A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies

Benjamin Klatt, Martin Küster, Klaus Krogmann
FZI Research Center for Information Technology
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
{*klatt,kuester,krogmann*}*@fzi.de*

***Abstract*—Software product lines are a widely accepted strategy to develop software products with variable features. Deriving a product line from existing customised product copies is still an open issue. Existing approaches try to extract encapsulated reusable components, or analyse features on a product management level. However, support for understanding implementation differences and designing variation points to consolidate the customised copies is still missing. In this paper, we present the analysis concept of our SPLevo approach, to identify related variation points in order to recommend reasonable variation point designs. To illustrate its application, we provide an example analysis.**

***Keywords*-Software Product Line, Reverse Engineering, Software Analysis, Product Copies**

I. Introduction

Productivity, reuse and customisation for individual requirements are major goals in software engineering. The software product line approach has been established to target these goals with explicit and managed variability [1]. A company can serve all their customers in a specific domain with a variable core software product, which can be configured, extended, or parametrised for the customer-specific needs. Building solutions on such a product line aims to deliver a better tested product with less development and maintenance effort in the long term.

Ideally, the required variability of a software product line is identified as part of the requirements engineering process, and suitable variability realisation techniques are chosen during the software design phase [2]. Such a proactive approach allows to benefit from the product line advantages right from the beginning. However, in reality, it requires high upfront investments, and postpones the first product delivery.

As a result, many companies start implementing the first product and copy and customise this code base afterwards for specific needs of other customers. After better understanding the domain and their customers' needs, they have to reactively transform those variants into a common product line [2]. This procedure ensures, that only mature, tested and actually needed features are integrated into the common product line, but is a challenge by itself.

Today, most existing approaches focus on forward engineering or treat variability without respecting the product life cycle at all. Many approaches exist for varibility specification as surveyed by Chen et al. [3] and different variability realisation techniques are available as surveyed by Patzke et al. [4] and Svahnberg et al. [5]. But, support for reactively migrating customised product variants into a common product line is still an open issue. Such a consolidation process is challenging as the number of differences between two product variants is typically high. A product line engineer, responsible for a consolidation, needs to find the product copies' differences, identify the ones relevant for the product line variability, understand their relations, and design proper variation points.

Only a few existing approaches for reactively build product lines try to handle the challenge and automation of variability reverse engineering. The existing approaches focus on a limited scope of the implementations under study or aim to extract encapsulated, reusable assets of the software. For example, Graaf et al. [6] analyse execution traces of product variants, which can only identify differences in the scope of their executed functionality. Koleilat et al. [7] facilitated clone detection to extract reusable assets. None of them considers the product variants as a whole.

In this paper, we present our variability analysis concept developed as part of our overall approach to consolidate customised product copies into a common software product line (SPLevo [8]). We use a graph-based representation of variation points to combine several basic relationship analysis strategies. We further derive design recommendations from the identified variation point relationships to support the product line engineer in his design task.

The contributions of this paper are i) a graph-based composite analysis to recommend variation point aggregations, ii) a set of basic analysis strategies to identify variation point relationships, and iii) a rule-based concept to derive recommendations from the variation point relationship graph.

The rest of this paper is structured as follows: Section II provides a short overview of the SPLevo approach incorporating the outlined concept. Section III introduces our model for describing variation points followed by our concept of how to analyse instances of this model in Section IV. Section V describes the basic variation point relationship analysis strategies, before an illustrating example of their application is given in Section VI. Section VII presents work related to our approach. In Section VIII we clarify our assumptions and limitations and give a conclusion of our work presented in this paper and an outlook on our future work in Section IX.

II. SPLEVO APPROACH OVERVIEW

The SPLevo approach aims to support product line engineers in consolidating customised product copies into a common product line [9]. The core idea is a semi-automatic, model-driven process to identify the differences between product variants and to guide the product line engineer in iteratively creating a proper variation point design to later derive refactoring support to perform the product line consolidation [10]. The large amount of fine-grained, source-code-level differences to consider and to assess, makes support for finding relevant and related differences necessary.

Figure 1 presents the main process defined in our SPLevo approach. First, software entity models (i.e. abstract syntax trees (AST) [11] and inventory models [12]) are extracted from the product copies' implementations. Those models are compared in the second step to identify the implementation differences. Next, we initialise a model describing fine-grained variation points based on these differences. This model is then analysed ("variation point analysis") to guide the product line engineer in iteratively refining the variation points until he is satisfied with the variability design. As a last step, refactoring support to change the implemention is derived (e.g. insert "#ifdefs" or change to dependency injection).

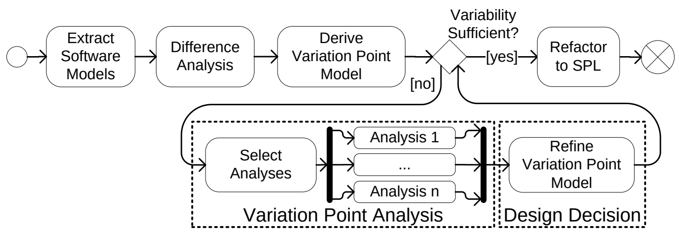

Figure 1. SPLevo Main Process

Such a consolidation will always require manual decisions because of organisational issues or personal preferences on equivalent alternatives in a typically large design space (e.g. because of available implementation techniques, code structuring, or feature granularity). The SPLevo approach supports automated variability reverse engineering, but expects the product line engineer to accept, decline or modify the recommended variation point refinements and to provide individual product line preferences.

III. VARIATION POINT MODEL

Similar to Svahnberg et al. [5], we explicitly distinguish between features on the product management level and variation points on the software design level [10].

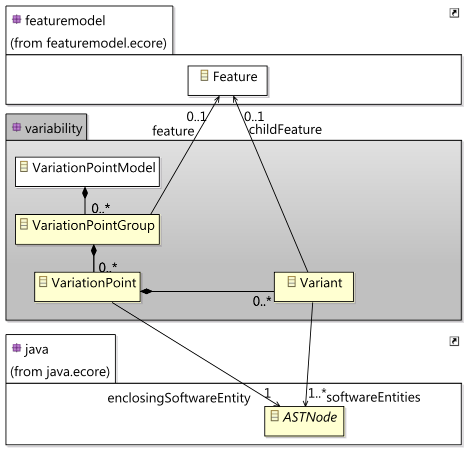

Figure 2. Variation Point Meta-Model (attributes omitted for simplicity)

Figure 2 presents the meta model of the Variation Point Model defined in the SPLevo approach. It is a software design model to describe explicit variation points in a software product line. It allows to correlate and aggregate fine-grained differences into more coarse-grained and better manageable ones.

A *VariantionPoint* represents a location in the software at which a variability mechanism should allow to switch between alternative implementations. The location of the *VariationPoint* is specified by it's reference to an *ASTNode* element of an abstract syntax tree model ("'java.ecore"' in Figure 2). The alternative implementations, available for a *VariationPoint*, are described by it's *Variant* elements. Each of them references one or more software entities (e.g. classes, methods, statements) that implement this specific alternative.

As identified by Svahnberg et al. [5], a single feature can be implemented by one or more variation points in the software. This is reflected in the Variation Point Model by the *VariationPointGroup* element. It groups all *VariationPoints* that might be located at different *ASTNodes* and links to the variable *Feature* they contribute to. In feature models, a variable *Feature* is linked with it's optional or alternative child features. For this, *Variant* elements reference the child feature they contribute to.

The *VariationPointModel* element represents the root element of the model. It contains all *VariationPointGroups* of related *VariationPoints*.

In Figure 2, the referenced *Feature* element origins from the EMF feature model [13], and the *ASTNode* element from the Eclipse MoDisco Java AST model [14]. Both models are used in the currently developed prototype of our approach [8], but the concept itself is not restricted to these concrete feature and software entity models.

As described in the previous section, an initial Variation Point Model is derived from the fine-grained differences between the software entity models. This is done by creating a *VariationPoint* element for each difference with a reference to the parent ASTNode of the differing ASTNodes as the variation point's enclosing software entity. The differing ASTNodes are referenced as software entities by *Variant* elements created for each of the product copies. This initial Variation Point Model is then analysed to identify related variation points and recommend refinements. How this is done is described in the following section.

IV. Variation Point Analysis

The result of the variation point analysis is a Variation Point Model describing a satisfying variability design to serve as input for refactoring the product copies into a software product line.

The initial Variation Point Model, derived from the differences between the product copies, represents all fine-grained differences at the source code level. To achieve a manageable amount of variability in the resulting product line, the Variation Point Model must be refined. Manually analysing all variation points to aggregate them into more corse-grained ones is very time-consuming due to the typically high number of differences.

To support this task, we aim to provide variation point analyses, to identify related variation points and provide recommendations for aggregations. In general, such product consolidations cannot be done in a fully automatic fashion because equivalent alternatives are possible which are selected due to non-technical criteria, such as organisational reasons or personal preferences. To cope with this, the analysis within our SPLevo approach returns only recommendations that the product line engineer can accept, decline, or adapt.

The analysis returns recommendations in terms of variation point aggregations. The following subsections provide details about our graph-based concept to combine and evaluate the different analyses and describe aggregation and filtering techniques as possible variation point refinements.

A. Graph-Based Analysis Composition

Relationships between variation points can exist because of many different aspects, such as their location or type of modification. As our approach is to recommend aggregations of variation points because of their relationships, we consider this scenario as an edge-labelled, undirected multigraph with the variation points as vertices, and their relationships as edges. An edge between two variation point nodes can be created for example because the variation points are located in the same method. The type of a relationship is stored as a label of the according edge (e.g. "CS" because the relationship is derived from the code structure). Because multiple different relationships can exist between two variation points and each of them is represented as an edge, this leads to the multigraph characteristic. In addition, sub-labels can exist for the edges, to allow to provide additional information about the relationship (e.g. the name of the method the variation points are located in).

As shown in Figure 3, the Variation Point Model is transformed into a graph representation by creating a node for each variation point. Next, this graph is handed over to the intended analyses which are performed in parallel. Each analysis creates edges corresponding to the relationships it has identified. Those edges are labelled with the type of relationship identified, e.g. R1, R2, and R3. In the next step, the edges returned from the individual analyses are merged into a combined graph which now contains all edges.

In the final step of the analysis process, the recommendations to refine the Variation Point Model are derived by detection rules inspecting the combinations of relationship types between variation points. A detection rule is specified for a set of edge labels as it's matching pattern. In addition, it specifies a refinement to recommend in case of a pattern match. If a detection rule matches a subgraph of variation points, a refinement recommendation is created for the involved variaton points according to the rule's specification. This refinement is stored in an overall recommendation set to be later presented to the product line engineer.

Detection rules are always applied in a defined order and if edges are matched by a rule's condition, they are ignored by any rules applied later on to prevent possibly conflicting recommendations. The set of rules to apply as well as their order depends on the individual product line requirements and needs to be adapted to the basic analyses performed in a concrete consolidation scenario. The resulting recommendations can consider two different types of variation point aggregations (grouping and merging) which are described in the following subsection.

B. Variation Point Aggregation

A good variation point design is a trade-off between providing variability for as much product options as possible and minimising the number of variabilities to manage. While the former obviously allows to provide more individual product variants, the latter is a best practice for SPL manageability, also documented by Svahnberg et al. [5].

In our SPLevo approach, we start with a fine-grained Variation Point Model on the level of identified code differences. In order to come up with the requirement to have manageable amount of variation points, this requires to aggregate variation points. We distinguish between "merge" and "group" as two types of variation point aggregations which we described in the following subsections.

1) Merge Variation Points: The variation point merge is based on the capability of *Variant* elements to reference several software entities (i.e. *ASTNode* elements) that implement a child feature. *VariationPoints* are merged by consolidating their Variant elements and the referenced software entities in only one of the *VariationPoint* elements. First, one

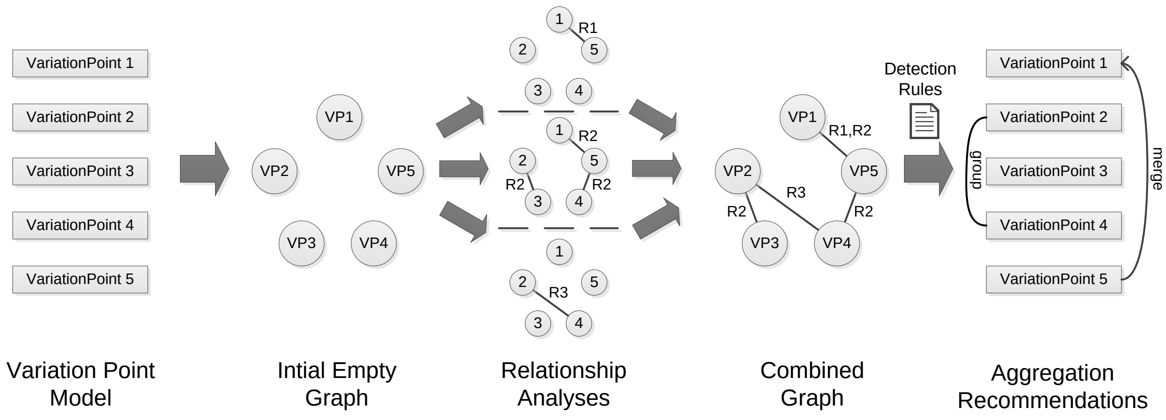

Figure 3. Graph-Based Analyses Composition

VariationPoint element that should survive is selected. Then, for all *Variant* elements in the *VariationPoints*, contributing to the same child feature, the *ASTNode* references are merged into one of the *Variant* elements. It is ensured, that this *Variant* element is contained in the surviving *VariationPoint*. Finally, the remaining empty *Variant* and *VariationPoint* elements are removed from the model.

2) Group Variation Points: The variation point grouping is based on the explicit group structure in the Variation Point Model as described in Section III. Even in the initial Variation Point Model, each *VariationPoint* is assigned to a *VariationPointGroup*. A set of *VariationPoints* is grouped by selecting one of their *VariationPointGroup* elements that should survive and moving all *VariationPoint* elements into this. Afterwards, the remaining empty *VariationPointGroup* elements are removed from the Variation Point Model.

C. Variation Point Filtering

Complementary to the previously described aggregations, variation points can be filtered from the Variation Point Model. This can be done, for example, to remove identified variation points which are not of interest for the product line engineer and his intended product line. In such a case, the variation point is simply removed from the Variation Point Model. A removed variation point, will no longer result in a refactoring advice in the downstream process.
Filtering strategies are not part of the analysis discussed in this paper but mentioned for completeness as an alternative Variation Point Model refinement.

V. Relationship Analyses

As described in the last section, our variation point analysis is a composed analysis of the relationships between variation points. It allows for the use of serveral basic analyses which focus on specific relationship types according to different aspects of software changes.

We have identified a list of different relationship types between variation points and according strategies to identify them:

- Code Structure (CS)
- Program Dependency (PD)
- Data Flow Dependency (DFD)
- Program Execution Trace (PET)
- Change Type (CT)
- Change Pattern (CP)
- Cloned Change (CC)
- Semantic Relationship (SR)
- Modification Time (MT)

Those strategies are discussed in more detail in the following subsections.

A. Code Structure (CS)

The code structure analysis studies the phyisically implemented code structure, such as statements contained in methods and methods contained in classes. Variation points located in the same structure are identified to have a code structure relationship. Examples for such relationships are:

- Statements in the same block statement
- Statements in the same method
- Methods in the same class
- Classes in the same component

Up to the level of classes, this hierarchy is directly provided by the software entity models extracted from the implementations under study. Depending on the product copies under study it might be possible to get higher level structures, e.g. a component architecture, from documentation or reverse engineering techniques. If architecture information is available it could be included in the code structure analysis [15].

B. Program Dependency (PD)

The program dependency analysis identifies dependencies between the variation points' software artifacts based on programming language features. For example, the PD analysis identfies relationships between a variation point containing a varying variable declaration statement and all variation points which describe added, deleted or changed statements consuming this variable. Program dependendencies overlap with data flow dependencies described below but are not the same.

C. Data Flow Dependency (DFD)

A data flow dependency identifies sofware elements which handle the same data object and potentially influence each by this data object. The software entity models extracted from the implementations are sufficient for a static data flow analysis. And, because the data flow is inspected, only data processing model elements need to be considered. Examples for potential dependencies are:

- Statements manipulate the same variable
- Method calls with one using the others' result as input
- A statement depends on a changed class attribute
- A method invocation of a changed method

Adding a data object identifier as an additional sub-label to the data flow dependency relationship label ensures to not mix up relationship edges resulting from different data flow dependencies.

D. Program Execution Trace (PET)

Program execution traces represent a programs execution flow monitored during the execution of one or more specifc features. They can be gathered from instrumenting the program code before it's execution. Alternatively, a profiler can be used, that returns information about the dynamic beahaviour of the software, e.g. method invocations or object instantiations.

Variation points represent locations of variability. If the locations of two variation points are contained in the same execution trace, this is considered as a program execution trace relationship between those variation points. An identifier of the executed feature, respectively of the execution trace is added to the relationship label. This information ensures to not mix up variation point relationships resulting from execution traces of different features.

The execution traces are external information sources which need to be included by the analysis. This also requires to match the elements of the execution traces to elements in the software entity models referenced in the Variation Point Model. This can be done by matching the full qualified names of the software entities.

E. Change Type (CT)

A change type describes a modification of a software entity, e.g. a parameter added to a method signature or a statement removed from a method body. Fluri et al. [16] have developed a taxonomy of such change types to analyse the evolution of a single software system.
However, those change types can also be used for a variation point analysis. In a first step, the differences between the variants of a single variation point can be classified according to a change type taxonomy as provided by Fluri et al.. In a second step, variation points classified with the same change type can be identified and a relationship edge can be created with a change type relationship label. Adding the specific type of change to the relationship label ensures that relatinships resulting from different changed types are not mixed up.

F. Change Pattern (CP)

A change pattern also describes a modification between the variants of a variation point. Change patterns can involve multiple software entities and can be specific to the system under study. For example, a change pattern can describe that a boolean parameter is added to a method signature, checked for a null value in a conditional statement at the beginning of the method, and the control flow is returned if a null value is found. In addition, it is possible to specify low-level change types more specifically. For example, not only all variation points with an added parameter can be detected, but all variation points with an added parameter of a specific data type and a specific name.

If two variation points match to the same change pattern, a relationship edge is created between them and labelled as change pattern relationship combined with an identifier for the specific change pattern as sub-label.

Compared to the change type analysis, the change pattern analysis is not limited to standard software changes. It is able to consider a combination of multiple changes and provides a higher flexibility. The patterns must be specified in advance of the analysis.

G. Cloned Change (CC)

According to Roy et al. [17], *"a code clone are two code fragments which are similar by a given definition of similarity"*. Clone types range from direct copies up to code sections that perform the same computation but have different implementations. Roy et al. further specified four types of clones:

- Type 1: Code Layout & Comments
- Type 2: Literals Changed
- Type 3: Added, Changed, or Removed Statements
- Type 4: Same Computation but other Implementation

The cloned change analysis makes use of this by applying a clone detection to the code changes of the variation points. For example, if two variation points are about a set of added statements, the clone detection is applied to those statements to check if the same code has been added at these variation points. If this is true, a code clone relationship edge is added for those two variation points.

Compared to the previous change type and change pattern analysis, the cloned change analysis makes use of a mining approach without predefined generic or specific change patterns to match. Furthermore, depending on the analysed types of clones, additional similarities which are not possible to be specified as patterns might be detected.

H. Semantic Relationship (SR)

Developers often introduce semantics not only in comments but also in the names of their variables, methods and classes [18]. Semantic code clustering techniques take use of this to find clusters of related code in software products. The semantic relationship analysis makes use of such techniques to identify relationships between variation points. The semantic clustering is applied to the software entities referenced by variants of the variation points under study. The clustering algorithm will return related software entities. The referencing variants will provide the link back to the enclosing variation points. If more than one variation point is connected to such a cluster, a relationship edges labelled as semantic relationship with an identifier for the semantic cluster are created between them.

I. Modification Time (MT)

The modification time analysis investigates in changes performed at the same time. This is applied to software elements referenced by variants contributing to the same child feature. If software entities referenced by variants of two variation points have been modified at the same time, an edge labelled as modification time relationship is created between those variation points.

The software entities' modification time is typically provided by a revision control system such as cvs, git or svn. When a developer has finished a modification, he commits one or more resources, each with one or more modified software entities, into the system. The time of the commit is interpreted as modification time instead of the exact point in time when a developer has modified a single file on his local system. The later is rarely available and the commit provides more useful information because more than one file is involved and a commit typically represents a completed modification.

In addition to the time of the commits, the commit message can be used to identify related commits. This potentially allows to identify even more software entities changed together compared to considering only a single modification.

VI. Illustrating Example

In [15] we have introduced a greatest common devisor (GCD) example program [19] with a native Java and a JScience-based product variant [20]. The two implementations differ in a method named `gcd()`. This method transforms two string parameters into numeric values, calculating their GCD, and returning the result as a string as shown in Listing 1 and 2.

Listing 1. Standard Java Implementation

```
import java.math.BigInteger; //VP1
...
public String gcd(String v1, String v2){
 BigInteger intV1 = new BigInteger(v1); //VP2
 BigInteger intV2 = new BigInteger(v2); //VP3
 BigInteger gcd = intV1.gcd(intV2);  //VP4
 return gcd.toString(); //VP5
}
```

Listing 2. JScience-Based Implementation

```
import org.jscience.mathematics.number.LargeInteger; //VP1
...
public String gcd(String v1, String v2){
 LargeInteger intV1 = LargeInteger.valueOf(v1); //VP2
 LargeInteger intV2 = LargeInteger.valueOf(v2); //VP3
 LargeInteger gcd = intV1.gcd(intV2); //VP4
 return gcd.toString();  //VP5
}
```

Diffing the abstract syntax trees, extracted from the two product copies, detects any changes the implementations. This leads to five variation points which are marked as VP1 to VP5 in the presented code listings. VP1 is a changed import. VP2 to VP4 are about statements declaring variables of different data types. Additionally, in VP5, methods with similar names — `gcd()` — but for variables of differing types are called. While lexical similar, the semantic difference between these calls is detected by comparing the methods' abstract syntax trees and not textual representations only. During the analysis, the graph derived from the initial variation point model contains nodes for each of the variation points as shown by the *Initial Graph* in Figure 4.

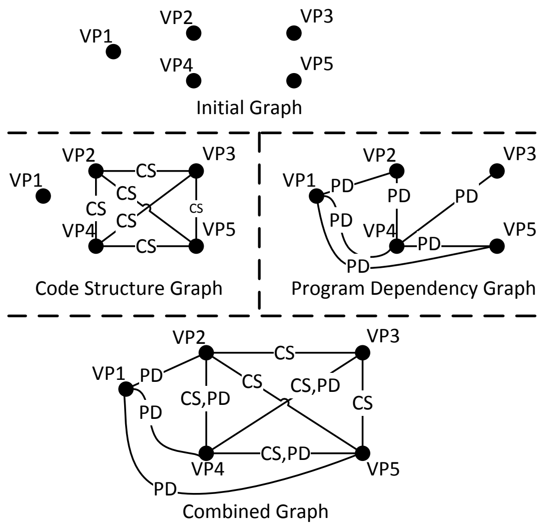

Figure 4. Illustrating Example Graphs (sub-labels omitted for simplicity)

Applying the Code Structure analysis identifies relationship edges, labelled with "CS", between the nodes VP2, VP3, VP4 and VP5, because all of them are located in the method `gcd()`.

Applying the Program Dependency analysis identifies relationships between VP1 and each of VP2, VP3, and VP4 because each of them makes use of the class imported in VP1. Each of the resulting relationship edges is labelled as "PD" and sub-labelled as "import". Furthermore, the Progam Dependency analysis identifies dependencies between VP2 and VP4, VP3 and VP4, as well as VP4 and VP5 because of their variable usages. Relationship edges with the label "PD" and the sub-label "variable-usage" are created.

In the last step, two detection rules are applied in the order as they are described here. The first rule detects edges labelled with "CS" and "PD (variable-usage)". For all matching cliques in the graph, a merge is recommended for the according variation points. VP2, VP3, VP4, VP5 in the example. The second rule applied, detects edges labelled as "PD (import)" and recommends to group the variation points involved in the identified cliques. VP1, VP2, VP3, and VP4 in the example.

Later on, when the recommendations are applied, the recommendation order will take effect. First, VP2 to VP5 are merged into a variation point VP6, and then VP6 is grouped with VP1.

VII. Related Work

Our analysis concept presented in this paper is developed in the context of variability reverse engineering from product copies to be consolidated into a software product line.

A strongly related approach has been developed by Graaf et al. [6] to identify variation points based on program execution traces. Similar to them, we consider feature location techniques based on static or dynamic program graph analysis as done by Rajlich et al. [21], and Wilde et al. [22], as relevant for the detection of program differences relating to a common feature. However, we handle this as one technique beside others to be considered. Furthermore, Graaf et al. do not discuss the influence or the handling of code differences which are not relevant for the product line variability (e.g. code beautifying etc.). In our approach, this can be handled by filtering the according variation points.

Our work focuses on the creation of SPLs from customised product copies with a common initial code-base, comparable to an approach of Koschke et al. [23] facilitating a reflexion method. However, our approach does not depend on a pre-existing module-architecture as the extended reflexion method of Koschke et al.. Furthermore, we aim to take custom SPL requirements ('SPL Profile') into account, such as preferences for variation point design and realisation techniques.

Alves et al. [24] formalised valid, feature-aware refactorings of SPLs. Their approach is complementary to ours and can be considered as part of our refinement step to refactor initially created variation point models to a more homogenious product line. They do not provide support for identifying code differences contributing to a feature. Instead, they assume feature models to be derived from documentation or manual code examination. However, they state automated support for this as desirable.

She et al. [25] have presented an approach for reverse engineering the hierarchal feature model structure and constraints from a given set of features and feature dependencies. We also aim to build a variability model, we do not assume a set of features and dependencies as input as they do, but we aim to reverse engineer this information from the given product copy implementations. While they cluster predefined features based on given dependencies, we cluster variation points based on analysed relationships to identify reasonable variable features.

Yoshimura et al. [26] also worked on identifying related variabilities to improve a product line's manageability. However, their goal is to recommend feature constraints based on customer preferences from the past for an existing product line. In contrast, we aim to identify technical and logical dependencies between varying code entities to design variation points as part of the process of creating a new product line.

VIII. Assumptions and Limitations

The main assumption of our approach is the consolidation of product copies with a common code base. For products developed completly independent from each other with a similar purpose only, the differencing in general and the analysis concept presented in this paper in specific, cannot be expected to return reasonable results.

Furthermore, the approach focuses on systems developed in object-oriented programming languages. For other programming languages, especially the concepts relying on the code structure need to be checked and probably adapted.

Within object-oriented programming languages, the lowest level of granularity we examine are statements. Modifications on the expression level, for example a partly modified composite condition of an if-statement, is interpreted as a modified if-statement rather than a changed sub-expression. This is done to reduce the amount of information to process.

Some changes in customised product copies are not relevant for variable features of the resulting software product line, but might be of interest from a maintenance perspective. For example, an improved code formating or documentation should be adopted in the resulting implementation. However, this requires a simple merge process from the customised variant into the resulting product line, which is not in the focus of this paper and part of our planed future work.

IX. Conclusion

In this paper we have presented our variation point analysis concept developed in the context of the SPLevo approach for consolidating product copies into a common product line. The analysis identifies recommendations to refine an intialy fine-grained Variation Point Model in terms of reasonable aggregations. A graph-based representation of the variation points is used to enable the composition of basic analyses for different relationship types. The identified relationships are represented as labelled edges within an overall graph. The refinement recommendations are derived from this graph by applying detection rules specifying combinations of relationship types as match patterns and the type of refinement to recommend in case of a match.

The analysis concept presented in this paper supports a product line engineer in understanding the differences between the product variants to consolidate and in creating a proper variation point design for the intended software product line.

Currently, we are developing a prototype implementation of the overall SPLevo approach as an Eclipse-based application [8] which already automates the presented example. We furher use and improve it within a case study facilitating the SPL variant of ArgoUML provided by Couto et al. [27]. In this case study we analyse differing implementations generated by their pre-processor facilities and try to reverse engineer the originated product line as a reference. Following this case study, we are going to analyse product copies of our industrial partners with customised product copies created on project contexts and due to organisational reasons. In a further step, we will investigate sets of predefined analysis and detection rule for typical software product line requirements.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF), grant No. 01IS12012B.

References

[1] L. Clements Paul ; Northrop, *Software product lines : practices and patterns*, 6th ed., ser. SEI series in software engineering. Boston, Mass.: Addison-Wesley, 2007.

[2] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[3] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: a systematic review," in *Proceedings of SPLC'09*. Carnegie Mellon University, 2009.

[4] T. Patzke and D. Muthig, "Product Line Implementation Technologies," Fraunhofer IESE, Kaiserslautern, Tech. Rep. 057, 2002.

[5] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[6] B. Cornelissen, B. Graaf, and L. Moonen, "Identification of variation points using dynamic analysis," in *Proceedings of R2PL'05*, 2005.

[7] W. Koleilat and N. Shaft, "Extracting executable skeletons," Cheriton School of Computer Science, University of Waterloo, Waterloo, Tech. Rep., 2007.

[8] B. Klatt, "SPLevo Website," 2013. [Online]. Available: http://www.splevo.org

[9] B. Klatt and K. Krogmann, "Towards Tool-Support for Evolutionary Software Product Line Development," in *Proceedings of WSR'2011*, 2011.

[10] ——, "Model-Driven Product Consolidation into Software Product Lines," in *Proceedings of MMSM'2012*, 2012.

[11] OMG, "Architecture-driven Modernization : Abstract Syntax Tree Metamodel (ASTM)," OMG, Tech. Rep. January, 2011.

[12] ——, "Architecture-Driven Modernization : Knowledge Discovery Meta-Model (KDM)," OMG, Tech. Rep., 2011.

[13] Eclipse Foundation, "EMF Feature Model," 2012. [Online]. Available: http://www.eclipse.org/modeling/emft/featuremodel/

[14] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of ASE'10*. ACM, 2010.

[15] B. Klatt and M. Küster, "Respecting component architecture to migrate product copies to a software product line," in *Proceedings of WCOP'12*. ACM Press, 2012.

[16] B. Fluri and H. Gall, "Classifying Change Types for Qualifying Change Couplings," in *Proceedings of ICPC'06*. IEEE, 2006.

[17] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, 2009.

[18] A. Kuhn, S. Ducasse, and T. Girba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[19] B. Klatt, "GCD Calculator Example," 2012. [Online]. Available: http://sdqweb.ipd.kit.edu/wiki/GCD_Calculator_Example

[20] J.-M. Dautelle, "JScience," 2012. [Online]. Available: http://www.jscience.org/

[21] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of IWPC'2000*. IEEE, 2000.

[22] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal Of Software Maintenance Research And Practice*, vol. 7, no. 1, 1995.

[23] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," *Software Quality Journal*, vol. 17, no. 4, pp. 331–366, Mar. 2009.

[24] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring Product Lines," in *Proceedings of GPCE'06*. ACM, 2006.

[25] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of ICSE'11*. IEEE, 2011.

[26] K. Yoshimura, Y. Atarashi, and T. Fukuda, "A Method to Identify Feature Constraints Based," in *SPLC'10*. Springer Berlin Heidelberg, 2010, pp. 425–429.

[27] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting Software Product Lines : A Case Study Using Conditional Compilation," in *Proceedings of CSMR'11*, 2011.