hierarchical representation is created in the background and exactly from the executed code, thereby allowing the capture of a predefined flow. No aspect-oriented weaving is necessary because of the optional insertion of such functionality strictly for data-extraction purposes in preconfigured cases. This process is illustrated in Fig. 11. The other functionality for logging conditionally related parameters to capture variable names and their values at a certain recursion depth for calling a particular component is managed similarly. The principle of how the logged variables from the input are represented is illustrated in Fig. 12. Similarly, to capture vector graphics as commands or representative entities, the helper code converts shapes into a vector representation instead of directly drawing them during execution.

Some tools do not need to be integrated into a particular sample of our five-sided software product lines. Third-party programs, including screenshoters, classifiers, data pre-processing, or analytic models, transform extracted raw data into final representations or directly open and execute scripts in the web browser. Additionally, screenshots are taken from the graphics elements generated by a specific

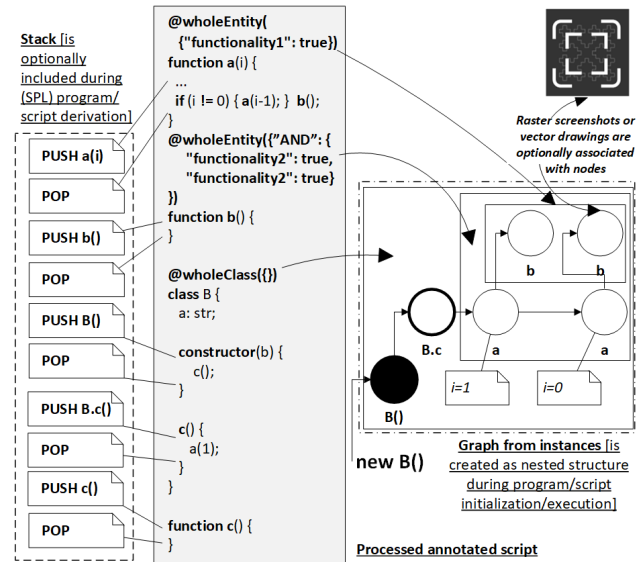


FIGURE 11. Injecting stack data into derived software product line script or final product and dynamically created hierarchic structure of pushed instances.

Processed annotated script

```
@wholeEntity({'functionality1': true})
function a(l, j, k) {
  ...
  if (i != 0) { a(i-1, j, k); } b();
}
```

Semi-structured logs

```
log(i, j, k)

{"k_i=1_j=10": k}    {"k_i=1": k}
{"j_i=1_k=5": j}    {"j_i=1": j}
{"i_k=5_j=10": i}    ...
```

FIGURE 12. Injecting logging functionality to log the combination of pairs where each consists of a variable name with its values.

component or code fragment. Semantic networks should ensure tracing between these screenshots and the elements that generated them. Screenshots are applicable because of the capability of the resulting five-sided fractals to be rendered and displayed immediately in the web browser. In the case of analytical tools, user preferences can be accurately predicted using GNNs [46], [47] (Graph Neural Networks) or logistic regression [48].

V. FLOW AND ALGORITHMS IN FRAMEWORK FOR FULLY AUTOMATED SOFTWARE PRODUCT LINE EVOLUTION

The core of our fully automated evolution process includes parsing and divisioning the script into variation points, extracting context information, using it to manage variability,

and producing newly evolved software product lines. These are optionally selected for the next iteration. The high-level flowchart diagram is shown in Fig. 13. Its initial phase involves processing the selected base script and other scripts with exported functionality to provide possible sources for extending this base script. The process can be divided into three phases. Firstly, the context is extracted from each or selected variation point in the application, and the changed copy of a abstract syntax tree (AST) with system annotations of negative and positive variability is produced. This phase is displayed as a subprocess called the *Divisioner process* in the flowchart diagram in Fig. 13, also completely in Fig. 14, and in more detail using the activity diagram in Fig. 15. Secondly, the configuration expressions in place of the variation points are extracted, collected, and persisted in the variation point sequence. The process flow is shown as a subprocess called the *Extraction process* in Fig. 13 and completely in Fig. 16. Finally, the evolution algorithm is applied to determine and handle the positive and negative variability in the evolution process. The remaining two subprocesses from the high-level flowchart diagram (Fig. 13), followed by their synthesis into a new version of the evolved software product line, close the evolution loop. Fig. 17 shows this phase in detail.

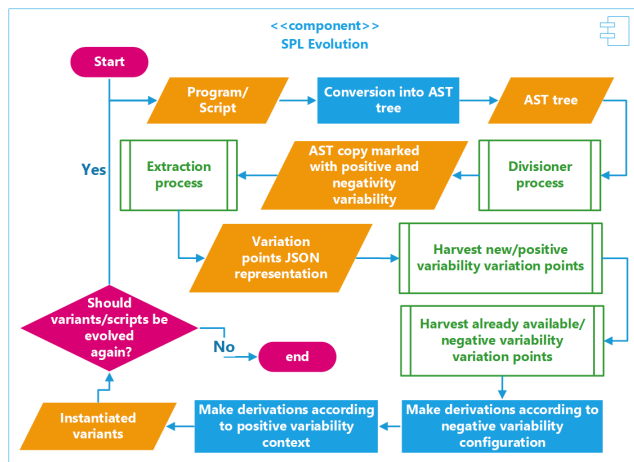


FIGURE 13. Overall flow of the core of fully automated software product line evolution process.

A. INITIAL PHASE: EXTRACTING CONTEXTS

Firstly, the base and suitable code fragments are chosen as extensions of existing applications/scripts. The base application is selected either manually or automatically. The initial phase is shown for each of the scripts on the input in Fig. 14, and the entire flow concerning details about the division of the abstract syntax tree into variation points and annotating/mark them is shown in Fig. 15. Secondly, the applications are loaded and parsed, which includes the observing local and global context inside and between the in-code constructs of a particular programming

language. A hierarchical data structure presented in the previous example (Fig. 2) is created for each entity, with nodes consisting of classes and methods. This data structure allows for flexible orientation and semantic associations of smaller fragments with nodes inside the hierarchy during the processing of an abstract syntax tree. Another benefit is the ability to harvest all entities before the processed entity in sequence or search for a given context. In our proposed software product line evolution process, these abilities are used to mark and harvest the application context of given variation points that belong to both positive and negative variability. Each harvested context is unified according to the chosen schema into a JSON document and concatenated into the sequence.

The mechanism presented earlier is in the case of software product line evolution applied repetitively to each application/script. Export statements from used extensions must be gathered from other applications/scripts. For this purpose, the previous process is used, but it can be simplified and optimized optionally, as shown in Fig. 14. In addition, mapping of each export to its context is discovered. Exports are then aggregated for each file and under the related context. Similarly, aggregation under a given type helps to effectively determine suitable function parameters during the instantiation of callable constructs.

B. THE ROLE OF COMMONALITIES AND VARIABILITY MANAGEMENT IN SOFTWARE PRODUCT LINE EVOLUTION

Commonality and variability are mutually exclusive if the original product line is not extended to new functionality. Too much variability usually results in fewer quality products and too much commonality in not many products. However, creating many evolved software product lines with their diverse representations is necessary because of the knowledge-driven nature of our evolution approach and decision-making dependence on big data. The decision of what should be common and variable is important for economic reasons to achieve as much reusability and satisfy as many customers as possible [49]. Swapping from common parts to variable parts is necessary for extendability and automation through dynamic updates. These commonality/variability updates are achieved by setting or removing user annotations for variability management from a processed application or its corresponding abstract syntax tree.

C. CONFIGURING AND SWAPPING COMMONALITIES AND VARIABILITY: NEGATIVE VARIABILITY HANDLING

Negative variability handling requires the extraction of all available user annotations with given configuration expressions. Additionally, system annotations can be created for these points. If these annotations are unavailable, system annotations can be created to mark possible variation points that are perceived as common. All such changes are propagated to the final copy of the abstract syntax tree in the cases allowed by the configuration. User annotations

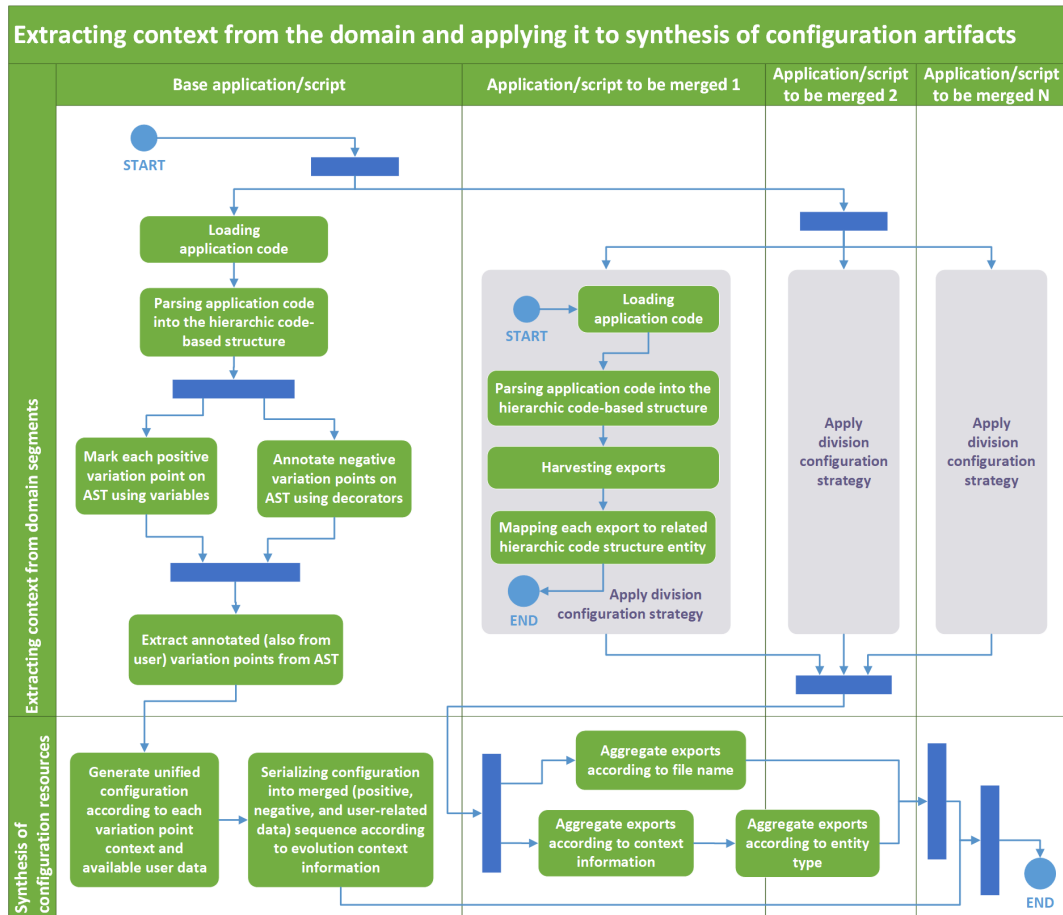


FIGURE 14. Initial phase of our automated evolution: Extracting context from applications and synthesizing configuration resources.

are always propagated in its copy. In the case of positive variability, the process is different. The variation point is marked with a declared variable initialized with context information, all possible calls, accessible global and local variables, and information if the context is inside a class. In abstract syntax tree, these places can be found inside an array of statements or members. Incorporation of changes requires taking this into account with proper indexing. Elements are inserted in the beginning, middle, and the end. Operations are visualized in Fig. 15. All information is extracted and persisted in the next phase, which is separated to allow its flexible adaptations.

Various strategies have been employed to direct evolution to produce extendable, maintainable, and economical solutions that can be instantiated and evaluated directly. The process of selecting and deselecting features as an extension of the negative variability handling is shown in Fig. 18. Some of these strategies focus on preserving the same number of common and variable features, decreasing or increasing common or variable features, and using metrics and user-annotated data to direct it towards the expected balance.

D. THE SEQUENCE OF PRECONFIGURED STRATEGIES TOWARDS CUSTOMIZED QUALITY-DRIVEN CODE SYNTHESIS: POSITIVE VARIABILITY HANDLING

Various code constructs can be synthesized to flexibly support various requirements that can be fulfilled during our evolution process. Thus, positive variability handling must be bound by the capabilities of programming languages. Many languages are based on hierarchical structures in which one entity is declared as part of another. In the case of classes, there is no dependence on the sequence of class method declarations when they call each other. Therefore, other restrictions should be met. For example, wrapped methods cannot be accessed directly at higher levels using JavaScript. All of these restrictions are necessary to incorporate positive variability handling strategies to derive compilable products. Flow of various strategies to manage the selection of features in place of variation points, creation of related constructs, and various score assignments are presented in Fig. 19.

Positive variability markers are created during the initial phase and inserted into the copy of abstract syntax tree. In addition, each possible code construct given by actual context, especially by position inside analyzed content,

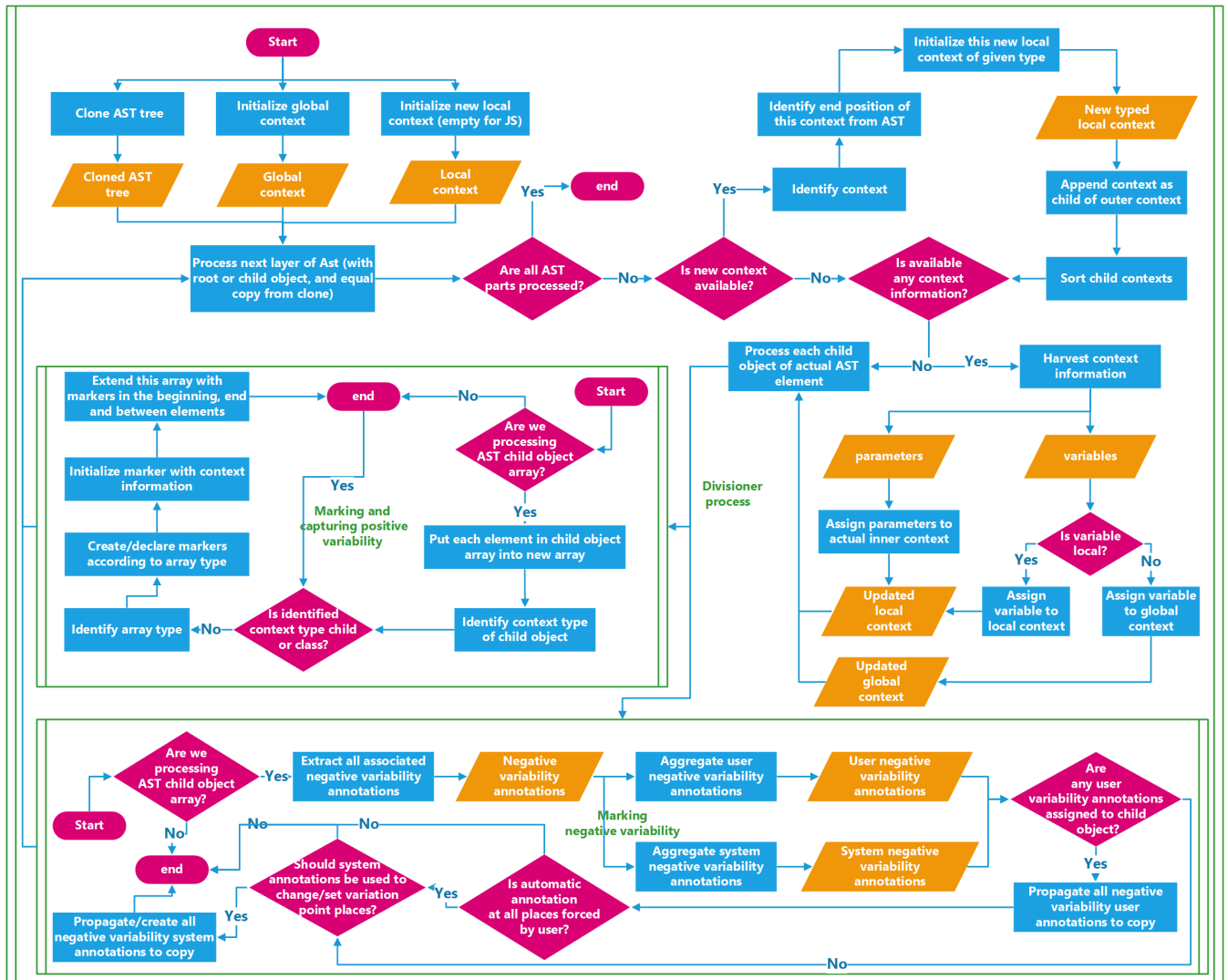


FIGURE 15. The process of getting application context from each variation point and marking/annotating these points on the copy of the abstract syntax tree.

is transformed into a template and assigned to the corresponding marker inserted behind the place of analyzed context at this phase.

Firstly, the strategy for obtaining stored calls from the variation points is selected and transformed into templates. These templates are aggregated for each variation point. This includes loading all such calls, obtaining those that do not include more entities to be instantiated or calls to be concatenated, and extracting them according to their previously assigned priority or only according to the necessary types. Restrictions in this step help to reduce the number of possible candidates and, ultimately software product line derivations. Secondly, the templates should obtain the assigned quality/priority scores according to their complexity, semantics, or coupling relatedness. However, this sub-phase can be omitted if the constructs have already assigned values. Thirdly, different metrics and scores must

be combined, ideally in a model with weights for each metric/score. Other possible strategies include choosing related metrics for a given iteration to focus directly on quality, semantics or structural aspects.

Additionally, the associated decorated parameters of functions or constructors are accessed from the variation point configuration and used to reduce the number of potential available substitution candidates. Similarly, various strategies can be used for this purpose. Match whole variables exactly, omit matching completely, use preconfigured regular expressions, preconfigured inheritance relations, or use similarity metrics amongst available parameters and words extracted from decorators. After this phase, the remaining available candidates are substituted into templates for each variation point, leading to the generation of callable constructs. These constructs can be directly called what is used to measure their complexity. One predefined strategy out of

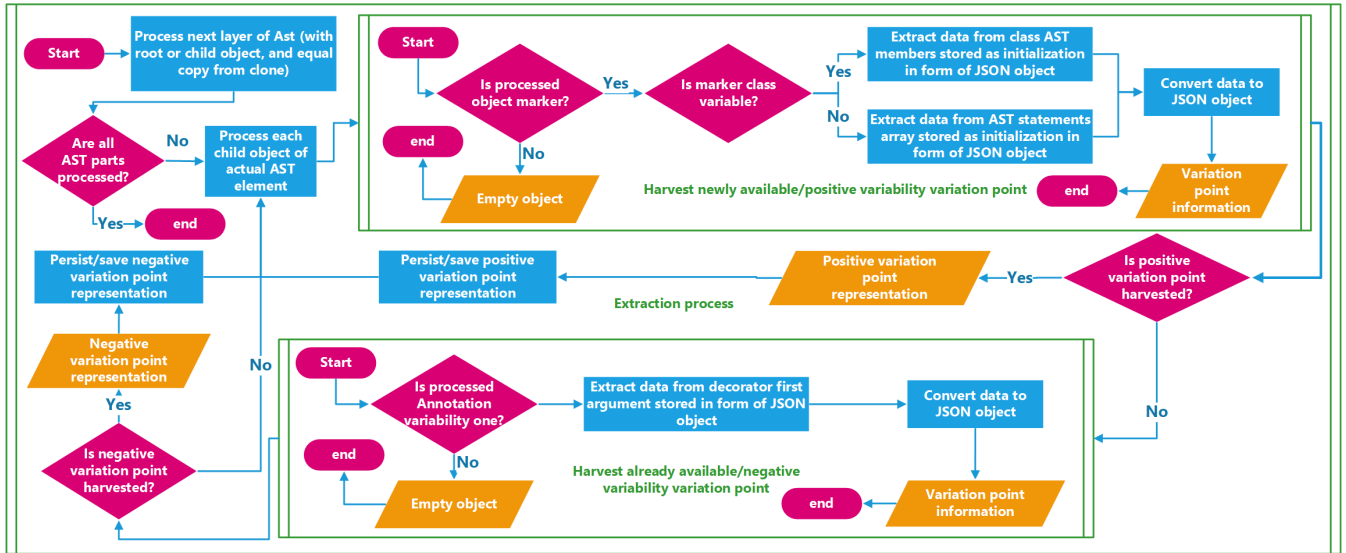


FIGURE 16. Extraction of expressions from annotations/markers with associated information into a sequence of variation points data.

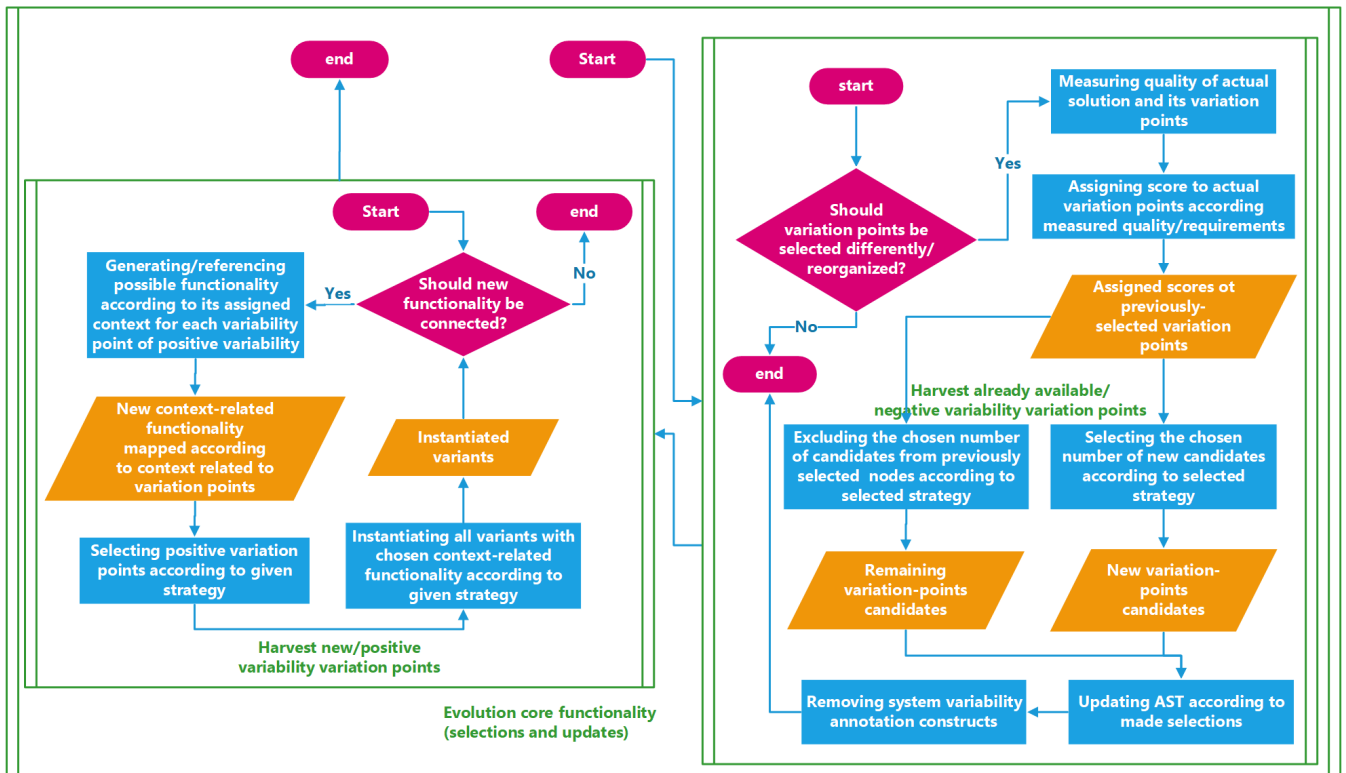


FIGURE 17. Evolution core mechanism: deciding about common and variable & incrementally incorporating new functionality.

the possible substitution strategies is used. All such calls can be instantiated according to the optionally available type, the candidate variable can be substituted into a given template construct only once or a restricted number of times, and the candidate variable can be substituted according to the similarity metrics between the variable and substituting parameter or based on the occurrence of parameter inside

the candidate variable. Similarly, for templates, the scores of the given constructs are evaluated, assigned, and used to calculate the overall score for each callable construct. Native tools are used to measure the complexity constructs in JavaScript. Similar strategies used for templates are applied more precisely because of the substitution of exact parameters, such as similarity metrics.

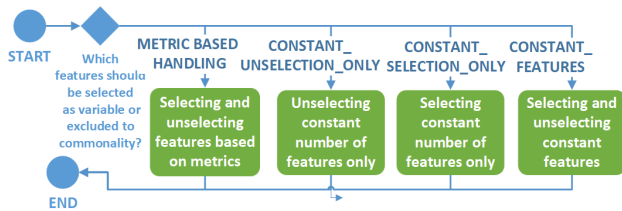


FIGURE 18. Selections and deselections of features during handling of negative variability.

E. SEQUENCE OF PRECONFIGURED STRATEGIES TO SELECT FEATURES FROM CONSTRUCTS FOR MASSIVE AUTOMATED SOFTWARE PRODUCT LINES PREPARATION

The critical step is to select which variation points and strictly which constructs associated with each feature should be used in the resulting prepared software product lines and then in their derived products. An additional selection of constructs amongst the features on the variation points is required to adequately model the scattered features that crosscut two or more variation points. Mentioned selections, in case of proper configuration, ensure that the software product line can be extended to include any functionality that is perceived as a user-visible aspect or feature. The final part of the evolution process core is shown in Fig. 20.

Firstly, the variation points provided by a set of templates are selected. We perceive these selections as the selection of features. The available strategies include selecting a constant number of features, similarity-based selections between previous functionality and functionality on a given variation point, or topologically based selections oriented according to where variation points are located, such as in global space, inside methods, or classes. Secondly, certain callable constructs in place of each variation point should be selected. Associated strategies include the selection of constant callable constructs for each selected variation point, session-based selection to balance performance and reduce the number of candidates given by configuration, and topological selection based on the similarity of each sub-selected aggregated construct. Used callable constructs can be proposed in different order, and thus, the number of resulting instances can increase.

Thirdly, a suitable data structure should be proposed according to the selected constructs for each variation point. Proposed structures, such as lines of code, including various declarations, variable assignments or calls, and methods, can usually be injected in place of a given variation point and thus do not crosscut. In the case of classes, their declaration should be moved to global space. Various strategies can be used to handle these operations. A more common strategy is based on granularity, which is determined by the number of lines of code. One or two lines of code are separately injected into a given variation point separately. A few lines of code are moved into this method. More lines of code or methods are converted into classes, particularly if many variables are assigned, or methods are aggregated. Variables

can be assigned inside the constructor, and setters and getters can be added automatically. In the case of methods, one method should be selected and converted into a constructor. Alternatively, all calls should be moved to a newly created constructor. The exact configurations for the size of specific modular fragments must be discovered during the evolution process. In contrast to this type of strategy, similarity-driven algorithms can be employed. Lines of codes are aggregated under the method only when the common context is identified or when the probability reaches a given threshold. Similarly, the same principles apply when aggregating the variable declaration methods under the given classes. This option is effective if many software product lines are generated in the actual evolution iteration and can be driven by data from code structures instantiated and harvested during the execution of evolved functionality from these software product lines. Aggregating variables inside the class also depends on their number, and thus requires combining both strategy types.

The next step to finalizing features is to select previously created data structures that should be propagated to finally produced software product lines, thus solving the problem concerning introducing and incorporating scattered features. Intended strategies include selecting all of these structures, selecting all up to n structures, selecting constructs according to similarities amongst different variation points, or selecting exactly one. The last-mentioned strategy helps to dramatically reduce the number of proposed evolved software product lines in this iteration compared to the remaining ones. The first and second generate up to a factorial of evolved software product lines produced in this iteration according to the number of generated structures in the previous step.

Each evolved software product line can be optionally checked for duplicity according to configuration, followed by its removal in the case of concordance. After this phase, various functionalities are optionally inserted, such as logging various events or simulating a stack with the possibility of harvesting all instantiated instances during the execution of the evolved software product line script. This allows for the production of semi-structured and graph data. Raster data can be obtained for specific code fragments owing to the possibility of launching a product inside the browser. These representations are extracted from evolved software product lines, from which products can be derived optionally, providing knowledge about the quality and structure of these evolved software product lines produced in this iteration based on the evolved software product family.

Finally, the software product line production mechanism ensures that all required assets are copied into the folder of the newly evolved software product line, inserts selected structures into an abstract syntax tree, removes system markers and annotations, preserve only user ones, inserts imports from used structures, transforms them into final code, and replaces the previous content of base application/script with this new one. Abstract syntax tree and the new version of the variation points sequence are persisted to properly reference previous iterations. The final evolved software

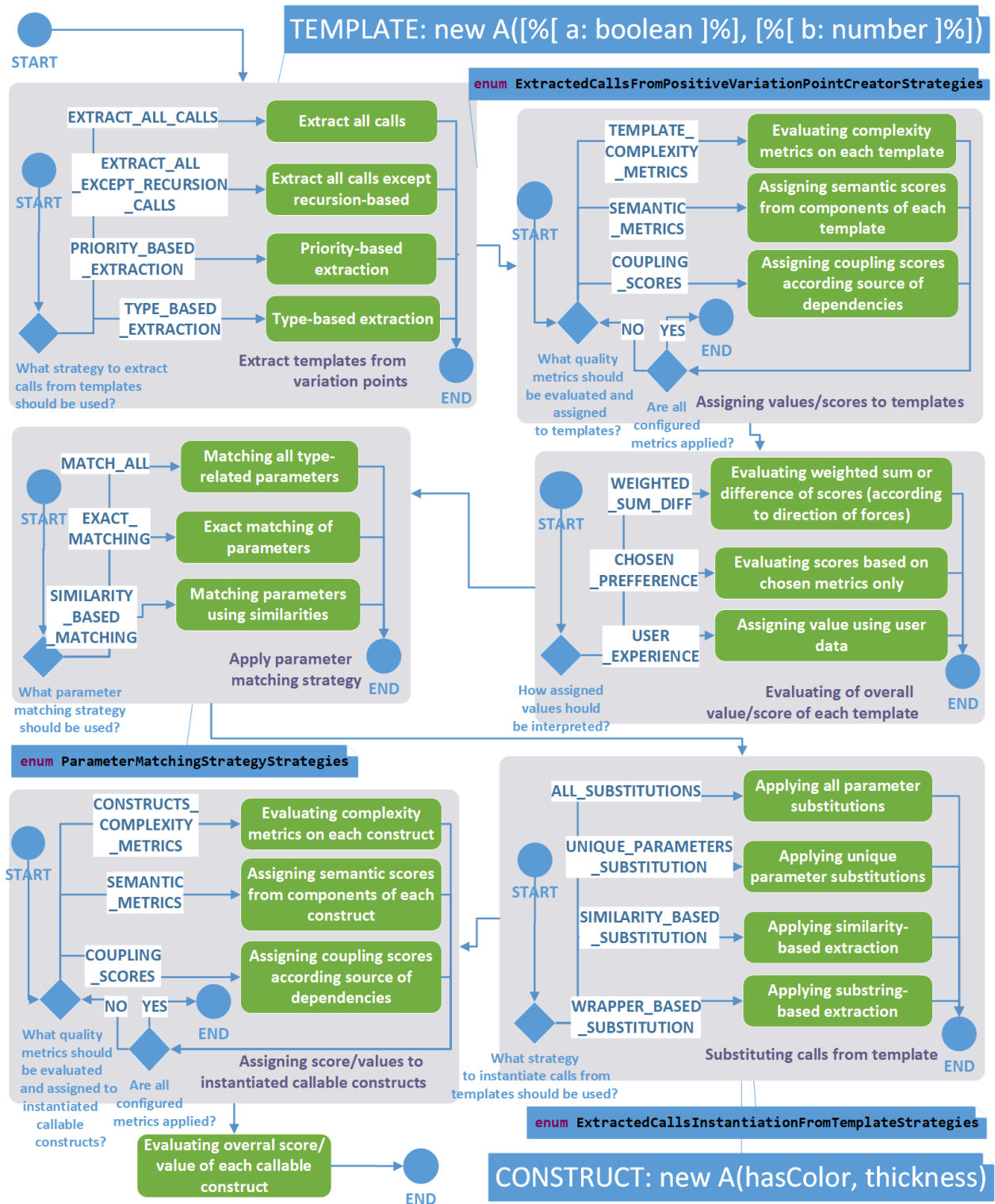


FIGURE 19. Integrating features: Template and construct creation with value assignment.

product lines produced in the actual iteration can be launched and tested immediately. Alternatively, if necessary, errors can be corrected manually. Runnable product is evaluated

using available tools as standard projects. Thus, such evolved software product lines are selected according to measures, user experience, and related data. The selected evolved



The core of our approach’s proposed iteration as part of fully automated evolution is adapted to comply with the code structure, particularly possible variation points. For this purpose, we restrict most automation to the abstraction level, on which the code fragments are processed from standalone

Similarly, some of the proposed and even some later extended strategies may require different parameters. These

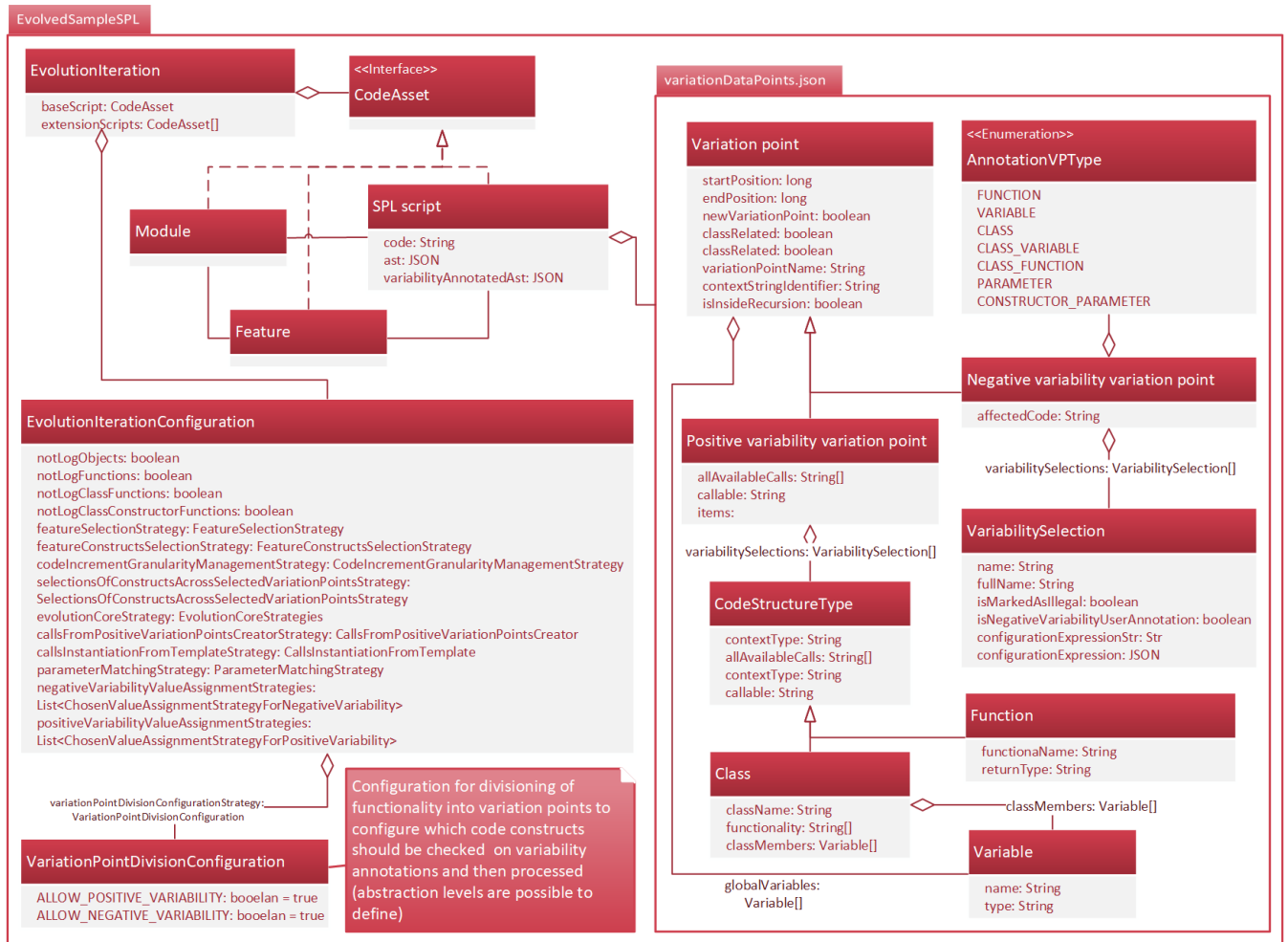


FIGURE 21. The configuration of a particular evolution iteration, including the list with representation of variation points.

adaptations will lead to the extension and further adaptation of these models. Alternatively, the aggregation of functionality that evolves in files into features and then into modules is a task for processes operating under our approach to automatically evolve particular software product line. These decisions should be propagated to experts or specialized models. They should be performed rarely during all evolution iterations. The mapping can be propagated back in the form of a configuration to the proposed data model to select the most appropriate strategies for the current evolution phase. The mapping of these entities according to the software product line meta-model for handling variability and traceability [50] is intended to integrate the evolved functionality during our evolution approach with other disciplines such as project and risk management, scoping, and testing in further application of the introduced framework.

VI. SCALING INTRODUCED EVOLUTION ENCOMPASSING DIVERSE ARTIFACTS CREATION: DEPLOYABLE PROTOTYPE

Full automation is ensured by orchestrating modules responsible for annotating variability, deriving final products,

and producing diverse representations across the software product line evolution process. Consequently, we designed a microservice architecture and followed it while introducing its containerized version, which is displayed in the deployment diagram in Fig. 22. Additional modules help automate side tasks during minimalistic variability handling performed by our framework as a central component (*Evolved SPL Framework Server*) in the evolution of fractal shapes. The introduced orchestration is ensured by docker volumes (*Docker Volumes* node), providing a way how to share data between containerized modules and an asynchronous message queue (*Evolved SPLs Processing Message Queue*) to handle time-consuming tasks, including the creation of diverse representations from evolved software product line instances using our artifact deployed on *Fractal Dynamic Data Collector Server* separately. Extracted graphs from object instances or even semantic triples made out of selected representations can be organized into *Fractal Semantic Base* running on the *Semantic Base Server* node. Another emerging time-consuming task is to incorporate various models capable of selecting software product line instances for the next evolution iteration according to the requested product quality

and customer requirements. Perceived models should cover enforcement learning or fact-based systems. When applying the presented evolution to aesthetic processing, we recommend putting these extensions into new nodes and ensuring communication through the asynchronous message queue.

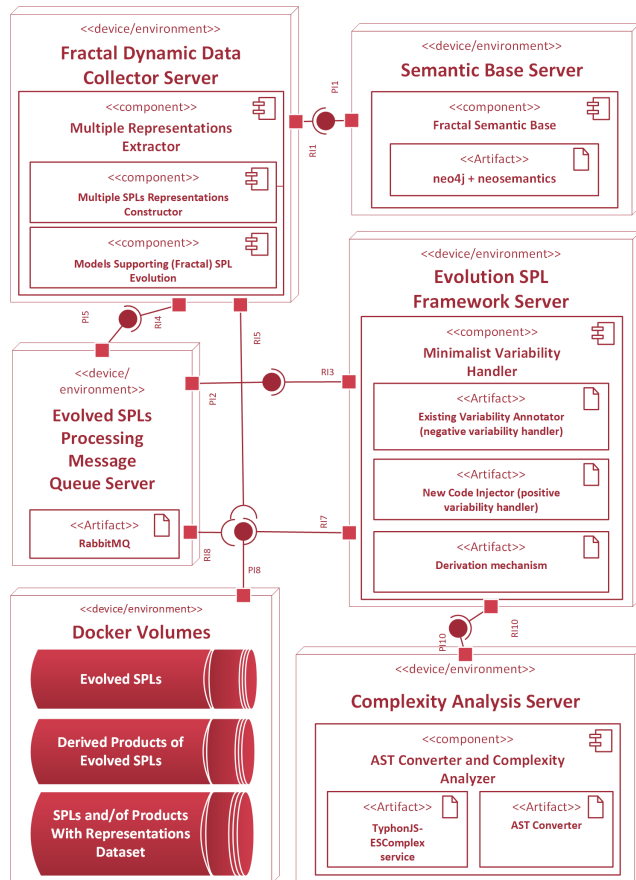


FIGURE 22. Deployment diagram of our fully automated software product line evolution.

The proposed architecture best manages these modules, primarily allowing their implementation in different languages and using technologies for processing big data. For example, converting and processing abstract syntax trees usually requires original libraries that are available only in native programming languages capable of compiling given codes. *NodeJS* running on the *Complexity Analysis Server* with preinstalled TypeScript libraries fulfills this task. The evaluation of code complexity is ensured and exposed through the API by this node. Additionally, data processing requires libraries that support analytical models and tools. We achieved this by exposing the *Flask* API written in Python and using various data-scraping and preprocessing libraries. It is deployed at the *Fractal Dynamic Data Collector Server* node. Another benefit is the reusability of already provided solutions, such as the implementation of an asynchronous queue to scale the solution horizontally. We orchestrate these services using a *docker-compose* that is provided as a separate service. The derivation of new facts can benefit

from logical programming languages that support the first-order logic. However, we mainly focused on the knowledge bases established from diverse representations obtained using *neosemantics* constructs [51] as the *neo4j* extension into the *neo4j* graph database [52] which runs on a separate container. *SPARQL* [53], [54] along with an *Apache Jena* server [55], can extend the existing architecture to manage triples directly in their native formats.

We containerized it using *Docker* [56] following its orchestration in a *docker-compose* [57]. The solution can be easily demonstrated by running only two commands in the directory containing the *docker-compose.yml* file, and *.env* file with environmental variables to easily configure the functionality. The first *pull* command downloads all associated images from the *Docker Hub* repository [58] and *up* command to run all orchestrated containers. On the other hand, the containers can be orchestrated using *Kubernetes* [59]. It allows instantiating the chosen number of worker objects and managing their automated replacement with similar server instances in case of failures.

VII. EVALUATION

We successfully applied our approach to evolve five-sided fractal scripts in three preconfigured evolution iterations. Each successfully extended the script with new code from the exported functionality, with other fractals preserved in separate files. We tested the possibility of concatenating the evolution iterations into a custom pipeline, where each is configured under different settings. Each version of the evolved software product line was successfully rendered in a web browser, demonstrating the smooth execution of all imported scripts. The printing of specific variables in the console returns instances from the stack. Similarly, we checked if the logs were produced if allowed. We automatically extract and synthesize the instances provided from each fractal software product line or its derived product into a graph. In addition, such graphs are merged into an overall graph with the application of aggregations on manually assigned aesthetic scores. The integrated version of the fully automated software product line evolution applied to evolve fractal scripts can now be preconfigured and launched using a *docker-compose*. Three preconfigured iterations are completed within a few tens of minutes, including the creation of diverse representations from these final evolved software product lines. Additionally, we successfully took screenshots automatically using our Python script, opening each solution in a simulated browser.

To evaluate the benefits of the extracted diverse representations for their backpropagation, we annotated 378 fractals (products) with aesthetic scores according to our perception. We used pre-trained models on other images that predict aesthetic scores automatically to account for the effects in automated evolution and compare them with manual ones. We split resulted scores into ten categories according to their aesthetic scores, each covering 10% of the overall range. The initial prediction of such models on our dataset showed

the coverage of various of these aesthetic ranges. Then, we create machine and deep learning models for multinomial classification presented with obtained accuracies from these models for both mentioned ways of aesthetics assignment scores in Table 1. The training set consists of 226 samples, and the remaining 152 were used for testing. An application using graph neural networks [60] proved effective in this task, reaching 71% accuracy in correctly selecting one aesthetic preference from ten aesthetic scores from one user. The results changed if data were automatically generated, whereas the best models were those that generated them. In this case, two convolution layers and one dense layer are used on the output. We based the model on one provided on the web page of used Keras framework [61], [62].

TABLE 1. Comparison of various machine and deep learning models based on diverse data representation.

Used model	Accuracy one user	Accuracy LeNet model [63]
LeNet (input size 28x28)	0,3158	0,8487
LeNet [63] (input size 600x600)	0,3421	0,8618
LeNet [63] multimodel for image with coordinates (input size 600x600)	0,2697	0,7961
Multinomial logistic regression [64] based on coordinates	0,3618	0,8092
Stepwise logistic regression [65] based on coordinates (backward)	0,3421	0,7894
Stepwise logistic regression [65] based on coordinates (forward)	0,4539	0,7960
Stepwise logistic regression [65] based on coordinates (both)	0,4539	0,7961
Graph neural network [61]	0,7133	0,6999

The possibility of comparing various machine and deep learning approaches (such as ordinal models [64] to capture scale, graph deep neural networks [46] to capture dynamics as simulated entity creation and destruction, stepwise classification [65] as a light machine learning method, etc.) on the data from the same product family associated with a particular code fragment is another benefit of this automated application. To the best of our knowledge, we have introduced a new representation consisting of a graph from instantiated entities during program execution, along with images or vectors generated by a specific entity, specifically its code fragment. We then merged these instances from the entire software family according to the similarity of most of the attributes extracted from the entities. Evaluated aesthetic scores must be aggregated or stored in an array. We provide this representation on the input to the graph neural network, the results of which can be found in the last row of Table 1. This representation deeply describes the dynamic behavior of the provided code through instantiated static structures as nodes which are connected in order how they are created. Raster images drawn for each instantiated entity in the program indicate the perceived value for the customer as part of the product. Their vector representation is abstracted from a visual form, such as drawn shapes, to obtain only its descriptive textual representation. It helps to store information about drawn objects together with their parameters, such as the positions of their points. It is possible to preserve these shapes under abstracted entities from the details of their composition. Consequently, the graph contains fewer nodes, and the hierarchical structure

is less deep. Focusing on essential entities that substantially and directly contribute to features can be beneficial. These entities are perceived as patterns, such as star or five-sided object entities, which hold even semantically characteristic attributes. Consequently, our approach can be applied to other, especially stateful, programs.

Injecting particular code fragments into recursively executed code in the case of the evolution of fractal drawings represents the application of a procedural rather than an object-oriented paradigm. The drawn entities are usually represented modularly using functions or objects, or remain as one-line calls. Consequently, we ensured that a particular code is maintained in a similar manner for aggregation into larger entities. On the other hand, the techniques to make various concerns modular are insignificant owing to the low complexity of the extended scripts. In the case of drawing fractals, the introduction of inheritance relationships requires classes to assume the role of the drawn patterns or fractals. Despite creating a slightly different version of the same fractal, combining multiple fractals usually leads to failures because many generated instances reach a deep recursion depth level. Such extensions are proposed for integration in the future.

The processing mechanism used to update one file containing content or export statement from another script tends to miss opportunities while extending specific concerns. For example, scattered features in the code are handled by injecting each of the selected functionalities into some of the multiple variation points in the evolved script, resulting in the processing of a scattered feature across multiple files in multiple iterations during its evolution. The evolution of a particular feature can be scattered throughout the entire evolution, particularly if particular feature is not modularized in a separate file. Advanced modularization techniques must be applied to evolve each feature independently and extend the existing implementation of new features in one place to benefit from separating each variable code fragment during its evolution. These capabilities provide ECAesarJ, where an extension to a particular class includes extended functionality through a mixin propagation composition mechanism [66]. Alternatively, TypeScript/JavaScript provides functionality for prototype-based programming, capable of dynamically extending the existing functionality with new methods and variables in separate files. The separation of such extending functionality is necessary to prevent code scattering and manage the variability of this code fragment as a module in one place. In conclusion, relying on object and aspect orientation is not necessary for fractal drawings but is still important for other programs. The disadvantage of applying ECAesarJ or prototype-based programming lies in the need to update the context of each variation point (the example is shown in the second Listing in Section IV) regarding its extensions, such as the available methods that can be used. In the case of extending content inside a recursive function, the flexible mechanism to inject existing or exported functionality and organize it further showed

more flexibility than advanced techniques for modularization that is dependent on semantics. Consequently, the data must be produced to manage reasoning under semantics and apply such mixin aspects until then. For this purpose, it is feasible to use our approach in the domain of evolving recursive fractal scripts where it is successfully applied. New strategies that support semantics are incorporated into the pipeline without changing the existing code. Processing other software requires more advanced models or relies on the exhausted number of structural updates when the operational data are generated.

We demonstrate the aforementioned prototype-based programming as a future extension to our evolution with its ability to evolve features independently [66] in separate files known as modules in the following JavaScript/TypeScript example. We demonstrate how the existing `VerticePair` class will be extended later in a different script to check whether coordinate `x` does not exceed a particular threshold in JavaScript to demonstrate modularization and extendability. Firstly, `VerticePair` class is defined according to Listing 12.

```
1 class VerticePair {
2   constructor(x, y) {
3     this.x = x;
4     this.y = y;
5   }
6   getX() { return this.x; }
7   getY() { return this.y; }
8 }
```

LISTING 12. Vertice pair class.

Secondly, a subsequently introduced object of `VerticePair` can be instantiated by executing the code from Listing 13.

```
1 var verticePair = new VerticePair(5, 6);
```

LISTING 13. Creating sample vertice pair instance.

Thirdly, the extensions are incorporated into the `VerticePair` class prototype, making them available for all `VerticePair` instances. A demonstrative extension is implemented in Listing 14.

```
1 VerticePair.prototype.checkPoint =
   function() { if (this.x > 2) { throw
   new Error('Coordinate X is greater!')
   ; } }
```

LISTING 14. Extending the class with new method using prototypes.

Such calls should be variability-annotated and moved directly into the initial class. However, as mentioned previously, a better way is to preserve them separately and manage the variability of this introduced module to evolve such features independently, as observed in the AMPLE project [66].

Finally, the extension is tested and proven to successfully incorporate a feature or concern for checking the coordinates what is shown in Listing 15.

```
1 verticePair.checkPoint();
```

LISTING 15. Testing incorporated functionality.

The aspect-oriented CantoJS library was incorporated into the resulting solution to preserve the recently used coordinates. Such functionality does not pollute the code, and stored coordinates help to follow up the drawing at the point where it previously stopped. Aspect orientation helps to reduce the parameters of particular functions, which relieves the cost of their substitution with representatives during the injection of new functionality. Using aspects such as wrappers helps manage the inner state of a drawing that can be accessed inside the incorporated methods provided as exported functionality in separate files on the input.

VIII. DISCUSSION

Introducing decorators as part of the aspect orientation still pollutes the resulting code with variability management constructs [66]. In some languages, such as Python, decorators can be automatically hung to a specific place when the program is loaded. This is useful when the rules for such operations are automatically generated and customized to visualize the included features and exclude unwanted features in a multistage process. The restrictive applicability of annotations only on classes, parameters, class functions, and class variables to handle access and other related functionality should be extended for variability handling, especially to annotate imports and functions. These applications should be restricted to supporting modularity by preserving only the modular fragments.

The evolutionary process is based on various strategies. Consequently, a strategy design pattern is extensively used for this purpose. This process can easily switch to a new strategy. Evolution algorithms [67] can be easily integrated for this purpose, primarily based on a manipulated sequence of variation points during evolution carrying the information about performed changes. The population evolves through crises or mutations owing to this information. Each variation point has a unique location with a similar number of bits. The first bit indicates whether the given variation point is active. The second bit classifies the variation point as common or variable. The third bit helps recognize any variation point inside/as a class. The following bits should represent each capability or operation, such as a list of available calls, where its own bit and sequence number describe each call. Quantitative information, such as scores or metrics, is directly used in the proposed fitness function according to related weights.

Compared to variability handling in systems that require semantic information, the processing semantics is much more complex, requiring the use of knowledge extracted from an additional quantity of data necessary for this purpose. The correctness of such information and exhausted resources are the main problems preventing a fully automated evolution.

For this purpose, we introduced and began observing this simple model. Consequently, it should be used to verify the capabilities of the extracted knowledge within a variability-annotated script and its further use in more complex scenarios. These scenarios demand the application of semantics, for which they must rely on domain experts during their configuration. However, existing approaches tend to remain semi-automated without extensive amounts of data and knowledge.

The evolution becomes stuck when incorrect parameters are used. Time limitations or proper checks of the number of potentially derived features, templates, constructs, and products should be incorporated for this purpose.

Information about similarity based on the value of each instantiated structural-semantic view from variation points can be realized according to our integrated matrix-based approach for structural and semantic analysis, supporting software product line evolution [36] during the joining of another (new) functionality. The incorporation of such functionality benefits from bulk joins. In this case, creating a new evolutionary population is ensured. Simultaneously, the capability of each member is treated according to the external qualities represented as numbers in matrices. They are usually defined by a specified model (for example, according to aesthetic perception [68], [69] and associated factors with direct effects such as a golden cut or correctly capturing of displayed shape [70]), followed by the use of complexity metrics, and finally by manual evaluation.

IX. RELATED WORK

An incremental aspect-oriented product line method for J2ME game development by Alves et al. [71] deals with platform variants by implementing the corresponding variation points by moving specialized behavior into aspects. The evolution of the code lies in the changes applied to the core code fragments when introducing or integrating new functionality. Affected code fragments are shared by one set of products, but not by all of them.

A wide range of real-world software product lines belongs to the annotation-based type [16], which relies on annotating variable code fragments in the code. Many existing versions of Mobile Media [72] and Health Watcher [73] software product lines fall into this category. They have even become research objects in many studies proposing reimplementations to analyze and verify the aspect-oriented paradigm [15], [74], comparison of the object- and aspect-oriented paradigm [15], [75], detecting modularity defects [76], identifying traceable connections among many layers of aspects [77], and even detection of code smells [78]. They even evolved iteratively [15], [72], [74]. By contrast, instantiating and modeling based on statistical analysis using instantiated products or performing various simulations are left aside or entirely intact. These findings have also been observed in the literature [1], [10], [79]. To the best of our knowledge, no study has proposed or analyzed

their automated evolution and domain knowledge extraction possibilities, which were left to developers and domain experts. Additionally, old techniques, such as conditional compilation, are still applied to handle variability [15] owing to a lack of advanced tools and techniques originating from the use of a particular programming language. Consequently, in our previous work [31], we showed that annotations are less complex than wrappers as representative constructs known from conditional compilation but also from other more advanced techniques for variability management, including frame technology, framed aspect, or pure::variants. We propose a minimalistic, fully automated model with diverse data representations to improve the configuration expressions used according to feature models preserved according to their hierarchical structure in code.

Current state-of-the-art techniques, such as feature-oriented modeling and decomposition, evolve features on their own modules separately [66], but this is a different technique than appears in annotation-based software product lines and does not tackle variability in a minimalistic way. In the case of inappropriate decomposition, we perceive some changes or updates across modules as problematic and require domain experts. Modeling of software families with a direct focus on the code itself and its representation for the further processing of artificial intelligence models is another significant outcome that allows the scale incorporation of various models and building knowledge bases restrictively concerning a particular software family as a closely defined area even to control these models. No available technique directs the optimization of configuration expressions according to their position in the code, respective hierarchical relations, and even hierarchic relations to the feature model. Analysis of such an expression in large seems to be a promising research topic that is more focused on how to reveal domain knowledge. It should make it even more suitable for practical applications, primarily to handle problems during feature interaction automatically referred to as problematic from large existing solutions owing to the exponential increase of features caused by incorporated variability [10]. Introduced minimalism allows the incorporation of other techniques and tools that are important in the area of software product lines.

Adaptive systems with autonomic software product lines as one of their important representatives are focused on knowledge gathering and are used to adapt to their goals or to handle sudden problems [5], [6]. Existing studies have identified the complexities caused by domain, cross-domain, and runtime variability [80]. Consequently, the way how to automate and propagate all possible changes, especially in managing a complex system that must be adapted according to the environment [5], [80], is left on the code/implementation level unspecified. We approached this in the minimalistic and automated evolution of small code fragments by diversifying them into representations carrying information about entities of final products, parameters of static structures, the code itself, the authentic look, and

even knowledge connecting them. Domain and cross-domain variability (from the new domain) are handled iteratively in an automated manner by reorganizing or incorporating new code fragments according to the context at the implementation level. For this purpose, we analyzed the existing approaches for managing variability in the code. Owing to automation, we can generate scenarios for runtime variability to adapt the inner components accordingly, even in a dynamic manner. Lightweight variability handling [13] and easy-to-express fractals make the approach more accessible, highlight its minimalism and easy expressiveness in diverse representations, and allow the direct provision and measurement of value for the end user-consumer. Most of these are implemented, provided in a containerized form, and orchestrated.

We even covered all four activities defined by the Taxonomy of variability realization techniques [81] at the implementation level related to the processed software product line scripts. The first one, variability identification, is fulfilled to the extent of ranging up to all possible places in the script where existing entities can be updated or supporting code can be inserted. It also helps to predict runtime variability adaptations. These can then be implemented as context variability [82] to react to the context changes. As a result, we gain the capability to manage code variability completely, but this can be narrowed because of the possibility of applying different strategies. The second one, constraining the variability, is performed directly in place of annotated code fragments, specifically in configuration expressions built hierarchically in accordance with feature models, thereby allowing for their preservation in code. The selection of what is variable and what is left as common is performed using easily substitutable variability-handling strategies. Additionally, variability cannot be entirely or feasibly constrained in the study by Abbas and Andersson [80]. However, we managed this under the constraints of the currently processed software product line script. The third one, variability implementation, is ensured by using an appropriate binding mechanism directly at a selected existing variation point or injected into a selected position in the code. However, this is limited by the capabilities of a particular programming language. The last one, managing variability, is applied in evolved software product lines with automated evolution of off-line path with parallel support of on-line path [83]. The second one is analyzed according to product outcomes evaluated from the proposed machine and deep learning models, such as aesthetics in fractals.

We repeatedly used a strategy pattern to propose replaceable strategies for incorporating variability handling. It allows the implementation and configuration of a specific strategy to fit predefined complexity requirements and fulfill its goals. On the contrary, existing algorithms for feature selections into a resulting product, including Falcon, Jaguar, Snail, or their respective auto-selection based on specified rules [84], are rather used to produce a product that complies with predefined constraints. They did not focus on selecting code constructs to build implementation-level

features or switching features from common to variable and vice versa. Compared with these algorithms, other studies fulfill this objective with nature-inspired computing, such as evolution algorithms [85], [86] or swarm intelligence [85]. However, customers usually select features according to the variability model, especially the feature model, when a particular product is requested to be derived. Consequently, many algorithms for feature selection are redundant or less adaptable to our minimalistic variability handling approach.

Compared with traditional frameworks and approaches to software product lines, such as DOPPLER or Pro-PD [87], our approach is fully automated and supports product derivation. In contrast to Pro-PD and DOPPLER, our process contains a phase in which pre-products are created at the end of each evolution cycle according to the requirements and settings.

These pre-products can be perceived as high-fidelity prototypes and are thus effectively analyzed and provided for customer negotiation. Our process is sequential and based on domain knowledge compared to the aforementioned approaches. In the pre-derivation phase of the Pro-PD, the available requirements are negotiated with customers. Then, the product configuration phase benefits more from the achieved reuse as a basis for following customization of related features ensured by product-specific development. Finally, the product is verified in the previous phase [87]. In the case of DOPPLER, variability models are first created in the configuration preparation phase. The product is configured by choosing variable functionality, and the assets are customized. Customer-specific functionality is then negotiated, followed by additional development. Finally, all assets are integrated into the resulting product [87]. Similarly, our evolution process is applied to code fragments or files at the implementation level and configured according to the observed context to avoid overcoming available resources in an automated manner. We propose to support such observations with data orientation. Therefore, different ways of evolution can be maintained within one project in parallel and potentially merged in the same process. Our approach is still restricted to properly driving evolution phases through a suitable configuration, and is thus complicated if no similarity measures or general rules are available. The existing systems and mechanisms that provide such outputs are required for this purpose. The extraction and organization of multiple representations from a product family can integrate various machine learning algorithms driven by extracted and aggregated data from code fragments, particularly their instantiations, measured complexity, decomposition into graphs, and captured screenshots (if available). Their use is intended to be balanced according to the needs and possibilities of explaining and controlling the overall process.

Dynamic software product lines can benefit from events that incorporate new features or adapt changes to features that occur during the evolution of an entire software product line. When captured, these events and actions can

be used for automated adaptation of dynamic software product lines or to test their adaptability according to these respective events. Consequently, we propose this to support prediction and recovery from unexpected states caused by the ability to bind and rebind variability at runtime [3], [4]. The resulting system implementation in our approach is iteratively modified directly as part of the iteration from the evolution. Consequently, they can be tested and verified as a single-system without deriving any product because of external adaptations and changes (offline activities). Despite these benefits, the configuration of run-time variability for the internal changes (online activities) that are performed by the adaptation logic of the system should be manually adapted, or even evolution rules according to Quinton [88], can be incorporated into our strategies for code synthesis to automate both evolution processes simultaneously, our, and those related to dynamic software product lines [88]. Switching from offline to run-time activities can be easily adaptable and automated, particularly with the help of aspect-oriented programming. Otherwise, in the case of mutually exclusive features, custom products are derived according to the selected negative variability from a given instance after iteration. These approaches also share both sources of uncertainty: lack of domain knowledge and accurate anticipation of run-time variability [5]. An autonomic software product line strategy (ASPL) can manage reuse, complexity mitigation, and uncertainty by separating concerns and asset specialization into product specifics [80]. Compared to this strategy, our approach operates primarily at different abstraction levels, such as the code level, rather than at the platform or process levels. These approaches can be combined to a certain extent. Modularization is strictly restricted to annotable code fragments with decorators that are optionally refined into more complex structures, such as methods and classes. Our approach is independent of development processes and thus allows developers to separate concerns with the help of aspect-oriented programming or design patterns according to the needs of a given product.

In their approach to software reuse, Abbas et al. recorded contextual events by inserting corresponding RDF triples into the ontology. They are then evaluated, and specific actions that comply with the feature model are performed [5]. This can be used to achieve effects similar to those of the proposed approach. However, the main design intention is to use triples to integrate various types of data under a given schema. In our case, the proposed automation captures the reconfiguration effects in the form of data harvested from products that emerge in evolution or directly expresses the diverse aspects of the variability-annotated code. From these, a dataset can be created, updated, and used despite this compared work. The data captured differences among products from the software product line family and even the association to a particular point in the software product line evolution among products from more families. However, the relationship with dynamic software product lines is unnecessary.

X. CONCLUSION AND FUTURE WORK

Various software product line tasks such as design, optimization, and partially also validation are difficult to manage owing to the rapid increase in the number of features. In addition, decision-making under multiple aspects of derived products is more complicated. Therefore, further automation using various simulations is required. Unfortunately, existing solutions usually neglect knowledge modeling and simulation of the interaction between features. Consequently, these solutions miss opportunities to resolve associated and emerging problems, such as defect detection or quality assurance, which can be solved by effectively extracting and utilizing knowledge from data based on the differences between variants. Alternatively, these associated problems can be resolved manually through the participation of fluctuating domain experts and developers. Consequently, we developed an approach that fully automatically and iteratively evolves software product lines based on annotated variability or independent code fragments according to the configuration. They are accompanied by the extraction, collection, and organization of variously represented information from variation points regarding code structure and semantics. The simulated activities are also tracked in the configuration spread across a particular variant of the evolution process and used to drive the simulation of the software product line evolution fully automatically or even autonomously. Variants of simulated evolution processes help to predict unexpected states and edge cases, which is necessary to reduce the effort required to design dynamic software product lines. Additionally, we implemented a framework to support the proposed approach.

A significant benefit of our software product line evolution is the possibility of configuring variability handling through custom strategies, particularly decisions made by selecting and processing variation points. This results in a comprehensive domain modeling performed on real software product lines or their simplified models. Applying different strategies according to the configuration of each evolution iteration allowed the evolution process to be driven towards fulfilling requirements and/or quality improvements. Negative and positive variability are managed according to the configured preferences, and the overall process is tuned based on restrictions according to the criteria of the allowed candidates. Decision-making under evolutionary processes is supported by various models that process data from potential products in many representations, such as graph data, semi-structured data, relational data, vectors, and raster images. Consequently, integrating GNNs or logistic regression is not problematic.

This approach has been applied to the fractal software product line. Its application initially allowed the iterative merging of various scripts without explicitly supporting variability management. Subsequently, extending such scripts resulted in supporting the evolution of numerous software product lines. Optionally, the sequence of representations of

the variation points and updated abstract syntax tree after the performed changes are generated. These outputs will be used further for decision-making and automated constitution of evolution scenarios. The solution can be automatically and repeatedly reconfigured for various tasks, such as quality improvements, changes in common and variable variation points, observing incremental code changes and their impact on software quality, and the possibility of adapting artificial intelligence (to an explicit extent), especially machine learning. Implementing selection strategies and rating assignments is also supported at different abstraction levels.

Another consequence of employing variability management on a large scale is the possibility of formulating configuration expressions in place of variation points. These are primarily responsible for the inclusion of these features in the code. Observations of dependencies from semantics and data structures among configuration parameters with directly associated information from variation points will help optimize these expressions. The user configuration expressions provided in the place of variation points can be compared, improved, or utilized according to the generated ones with the help of various models driven by knowledge and data. However, optimizing these configuration expressions in a fully automated fashion can benefit from the full potential of automated evolution and the ease of introducing new features in large numbers of high-quality aesthetic fractal products. Making each of such configuration expressions carry hierarchical information from the feature model (idea to preserve feature models in code) can help explore the effects on user comprehension of variability in code and propose new approaches to measure the complexity of code belonging to variability and filter it during software development to present it more comprehensively. In a future study, we plan to propose and evaluate various strategies to improve user comprehension during the design of these expressions.

Our future work will focus on determining the best configurations for the evolution of the fractal scripts. Not only their quality but also how frequently it is necessary to switch from traditional evolution to quality improvements and the incorporation of domain knowledge. Traditional evolution involves a synthesis of various fractals and their fragments. On the contrary, quality improvements depend on various metrics evaluated from various diverse artifacts associated with related variation points, particularly on particular code constructs. Similarly, domain knowledge helps increase reuse in a way that can optimally select common and variable features to satisfy the majority of requirements across products from the same family. We test time-series models created from an ordered sequence of variation points across different evolution iterations and update or choose among configuration settings accordingly.

Nevertheless, our approach is capable of challenging other problems associated with the variability. The interaction between features will be presented and visualized in simulations of fractal evolution phases, focusing on modifying their features. Dependencies across various evolution-division

sequences will show how the features depend on each other. Otherwise, the scripts will not be executed. An associated goal will be to develop a detection mechanism to recognize incomplete fractals. This capability will help eliminate divisions that lead to malformed features during evolution. Additionally, the creation of such tools can help predict the best-performing evolution. In the case of dynamic software product lines, various scenarios can be designed and tested for dynamic self-adaptability. Similarly, it allows for testing the effectiveness of partitioning into online and offline software activities.

Future studies can be conducted based on the fact that the variability in the presented approach is found and/or discovered and then iteratively updated directly in the code to fulfill some of the provided requirements. Owing to its bottom-up nature, it can potentially be supported with information, especially knowledge, from other high-layer disciplines, such as project and risk management, scoping, and testing, through the specific configuration of initial settings to run each evolution phase. Fuzzy logic can be employed to tackle interpretation problems about uncertainty, such as “not much-derived products” or “fast evolution phase.” After applying our proposed evolutionary process, specific findings with domain knowledge of the variability configuration will emerge. We expect the opportunity to analyze outcomes from the modeling of emerged variability, which will be based on possible mappings created according to the relations of the software product line metamodel [50] to handle variability and traceability among its entities. The entities of the mentioned software product line meta-model fall into the proposed software product line disciplines.

REFERENCES

- [1] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, “Variability in software systems—A systematic literature review,” *IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 282–306, Mar. 2014.
- [2] K. Pohl, G. Böckle, and F. Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Berlin, Germany: Springer-Verlag, Jan. 2005.
- [3] M. Hinchey, S. Park, and K. Schmid, “Building dynamic software product lines,” *Computer*, vol. 45, no. 10, pp. 22–26, Oct. 2012, doi: [10.1109/MC.2012.332](https://doi.org/10.1109/MC.2012.332).
- [4] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, “Dynamic software product lines,” *IEEE Comput.*, vol. 41, no. 4, pp. 93–95, Apr. 2008, doi: [10.1109/MC.2008.123](https://doi.org/10.1109/MC.2008.123).
- [5] N. Abbas, J. Andersson, and D. Weyns, “ASPL: A methodology to develop self-adaptive software systems with systematic reuse,” *J. Syst. Softw.*, vol. 167, Sep. 2020, Art. no. 110626. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0164121220301047>
- [6] B. H. C. Cheng et al., “Software engineering for self-adaptive systems: A research roadmap,” in *Software Engineering for Self-Adaptive Systems*, vol. 5525. Berlin, Germany: Springer, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, 2009, pp. 1–26, doi: [10.1007/978-3-642-02161-9_1](https://doi.org/10.1007/978-3-642-02161-9_1).
- [7] A. E. Hassan, “The road ahead for mining software repositories,” in *Proc. Frontiers Softw. Maintenance*, Beijing, China, Sep. 2008, pp. 48–57. [Online]. Available: <https://ieeexplore.ieee.org/document/4659248/>
- [8] H. Kagdi, M. L. Collard, and J. I. Maletic, “A survey and taxonomy of approaches for mining software repositories in the context of software evolution,” *J. Softw. Maintenance Evolution: Res. Pract.*, vol. 19, no. 2, pp. 77–131, Mar. 2007, doi: [10.1002/smr.344](https://doi.org/10.1002/smr.344).

- [9] L. Bigliardi, M. Lanza, A. Bacchelli, M. D'Ambros, and A. Mocci, "Quantitatively exploring non-code software artifacts," in *Proc. 14th Int. Conf. Quality Softw.*, Dallas, TX, USA, Oct. 2014, pp. 286–295. [Online]. Available: <https://ieeexplore.ieee.org/document/6958416/>
- [10] L. Chen, M. A. Babar, and N. Alf, "Variability management in software product lines: A systematic review," in *Proc. 13th Int. Softw. Product Line Conf.*, Jan. 2009, pp. 81–90.
- [11] F. Loesch and E. Ploedereder, "Optimization of variability in software product lines," in *Proc. 11th Int. Softw. Product Line Conf. (SPLC)*, Kyoto, Japan, Sep. 2007, pp. 151–162, doi: [10.1109/SPLINE.2007.31](https://doi.org/10.1109/SPLINE.2007.31).
- [12] A. van Deursen, M. de Jonge, and T. Kuipers, "Feature-based product line instantiation using source-level packages," in *Software Product Line*, vol. 2379, G. Goos, J. Hartmanis, J. van Leeuwen, and G. J. Chastek, Eds., Berlin, Germany: Springer, 2002, pp. 217–234.
- [13] J. Perdek and V. Vranic, "Lightweight aspect-oriented software product lines with automated product derivation," in *New Trends in Database and Information Systems*, A. Abelló, P. Vassiliadis, O. Romero, R. Wrembel, F. Bugiotti, J. Gamper, G. Vargas Solar, and E. Zuppano, Eds., Cham, Switzerland: Springer, 2023, pp. 499–510.
- [14] C. Denger and R. Kolb, "Testing and inspecting reusable product line components: First empirical results," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, Sep. 2006, pp. 184–193.
- [15] E. Figueiredo and N. Cacho, "Evolving software product lines with aspects: An empirical study on design stability," in *Proc. 13th Int. Conf. Softw. Eng.*, 2008, p. 261.
- [16] W. Fenske, T. Thüm, and G. Saake, "A taxonomy of software product line reengineering," in *Proc. 8th Int. Workshop Variability Model. Software-Intensive Syst.*, Jan. 2014, pp. 1–8.
- [17] N. Loughran and A. Rashid, "Framed aspects: Supporting variability and configurability for AOP," in *Proc. 8th Int. Conf. Softw. Reuse*, Madrid, Spain: Springer, Jan. 2004, pp. 127–140.
- [18] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek, "Supporting product line evolution with framed aspects," in *Proc. AOSD ACP4IS Workshop*, 2004, p. 5.
- [19] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto, "AOJS: Aspect-oriented Javascript programming framework for web development," in *Proc. 8th Workshop Aspects, Compon., Patterns Infrastructure Softw.*, Charlottesville, VA, USA, Mar. 2009, pp. 31–36, doi: [10.1145/1509276.1509285](https://doi.org/10.1145/1509276.1509285).
- [20] Pure: Systems. (2020). *Plé & Code—Managing Variability in Source Code*. [Online]. Available: <https://youtu.be/RIUYjWhJfK>
- [21] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Generative Programming and Component Engineering* (Lecture Notes in Computer Science), vol. 3676, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. P. Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Glück, and M. Lowry, Eds., Berlin, Germany: Springer, 2005, pp. 422–437.
- [22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," in *Carnegie-Mellon Univ. Software Eng. Inst., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-90-TR-021*, Nov. 1990.
- [23] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A survey of variability modeling in industrial practice," in *Proc. 7th Int. Workshop Variability Model. Software-Intensive Syst.*, Pisa, Italy, Jan. 2013, p. 1, doi: [10.1145/2430502.2430513](https://doi.org/10.1145/2430502.2430513).
- [24] B. B. P. Cafeo, F. Dantas, A. Gurgel, E. Guimarães, E. R. Cirilo, A. Garcia, and C. J. P. Lucena, "Analysing the impact of feature dependency implementation on product line stability: An exploratory study," in *Proc. 26th Brazilian Symp. Softw. Eng.*, Natal, Brazil, Sep. 2012, pp. 141–150, doi: [10.1109/SBES.2012.23](https://doi.org/10.1109/SBES.2012.23).
- [25] T. J. Young and B. Math, "Using AspectJ to build a software product line for mobile devices," in *Proc. 4th Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, Chicago, U.K., 1999, p. 73, doi: [10.1145/1985374.1985387](https://doi.org/10.1145/1985374.1985387). [Online]. Available: <https://scholar.google.com/scholar?q=Young%2C+T.+and+Murphy%2C+G.+2005.+Using+AspectJ+to+Build+a+Product+Line+for+Mobile+Devices.+Proc>
- [26] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ," *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 174–190, Nov. 2002, doi: [10.1145/583854.582437](https://doi.org/10.1145/583854.582437).
- [27] C. Kastner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *Proc. 11th Int. Softw. Product Line Conf. (SPLC)*, Kyoto, Japan, Sep. 2007, pp. 223–232, doi: [10.1109/spline.2007.12](https://doi.org/10.1109/spline.2007.12).
- [28] G. Botterweck, K.-W. Lee, and S. Thiel, "Automating product derivation in software product line engineering," in *Proc. Softw. Eng.*, 2009, p. 143.
- [29] L. Blair and J. Pang, "Aspect-oriented solutions to feature interaction concerns using AspectJ," in *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press, 2003, p. 17.
- [30] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Proc. Workshop Adv. Separat. Concerns Object-Oriented Syst., ACM Conf. Object-Oriented Program., Syst., Lang., Appl.* Minneapolis, MN, USA: RIACS, Jan. 2000, p. 9.
- [31] J. Perdek and V. Vranić, "Complexity of in-code variability: Emergence of detachable decorators," in *Reuse and Software Quality*, A. Achilleos, L. Fuentes, and G. A. Papadopoulos, Eds., Cham, Switzerland: Springer, 2024, pp. 51–71.
- [32] J. Bayer, O. Flège, and C. Gacek, "Creating product line architectures," in *Software Architectures for Product Families*, vol. 1951, G. Goos, J. Hartmanis, J. van Leeuwen, and F. van der Linden, Eds., Berlin, Germany: Springer, 2000, pp. 210–216.
- [33] E. Y. Nakagawa, P. Oliveira Antonino, and M. Becker, "Reference architecture and product line architecture: A subtle but critical difference," in *Software Architecture*, vol. 6903, I. Crnkovic, V. Gruhn, and M. Book, Eds., Berlin, Germany: Springer, 2011, pp. 207–211.
- [34] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory and Practice*. Reading, MA, USA: Addison-Wesley, 2007.
- [35] H. Ahn and S. Kang, "Analysis of software product line architecture representation mechanisms," in *Proc. 9th Int. Conf. Softw. Eng. Res., Manage. Appl.*, Baltimore, MD, USA, Aug. 2011, pp. 219–226.
- [36] J. Perdek and V. Vranic, "Matrix based approach for structural and semantic analysis supporting software product line evolution," in *Proc. 10th Workshop Software Quality Anal., Monitoring, Improvement, Appl.*, B. Zoran, V. Vranic, and J. Lang, Eds., Bratislava, Slovakia: Springer, 2023.
- [37] C. Salinesi and R. Mazo, "Defects in product line models and how to identify them," in *Software Product Line*, A. O. Elfaki, Ed., Rijeka, Croatia: IntechOpen, 2012, sec. 5, doi: [10.5772/35662](https://doi.org/10.5772/35662).
- [38] Marek Vajgl A Zdeňka Telnarová Alena Lukášová Martin Záček, *Formální logika a Sémantický Web*. Plzeň: Západočeská univerzita, 2019.
- [39] J. Cardoso. (Sep. 2021). *How to Use Decorators in Typescript*. [Online]. Available: <https://www.digitalocean.com/community/tutorials/how-to-use-decorators-in-typescript>
- [40] K. Terzidis, *Algorithms for Visual Design Using the Processing Language*. Indianapolis, IN, USA: Wiley, 2009.
- [41] L. Hřebíček, *Vyprávění o lingvistických Experimentech S Textem*. Praha: Academia, 2002. [Online]. Available: <https://books.google.sk/books?id=XcPIAAAAAAAJ>
- [42] D. Flanagan. (2010). *Canto-JS: An Improved API for the HTML <CANVAS> Tag*. [Online]. Available: <https://code.google.com/archive/p/canto-js/>
- [43] M. Leahy. (2018). *Typhonjs-Escomplex: Next Generation Complexity Reporting for JavaScript & Typescript Based on the Babel Parser*. [Online]. Available: <https://github.com/typhonjs-node-escomplex/typhonjs-escomplex?tab=readme-ov-file>
- [44] W. E. Lorenz, J. Andres, and G. Franck, "Fractal aesthetics in architecture," *Appl. Math. Inf. Sci.*, vol. 11, no. 4, pp. 971–981, Jul. 2017, doi: [10.18576/amis/110404](https://doi.org/10.18576/amis/110404).
- [45] W. Yun, X. Zhang, Z. Li, H. Liu, and M. Han, "Knowledge modeling: A survey of processes and techniques," *Int. J. Intell. Syst.*, vol. 36, no. 4, pp. 1686–1720, Apr. 2021.
- [46] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*. OpenReview.Net, 2017, p. 14.
- [47] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, Jan. 2020, doi: [10.1016/j.aiopen.2021.01.001](https://doi.org/10.1016/j.aiopen.2021.01.001).
- [48] T. V. Montshiwa and T. Botlhoko, "Stepwise logistic regression, hierarchical logistic regression, CART and Naïve Bayes for predicting learners' numeracy test results," *Tech. Rep.*, 2022, doi: [10.21203/rs.3.rs-1595992/v1](https://doi.org/10.21203/rs.3.rs-1595992/v1). [Online]. Available: https://www.researchgate.net/publication/372317387_Modeling_and_Predicting_Learners%27_Numeracy_Test_Results_using_Some_Regression_and_Machine_Learning_Classifiers

- [49] T. Zhang, L. Deng, J. Wu, Q. Zhou, and C. Ma, "Some metrics for accessing quality of product line architecture," in *Proc. Int. Conf. Comput. Sci. Softw. Eng.*, Wuhan, China, 2008, pp. 500–503, doi: 10.1109/csse.2008.500.
- [50] R. D. O. Cavalcanti, E. S. de Almeida, and S. R. Meira, "Extending the RIPLE-DE process with quality attribute variability realization," in *Proc. Joint ACM SIGSOFT Conf., QoSA ACM SIGSOFT Symp., ISARCS Quality Softw. Architectures, QoSA Architecting Critical Syst.*, Boulder, CO, USA, 2011, p. 159.
- [51] I. Neo4j. (2024). *Neosemantics*. [Online]. Available: <https://neo4j.com/labs/neosemantics/>
- [52] I. Neo4j. (2024). *Neo4j Graph Database*. [Online]. Available: <https://neo4j.com/>
- [53] B. Ducharme, *Learning SPARQL: Querying and Updating With SPARQL 1.1*, 2nd ed., Sebastopol, CA, USA: O'Reilly Media, 2013.
- [54] W3C. (2007). *SPARQL Query Language for RDF*. [Online]. Available: <https://www.w3.org/TR/rdf-sparql-query/>
- [55] T. A. S. Foundation. (2024). *Apache Jena Releases*. [Online]. Available: <https://jena.apache.org/download/index.cgi>
- [56] I. Docker. (2024). *Docker*. [Online]. Available: <https://www.docker.com/>
- [57] I. Docker. (2024). *Docker Compose*. [Online]. Available: <https://docs.docker.com/reference/cli/docker/compose/>
- [58] I. Docker. (2024). *Docker Hub*. [Online]. Available: <https://hub.docker.com/>
- [59] CNCF. (2024). *Kubernetes*. [Online]. Available: <https://kubernetes.io/>
- [60] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.
- [61] K. Salama. (2022). *Node Classification With Graph Neural Networks*. [Online]. Available: <http://web.archive.org/web/20220323010427/>
- [62] K. Salama. (2024). *Node Classification With Graph Neural Networks*. [Online]. Available: https://keras.io/examples/graph/gnn_citations/
- [63] K. Degila. (2019). *Lenet*. [Online]. Available: <https://github.com/kayveen/pyimagesearch/blob/master/nn/conv/lenet.py>
- [64] J. Bruin. (Feb. 2011). *Newtest: Command to Compute New Test @Online*. [Online]. Available: <https://stats.oarc.ucla.edu/stata/ado/analysis/>
- [65] S. Steven. (2023). *A Complete Guide to Stepwise Regression in R*. [Online]. Available: <https://www.r-bloggers.com/2023/12/a-complete-guide-to-stepwise-regression-in-r/>
- [66] A. Rashid, J.-C. Royer, and A. Rummler, *Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way*, 1st ed., New York, NY, USA: Cambridge Univ. Press, 2011.
- [67] X. Yu, *Introduction to Evolutionary Algorithms*, vol. 9. Seoul, South Korea: Korean Institute of Industrial Engineers, Dec. 2010, p. 1, doi: 10.1109/ICCIE.2010.5668407.
- [68] L. Mai, H. Jin, and F. Liu, "Composition-preserving deep photo aesthetics assessment," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA, Jun. 2016, pp. 497–506.
- [69] N. Ponomarenko, L. Jin, O. Ieremeiev, V. Lukin, K. Egiazarian, J. Astola, B. Vozel, K. Chehdi, M. Carli, F. Battisti, and C.-C. Jay Kuo, "Image database TID2013: Peculiarities, results and perspectives," *Signal Process., Image Commun.*, vol. 30, pp. 57–77, Jan. 2015.
- [70] B. Zhang, L. Niu, and L. Zhang, "Image composition assessment with saliency-augmented multi-pattern pooling," Oct. 2021, *arXiv:2104.03133*.
- [71] V. Alves, P. M. Jr., and P. Borba, "An incremental aspect-oriented product line method for J2ME game development," in *Proc. Workshop Managing Variability Consistently Design Code*, Jan. 2004, p. 3.
- [72] L. P. Tizzei. (2012). *Mobilemedia-Cosmos-Vp-V7*. [Online]. Available: <https://github.com/leotizzei/MobileMedia-Cosmos-VP-v7>
- [73] S. Soares. (2011). *Health Watcher Common Zip*. [Online]. Available: <https://www.cin.ufpe.br/scbs/testbed/implementation/HW/HealthWatcherCommon.zip>
- [74] S. Soares. (2011). *Tao: A Testbed for Aspect Oriented Software Development*. [Online]. Available: <https://www.cin.ufpe.br/>
- [75] J. Fabry, C. De Roover, C. Noguera, S. Zschaler, A. Rashid, and V. Jonckers, "AspectJ code analysis and verification with GASR," *J. Syst. Softw.*, vol. 117, pp. 528–544, Jul. 2016.
- [76] H. Cherait, N. Bounour, and L. Laboratory, "History-based approach for detecting modularity defects in aspect oriented software," *Informatica*, vol. 39, no. 2, pp. 187–194, Jun. 2015.
- [77] A. Sardinha, Y. Yu, N. Niu, and A. Rashid, "EA-tracer: Identifying traceability links between code aspects and early aspects," in *Proc. 27th Annu. ACM Symp. Appl. Comput.*, Trento, Italy, Mar. 2012, pp. 1035–1042, doi: 10.1145/2245276.2231938.
- [78] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *J. Softw. Eng. Res. Develop.*, vol. 5, no. 1, p. 7, Dec. 2017.
- [79] M. Marques, J. Simmonds, P. O. Rossel, and M. C. Bastarrica, "Software product line evolution: A systematic literature review," *Inf. Softw. Technol.*, vol. 105, pp. 190–208, Jan. 2019. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0950584918301848>
- [80] N. Abbas and J. Andersson, "Harnessing variability in product-lines of self-adaptive software systems," in *Proc. 19th Int. Conf. Softw. Product Line*, Jul. 2015, pp. 191–200, doi: 10.1145/2791060.2791089.
- [81] M. Svahnberg, J. van Gorp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Pract. Exper.*, vol. 35, no. 8, pp. 705–754, Jul. 2005, doi: 10.1002/spe.652.
- [82] R. Capilla, Ó. Ortiz, and M. Hinchey, "Context variability for context-aware systems," *Computer*, vol. 47, no. 2, pp. 85–87, Feb. 2014. [Online]. Available: <http://ieeexplore.ieee.org/document/6756724/>
- [83] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi, and T. Vogel, "Software engineering processes for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, vol. 7475, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds., Berlin, Germany: Springer, 2013, pp. 51–75, doi: 10.1007/978-3-642-35813-5_3.
- [84] F. Q. Khan, S. Musa, G. Tsaramiris, and S. M. Buhari, "SPL features quantification and selection based on multiple multi-level objectives," *Appl. Sci.*, vol. 9, no. 11, p. 2212, May 2019, doi: 10.3390/app9112212. [Online]. Available: <https://www.mdpi.com/2076-3417/9/11/2212>
- [85] A. A. Mamun, F. Djatmiko, and M. K. Das, "Binary multi-objective PSO and GA for adding new features into an existing product line," in *Proc. 19th Int. Conf. Comput. Inf. Technol. (ICCIT)*, Dec. 2016, pp. 581–585.
- [86] J. Guo, J. White, G. Wang, J. Li, and Y. Wang, "A genetic algorithm for optimized feature selection with resource constraints in software product lines," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2208–2221, Dec. 2011, doi: 10.1016/j.jss.2011.06.026. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121211001518>
- [87] R. Rabiser, P. O'Leary, and I. Richardson, "Key activities for product derivation in software product lines," *J. Syst. Softw.*, vol. 84, no. 2, pp. 285–300, Feb. 2011.
- [88] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schuhmayer, "Evolution in dynamic software product lines," *J. Softw., Evol. Process*, vol. 33, no. 2, p. 2293, Feb. 2021, doi: 10.1002/smr.2293.



JAKUB PERDEK received the M.Sc. degree in computer science from the Slovak University of Technology in Bratislava, in 2022, where he is currently pursuing the Ph.D. degree in software engineering.

Currently, he is analyzing how to optimize configuration expressions used for variability handling towards comprehensive in-code representation of feature models. His research interest includes software product lines, especially their variability management.



VALENTINO VRANIĆ is currently working as a Professor of computer science with Pan-European University, Bratislava, Slovakia. His scientific focus is on the methodology of software development with a particular interest in interconnecting realizational (code) and organizational (people) perspective of software, especially through patterns. He published close to a 100 of scientific papers. He is a member and the Director of the Hillside Board of Directors.

...