Abstract—To cope with project constraints, copying and customizing existing software products is a typical practice to flexibly serve customer-specific needs. In the long term, this practice becomes a limitation for growth due to redundant maintenance efforts or wasted synergy and cross selling potentials. To mitigate this limitation, customized copies need to be consolidated into a single, variable code base of a software product line (SPL). However, consolidation is tedious as one must identify and correlate differences between the copies to design future variability. For one, existing consolidation approaches lack support of the implementation level. In addition, approaches in the fields of difference analysis and feature detection are not sufficiently integrated for finding relationships between code modifications. In this paper, we present an approach to this problem by integrating a difference analysis with a program dependency analysis based on Program Dependency Graphs (PDG) to reduce the effort of consolidating developers when identifying dependent differences and deriving clusters to consider in their variability design. We successfully evaluated our approach on variants of the open source ArgoUML modeling tool, reducing the manual review effort about 72% with a precision of 99% and a recall of 80%. We further proved its industrial applicability in a case study on a commercial relationship management application.

I. INTRODUCTION

Almost all vendors of individual software need to adapt their products for customer specific needs in a flexible and timely manner. Copying an existing implementation and customizing it independently provides a reasonable approach to cope with this challenge in the short term [1]. However, in the long term, such product copies become a limitation for growth as they require parallel maintenance and do not permit selling their individual features in new combinations. A software product line (SPL) [2,3] approach with a single code base and managed variability offers remedy for this problem. Introducing such an approach in retrospect requires consolidating the customized copies and implementing variability mechanisms to flexibly select features. However, such a consolidation is a challenging task as developers consolidating the code are typically not the ones who implemented the customizations. Especially finding code modifications contributing to the same feature within a complete single code base but do not identify copy-specific features, which manifest in the differences only. Finally, existing approaches for the consolidation of customized copies into an SPL are either conceptual [10], fixed to a specific variability mechanism [11], do not target or support the implementation level [12]–[15], only mention its necessity [16,17], or are limited to a coarse grain level [18,19].

In this paper, we present an approach for identifying related modifications between customized code copies based on program dependencies as represented in Program Dependency Graphs [20] (PDG). Many approaches analyzing PDGs exist for different purposes such as change impact analysis [21] and feature location [8,9]. Accordingly, PDGs exist in many different flavors such as the extension for higher program structures proposed by Horwitz [22]. Their value in the field of feature location within a single code base has been proven [23,24] and especially Robillard et al. [24] propose a set of dependencies to study in object-oriented systems. Furthermore, Robillard’s approach has already been used in the context of copy consolidations by Alves et al. [11] (see Section V).

The contributions of this paper are i) a specific set of dependencies to analyze in the context of consolidation, ii) an algorithm specific to code differences combining program dependencies from multiple PDGs, and iii) an approach for deriving variability design recommendations for developers.

For the set of program dependencies in particular, we propose an extension of the set suggested by Robillard et al. [24] and their adaptation for analyzing differences between copied code bases. Within our approach, we focus on analyzing applications based on Java similarly to their work.

In two evaluation scenarios on the ArgoUML tool and a commercial relationship management application,
we examined recall, precision, benefit, and industrial applicability of our approach.

The rest of the paper is structured as follows: Section II introduces the consolidation context the proposed analysis is used in. Section III describes the analysis and how aggregations are derived. Section IV presents a summary of our evaluation scenarios. Section V discusses related approaches and Section VI concludes the paper and provides an outlook on future work.

II. CONSOLIDATION PROCESS CONTEXT

A copy consolidation process requires i) detecting differences, ii) designing variability, and iii) refactoring the actual code as targeted by our overall SPL-evo\(^2\) approach [25,26]. Designing variability requires identifying copy-specific code contributing to the same custom feature and deciding about its representation in the future SPL. Our program dependency analysis aims at supporting consolidating developers in this task by providing reasonable design recommendations.

To enable the consolidation process, we have developed a Variation Point Model (VPM) describing variability on a software design level (Fig. 1). Similar to Jacobson et al. [27], we define a Variation Point (VP) as a concrete location of variability. In contrast to them, we define a VP as a single location of variability and an explicit Variation Point Group (VPG) containing related VPs contributing to the same feature. Finally, we define a Variant (V) as a specific implementation for a VP. Similar to Svalnung et al. [28], we distinguish VPs on the software design level and features on a product management level. Thus, Variation Point Groups can reference features implemented at their contained Variation Points, and Vars can reference respective child features describing a feature’s available options. Furthermore, a Variation Point’s location—only one as mentioned above—and a Variant’s implementation are specified by Software Elements as an abstraction for wrappers referencing elements of concrete software models (e.g., nodes of an abstract syntax tree).

![Variation Point Model](image1)

As part of the consolidation process, we derive an initial Variation Point Model from the differences detected in the first step. Within the variability design step, the Variation Points and Groups are refined. When the consolidating developers are satisfied with the structure and have specified the intended Variation Point characteristics (i.e., binding time, variability type, extensibility, and variability mechanism), the model is transferred to the refactoring step.

The initial Variation Point Model contains a separate Variation Point for each difference. Each initial Variation Point is stored in a separate Group as no relationships have been identified yet. Following, the consolidating developers identify and aggregate Variation Points contributing to the same feature that have to be configured consistently in the future. Aggregating Variation Points is done in two manners: Grouping and

![Fig. 2. Variation Point Aggregations](image2)

Merging (Fig. 2). Grouping represents a logical connection and joins the Variation Points’ groups into a single one. Merging is more sophisticated as it truly melts the Variation Points. If a Variant—identified by its id—exists in more than one of the melted Variation Points, only one Variant element for each id will remain with references to the total quantity of implementing elements. In general, merging is preferable as it offers a greater reduction of the model’s complexity. However, merging several VPs into a single VP means that only one variability mechanism may be implemented at the single location identified by the VP. Thus, merging is limited as not all related software elements can be merged with each other, for example due to their location (e.g., a method call and the according method). If VPs cannot be merged, they will still be grouped as already recommended before the merge detection.

III. VARIATION POINT RELATIONSHIP ANALYSIS

Designing features of an SPL is affected by many soft factors, such as organizational reasons or product management decisions [2] and, thus, cannot be fully automated. However, when consolidating copies and their already implemented custom features, there are given technical relationships between modifications (i.e., Variation Points) one must consider to not perform an implementation from scratch.

To cope with this challenge, we propose a program dependency analysis specialized for dependencies between Variation Points. In particular, this means i) consider code of more than one code base, ii) focus on dependencies between modified software elements, and iii) support groups of software elements in case of previously merged Variation Points. In the following sub-sections, we present our set of considered dependencies, describe our algorithm to find them between Variation Points, and outline how we derive recommendations for developers regarding aggregations.

A. Program Dependencies

Robillard et al. [24] recommend studying dependencies between Classes (C), Fields (F), and Methods (M). We explicitly distinguish Method Signatures (M) and Statements (S) implementing a method’s body. Furthermore, we study dependencies involving Parameters (P), Variables (V), Interfaces

---

\(^2\)http://www.splevo.org
The following, we use the term type to refer to class, interface, or enumeration.

In addition to the types of elements to study, Robillard recommends a set of dependencies to consider: `superType` represents inheritance relationships, `calls` represents functional invocations, and `reads` means accessing the value of another element. Furthermore, `writes` means replacing the value of another element and `creates` means instantiating a new instance of a type. Furthermore, Robillard defines a `declares` dependency between container and child elements (e.g., a class declares a field). This dependency is redundant when analyzing dependencies between differences of software models as a declared element (e.g., a field) is also contained by the declaring element (e.g., a class). Furthermore, if a containing element differs, its children are always assumed as differing, too. Accordingly, there is no value for identifying relationships in analyzing dependencies between declaring and declared software elements. In addition to these dependencies, we introduce `typed` if an element is declared with a specific type. Furthermore, we define `import` relating an element to an import declaration of a required type. Finally, we propose a `modifies` dependency: In object oriented languages, if a variable `v` references an object and one of the object’s method is called or fields is changed, variable `v` still references the same object (e.g., for `myVar.doSth()`, `myVar` still references the same object). Such dependencies are referenced as `modifies` in contrast to the `writes` dependency, which indicates a full replacement of the reference.

Table I summarizes the dependencies considered by our analysis. Each cell represents the dependency of the element in the column’s header linked by the dependency type in the row’s leftmost cell with the element in the row’s second cell. For example, the top right cell represents the dependency “`Interface I` superType of `Class C`”. Not all combinations are reasonable to consider, such as “Class superType Interface”. Dependencies proposed by Robillard are marked with an `R` and our additional ones are marked with an `E`. All of them are considered by our analysis.

![Table](image)

Our analysis considers direct dependencies only. Considering complete forward or backward program slices derived from program dependency graphs might provide additional relationships. However, it is part of our future work and is not yet clear to which degree full or partial slices provide additional value as they are harder to establish and probably lead to an increase in false positive relationships.

### B. Analysis Algorithm

We have developed an algorithm to cope with the three requirements of i) handling multiple code bases, ii) focusing on differences, and iii) supporting variation points with variants implemented by several software elements.

As illustrated in Fig. 3, the algorithm has access to the complete software models of the copies’ code bases. Those models are trees of software elements according to the elements’ containment references. In addition to the containment references, the software models contain cross-references identifying dependencies between software elements not contained by each other. The dependencies studied by our analysis are represented as such cross-references as well. Each variant element references one or more elements in the software models as its implementing elements. Such implementing elements can represent the root of a complete sub-tree containing further software elements.

Initially, the algorithm receives an VPM as input (i.e., step 0). In step 1, the algorithm marks sub graphs of the Variation Points, their Variants and the referenced implementing software elements. In step 2, the sub-graphs are extended to the software elements contained by the implementing elements. In step 3, sub-graphs are further extended by the cross-references representing studied dependencies. In step 4, the resulting sub-graphs are illustrated. Finally, in step 5, all Variation Points that are part of the same sub-graph are considered as related to each other and respective aggregation recommendations are derived.

By default, it is recommended to have aggregations in the form of Groupings. As described in Section II, Groupings can be applied to any set of Variation Points as they represent mere links and the Variation Points remain untouched. In contrast, Merging comes with technical restrictions and requires additional effort to ensure it can be applied. To reduce this effort for developers, we have devised a merge detection that transforms Groupings, either fully or partially, into Mergings if technically possible. A partial transformation has to be used if only a subset of a Grouping’s Variation Points can be merged. In this case, an adequate Sub-Merging will be recommended and the resulting Variation Point will be included in the new Group.

### IV. Evaluation

We have implemented our approach in the publicly available SPLv tooling and evaluated it on variants of the open source UML modeling tool ArgoUML and an industrial use case examining four characteristics:

- **Recall**: To which degree can we aggregate code modifications contributing to the same feature?
- **Precision**: Do all recommended aggregations belong to the same feature?
- **Benefit**: To which degree can we reduce effort of developers in terms of VPGs having to be reviewed for connections?
- **Industrial Applicability**: Can the approach be applied on copies evolved for several years?
We assessed our findings with code annotations published by Couto et al. [29]. They have annotated the code of 8 ArgoUML features with pre-processor statements to permit generation of variants of the tool with desired feature-sets.

To assess the recall, we generated single-feature variants of ArgoUML and applied our approach on each of them compared to a basic variant with no features enabled. Thus, the differences of each comparison belong to a distinct feature and we define the recall as the number of Variation Points we reached a recall of 80% in average. In contrast, analyzing Robillard’s dependencies led to a significantly lower recall of 33% in average.

To assess the precision, we applied our approach to an ArgoUML variant with all features enabled compared to one with no features enabled. We defined the precision as the relation of valid aggregations to the total number of aggregations returned. In [29], Couto identified code intentionally shared by multiple features (i.e., tangling features). Hence, valid aggregations contain either distinct or tangling features. Our analysis returned 216 aggregations in total, with 213 valid aggregations containing either distinct or tangling features. Reviewing the 3 invalid aggregations revealed that they all result from multiple modifications of if-else chain conditions, which are possible with pre-processor statements but are cascaded in parallel code histories. However, their argument is in the general context of feature extraction for consolidation. However, they focus on a downstream feature model refactoring. Furthermore, they establish the need for handling legacy code in their later work [17] similarly to Schütz [16] as part of his general process description. Nunes et al. [32] analyze parallel code bases to improve degraded SPLs. However, they require an initial SPL model as well as parallel code histories. Both these assumptions are too strong for a consolidation. Rubin et al. [33] argue for the value of analyzing the differences between code copies. However, their argument is in the general context of improving feature location techniques and is not specific to a consolidation as in our approach.

We further proved industrial applicability of our approach on copy-based customized components of an industrial relationship management application that were created and evolved since the year 2009. We analyzed the copies and had developers of the company review our recommended aggregations. They reported a satisfying precision of 100% as all aggregated Variation Points were connected to each other. The recall could not be calculated as the copies have not been consolidated yet and the respective design decisions are not available for comparison. However, we already identified related Variation Points without program dependencies and derived directions of future research as summarized in Section VI.

V. RELATED WORK

Approaches related to the one presented in this paper can be distinguished into 3 areas: Handling of customized product copies, code merging, and feature location techniques.

To handle customized product copies, several approaches for SPL feature reconstruction have been developed analyzing architectural information [12]–[14,30] or requirements documentation [15]. However, they lack support for the implementation level. For example, Koschke et al. [30] recommend mapping custom code to module specifications. They provide a structured process but do not reduce the actual effort for code comprehension. Ziadi [18] proposes support in this direction, which, however, is limited to a coarse grain code level. Al-Msie’Deen et al. [31] propose an approach based on lexical and structural similarity—a direction we will also investigate in our future work. Rubin et al. [10,19] propose a “merge-in” algorithm that, however, is limited to UML models and does not consider variability mechanisms. Alves et al. [11] mention Robillard’s [24] dependency analysis in context of manual aspect extraction for consolidation. However, they focus on a downstream feature model refactoring. Furthermore, they explain the need for handling legacy code in their later work [17] similarly to Schütz [16] as part of his general process description. Nunes et al. [32] analyze parallel code bases to improve degraded SPLs. However, they require an initial SPL model as well as parallel code histories. Both these assumptions are too strong for a consolidation. Rubin et al. [33] argue for the value of analyzing the differences between code copies. However, their argument is in the general context of improving feature location techniques and is not specific to consolidation as in our approach.
In the area of software merging—including difference analysis—many approaches exist as surveyed by Mens [7]. However, none of them is about merging differences as optional features. Difference analysis approaches as proposed by Fluri [4], Apiwattanapong [5], and Maletic [6], provide enhanced difference results. However, they have a focus on consecutive versions of a single code base in order to analyze the code’s evolution. Similarly, approaches as recommended by Wu et al. [34], focus on the evolution of a single code base in order to derive guidance for adapting to programming interface changes. In the area of merging, approaches such as the one of Hunt [35], aim for optimized language-aware merging but without introducing additional variability.

VI. CONCLUSION & OUTLOOK

To cope with the challenge of consolidating customized product copies in an SPL with a single, variable code base, we presented an analysis to reduce the consolidating developers’ effort for designing variability of a future SPL. We described our set of analyzed dependencies as well as our algorithm for analyzing relationships between the differences of the copies’ implementations. Finally, we presented an evaluation regarding the recall, precision, benefit, and industrial applicability of our approach.

In our future work, we will study the analysis for further types of relationships (e.g., shared terms [36] and similar modifications [31]) to cope with the Variation Point aggregations not manifested as program dependencies. We will also work on the overall consolidation process for i) automated refactorings introducing variability mechanisms, ii) improvements for iterative consolidation processes, and iii) user studies to identify intuitive integrations of consolidation tasks into state-of-the-art development environments.

Acknowledgment This work was supported by the German Federal Ministry of Education and Research (BMBF), grant No. 01IS13023 A-C, as well as the ESF (European Social Fund) and the Federal State of Saxony within project VICCI #100098171.

REFERENCES