Automating the Extraction of Model-based Software Product Lines from Model Variants

CONFERENCE PAPER · NOVEMBER 2015

READS
108

5 AUTHORS, INCLUDING:

Tewfik Ziadi
Pierre and Marie Curie University - Paris 6
37 PUBLICATIONS **393** CITATIONS
SEE PROFILE

Tegawendé F. Bissyandé
University of Luxembourg
45 PUBLICATIONS **128** CITATIONS
SEE PROFILE

Jacques Klein
University of Luxembourg
93 PUBLICATIONS **925** CITATIONS
SEE PROFILE

Automating the Extraction of Model-based Software Product Lines from Model Variants

Jabier Martinez*†, Tewfik Ziadi†, Tegawendé F. Bissyandé*, Jacques Klein* and Yves le Traon*
*SnT, University of Luxembourg, Luxembourg. firstname.lastname@uni.lu
†LiP6, Sorbonne Universités, UPMC Univ Paris 06, Paris, France. tewfik.ziadi@lip6.fr

***Abstract*—We address the problem of automating 1) the analysis of existing similar model variants and 2) migrating them into a software product line. Our approach, named MoVa2PL, considers the identification of variability and commonality in model variants, as well as the extraction of a CVL-compliant Model-based Software Product Line (MSPL) from the features identified on these variants. MoVa2PL builds on a generic representation of models making it suitable to any MOF-based models. We apply our approach on variants of the open source ArgoUML UML modeling tool as well as on variants of an In-flight Entertainment System. Evaluation with these large and complex case studies contributed to show how our feature identification with structural constraints discovery and the MSPL generation process are implemented to make the approach valid (i.e., the extracted software product line can be used to regenerate all variants considered) and sound (i.e., derived variants which did not exist are at least structurally valid).**

I. Introduction

In the realm of software engineering, models, which are high-level specifications of systems, have progressively taken importance for researchers and practitioners as the primary artefacts of development projects. Traditionally, modeling has been used in a descriptive way to represent systems by abstracting away some aspects of the systems and emphasizing others [1]. Nonetheless, prescriptive modeling is now trending and is relied upon to automate the generation of products as well as their validations [2]. In this context, models are often extended, customized or simply reconfigured for use in particular system settings. Thus, an important challenge in Model-Driven Engineering (MDE) is to develop and maintain multiple variants, i.e. similar models, by exploiting the features that the models share (commonalities) and managing the features that vary among them (variabilities) [3]. To address this problem, which is relevant to many kinds of software artefacts, Software Product Line (SPL) engineering [4] has matured as an approach for 1) managing the commonalities and variabilities of a product family and 2) managing the derivation of tailored products by combining reusable assets.

Unfortunately, SPL adoption is still a major challenge in industry [5], [6]. Instead, practitioners use to rely on ad-hoc mechanisms, such as copy-paste-modify, to perform reuse when creating similar products. It has even been reported that more than 50% of industrial practitioners formally implement an SPL only after the instantiation of several similar variants using ad-hoc reuse techniques [6]. Figure 1, that will be discussed later, illustrates ad-hoc reuse for variants creation and the circumstances to consider the adoption of more advanced variability management approaches. However, migrating product variants into an SPL is a known challenge in SPL engineering [7], [8]. Several approaches, referred to as *extractive* or *bottom-up* approaches, have been proposed for source code artefacts. In this work, we focus on models, and propose an end-to-end solution for dealing with the two requirements for building a bottom-up approach to the adoption of Model-based Software Product Lines (MSPL):

- *Feature identification*. Analysing and comparing the existing model variants (i.e., models that are used using ad-hoc reuse techniques) to identify commonality and variability in terms of features. A feature is a prominent or distinctive characteristic, quality or user-visible aspect of a software system or systems [9].
- *Reengineering*. Once features are identified, and the constraints among them are detected, extractive approaches should propose a transformation phase where the model variants are actually refactored to conform to an SPL approach.

With MoVa2PL (Model Variants to Product Line) we address simultaneously in a single framework both requirements for extracting an MSPL from model variants. In the realization of our approach, we used the Common Variability Language (CVL) [10] to implement the MSPL. The benefits of this work for easing the adoption of SPL engineering for models are numerous. Indeed, first, adopting an MSPL, as for any other kind of SPL, will allow practitioners to easily and efficiently propagate changes done in one feature to all existing variants by simply re-deriving them automatically. Second, the extracted MSPL can be relied upon to derive efficiently new products by combining features. This work does not focus on the challenging problem of model similarity analysis in the sense that we assume that the variants are not independently generated out of the same family of systems.

The contributions of this work are:

1) We provide an approach for commonality and variability analysis that includes the automatic detection of constraints between identified features.
2) We propose a method for MSPL reengineering to automatically extract a CVL-compliant MSPL.
3) We assess the approach with two case studies.

The remainder of this paper is structured as follows: Section II summarizes our previous work on feature identification in models and presents the motivation to MSPL adoption. Section III presents our approach. Section IV presents experiments based on case studies with large real-world systems. Section V discusses the approach and its limitations. Section VI presents related work. Finally, Section VII concludes this work and outlines future directions.

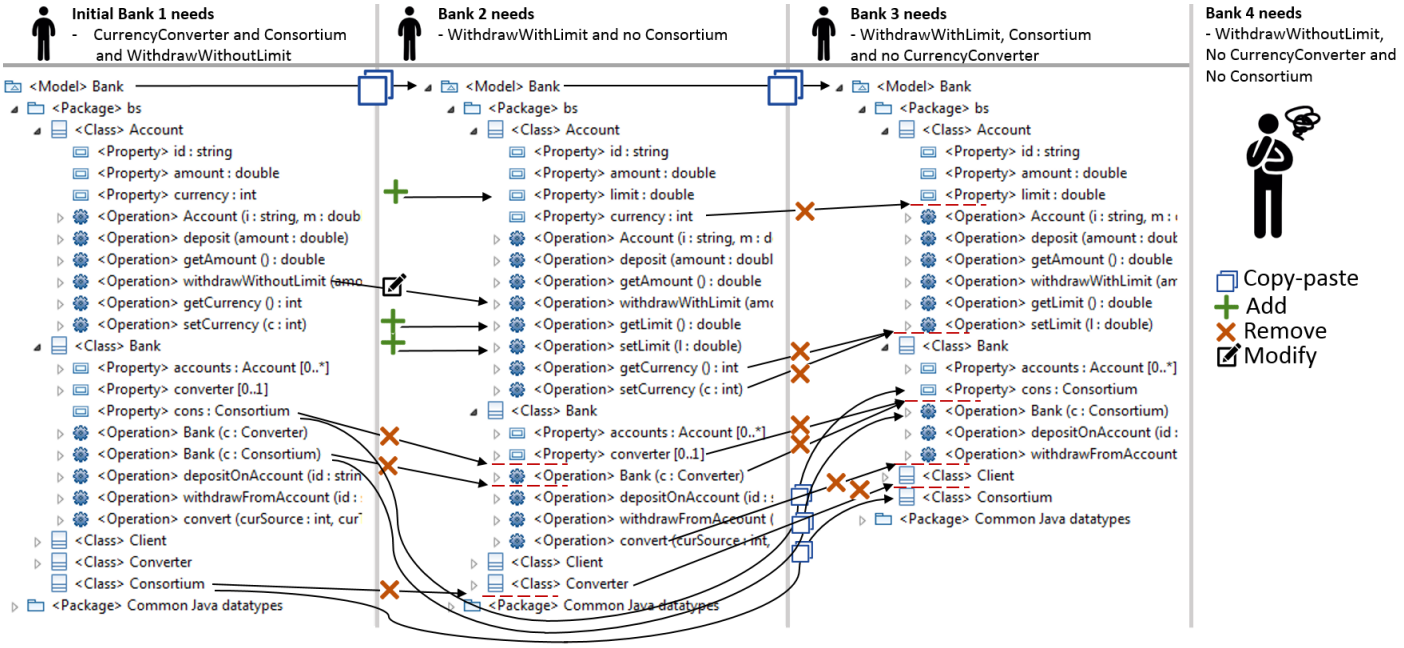


Fig. 1: Three UML model variants fulfilling different bank systems' needs and the manual actions for their creation

II. Background and Motivation

In previous work we introduced MoVaC [11] as a meta-model independent approach for comparing simultaneously a set of model variants in order to identify commonality and variability. The approach was flexible to apply or integrate different matching techniques. Figure 2 shows the proposed process whose objective was to help domain experts in feature identification. The approach was designed to support concepts from the Meta-Object Facility (MOF) [12] which is a well-known standard in MDE.

Each model variant is decomposed into Atomic Model Elements (AMEs) since models are artefacts that can be expressed as a sequence of elementary construction operations [13]. The computed AMEs are either of type *Class* (not to be confused with UML classes), *Attribute* or *Reference*. Then, we proposed an algorithm, called *Interdependent AMEs*, which automatically identify sets of AMEs that correspond to the distinguishable features from the model variants. We named Blocks to these sets of AMEs. Unfortunately, MoVaC has two main limitations: although it helps in feature identification, (1) it does not manage to identify the constraints that exist among the features and (2) it only focuses on one part of the problem for MSPL extraction since it does not propose a method to extract an SPL from the identified commonalities and variabilities.

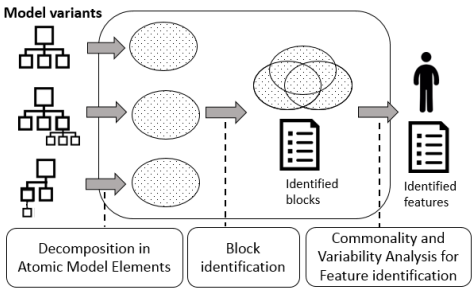


Fig. 2: Model Variants Comparison (MoVaC) approach to identify and analyse commonality and variability

We present in Figure 1 our running example illustrating a scenario of building UML model variants for different banking systems [14], [15], [16]. In this scenario we have created three models through ad-hoc reuse processes with variations on the limit of bank withdrawal, the consortium entity and currency conversion. Thus, the first created variant, `Bank 1` UML model, is implemented with information related to currency conversion and consortium. The `Bank 2` UML model includes a new requested feature that is the support of a limit in the withdrawal. This new banking system does not however need consortium. Thus, in order to create it, we consider a copy of `Bank 1` where we added one UML property, two UML operations, we modified the *name* attribute of a UML operation and we removed the UML class Consortium and all its related UML elements (one UML class and two UML operations). The needs of `Bank 3` on the other hand include limit in the withdraw and consortium, but no currency conversion. To create this variant, we build from a copy of `Bank 2` where we removed all UML elements related to currency conversion (one UML property, four UML operations and one UML class). However, we also selected and copied UML elements from `Bank 1` variant to complete the implementation of this `Bank 3` variant.

The building of variants for such a basic running example aims to illustrate how time-consuming and error-prone the manual creation of variants can be in real-world complex scenarios. This manual process quickly ceases to be sustainable if we consider the possibility that continuously creating new variants will require even more effort in finding, selecting and reusing elements from other variants. Furthermore, because of a lack of an explicit formalization of feature relationships, potential inconsistencies in the requested features for a new variant will be found during/after the variant creation.

To extract an MSPL from the variants of our running example, feature identification would consist in analysing the three UML variants of Figure 1 to identify the core elements of a banking system and the different features related to currency converter, consortium and limit support, as well as the constraints among them. *Reengineering* will then exploit the identified features and the existing variants to build the necessary assets for the MSPL.

In MDE, CVL [10] is a modeling approach that is specialized for implementing MSPL in a non-invasive, language independent way. CVL further provides the necessary expressiveness to support systematic reuse for the derivation of models [17]. CVL was thus designed to simplify a top-down approach to the adoption of MSPL, where practitioners directly write the assets for defining the MSPL. In a bottom-up approach, the challenge is to leverage existing variants to extract these assets. CVL consists of two layers:

- **Variability definition layer:** This layer defines the variability of the domain in a very similar fashion as feature modeling. Feature modeling is the most widely used formalism in SPL engineering to represent the features of a product family and the relationships between them [9].
- **Product realization layer:** The replacement actions in a *Base Model* must be defined according to the variability specification.

In this work we are concerned with the following research questions:

RQ1: Based on existing model variants, can we derive automatically a Variability definition layer that ensures the validity of the configuration space?

RQ2: Can we automatically infer a Product realization layer for variants of complex systems?

III. Extraction of Model-based Product Lines with MoVa2PL

The overall process of MoVa2PL is illustrated in Figure 3. In the first step, we leverage MoVaC [11] to decompose the model variants into AMEs (the *Decomposition in Atomic Elements including dependencies* step of Figure 3) and we perform Block and Feature identification. *Block identification* is performed by computing the interdependence among AMEs. As defined in our previous work [11], given a set of model variants MVs, two AMEs ame_1 and ame_2 (of models from MVs) are interdependent if the two following conditions are fulfilled:

1) $\exists M \in MVs \;\; ame_1 \in M \wedge ame_2 \in M$
2) $\forall M \in MVs \;\; ame_1 \in M \Leftrightarrow ame_2 \in M$

A Block is thus a set of interdependent AMEs that are distinguishable in the model variants. Feature identification is a semi-automatic process where domain experts manually review the elements from the identified blocks to map them with the functionalities (i.e., features) of the system.

However, we augment the information of AMEs with the computation of all dependencies among AMEs in all variants. Thus, once the features are already identified, we can implement a *Constraints discovery* step which will allow a more reliable definition of the variability model for the MSPL.

Also we modify the decomposition process. In MoVaC we considered only the attributes and references which have not the property of being derived, volatile or transient. For example, if an attribute has the derived flag in a given metamodel, it means that its value is automatically calculated from other attributes or by some function, thus MoVaC does not add this as attribute AME during decomposition of the model in AMEs. In MoVa2PL, apart from these conditions, we also add the condition that their values must have been set before (e.g. non null values or empty lists of referenced elements).

As presented in Figure 3, MoVa2PL has a *Visualisation* layer that covers its different steps. Indeed, as we will show above, real-world case studies yield model variants that can be large and complex with a high number of model elements and constraints. Apart from automating the process of extracting MSPL, MoVa2PL uses visualisation techniques [18] to help domain experts to visually analyse, not only the results of this automation, but also the intermediary results in order to provide a better understanding of the automatic underlying process. The generation of feature constraints' graphs is thus automated in MoVa2PL.

The tool-support of MoVa2PL is implemented and integrated in the Bottom-Up Technologies for Reuse framework [19]. In the following subsections we provide details of the constraints discovery strategy and the MSPL extraction.

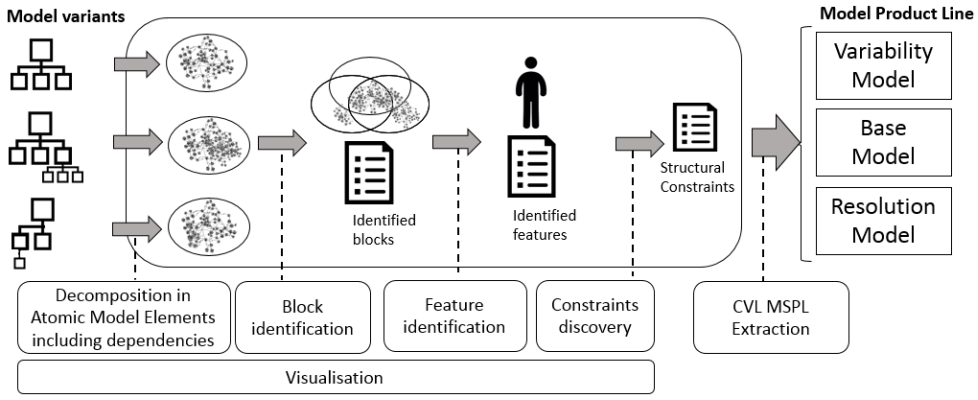


Fig. 3: MoVa2PL Process

A. Structural constraints discovery

Variability modeling in product line engineering is known as an efficient paradigm to describe at once several variants of a system. Precisely specifying a variability model is essential to guarantee that any derived model is valid. A structurally valid model is a model that does not violate any constraint defined in the meta-model such as cardinality of the model references or the non existence of dangling elements (i.e. an element that does not have a parent). Semantic validity of model variants, which is checked by domain experts, is out of the scope of this paper. In this section, we focus on how we compute the dependencies between AMEs and how, by reasoning on these dependencies, we infer constraints between features to ensure that the models that will be derived from the extracted MSPL are structurally valid.

1) Dependencies between Atomic Model Elements: A dependency involves a pair of AMEs and it has an ID which corresponds to the dependency type. Each dependency type has an upper bound representing the maximum number of times an AME can be referenced with its dependent counterpart.

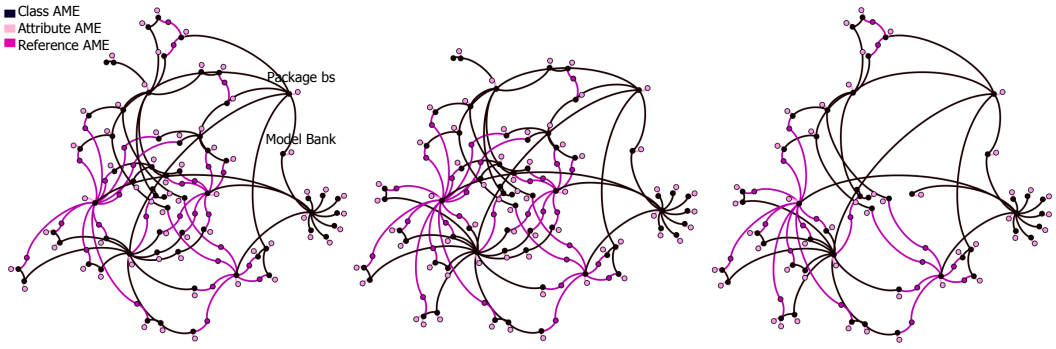
The AME dependencies are calculated as following:

- **Class**: A class AME depends on its parent class AME. By parent we mean the container relation (not to be confused with inheritance in a UML sense). The dependency ID is the ID of the containment reference as it was given in the meta-model. The upper bound is the containment reference upper bound. For instance, from the running example, the class AME `UML Operation deposit` depends on the class AME `UML Class Account`, its dependency ID is `ownedOperation` and, in this case, there is no upper bound given that a UML Class can have unlimited owned operations.
- **Attribute**: An attribute AME depends on the class AME that hosts the attribute. The dependency ID is the ID of the attribute. The upper bound is 1 in the sense that a class cannot have the same attribute twice. As an example of such dependency we note that any UML Operation has an attribute AME `name`. In `UML Operation deposit`, the attribute AME `name` depends on `UML Operation deposit` and there can only be one attribute `name`.
- **Reference**: A reference AME depends on the class AME that hosts the reference and on each of the class AMEs that are referenced. The dependency ID is the ID of the reference in the case of the host class, and the hard-coded value `referenced` is used as ID for the dependencies to the referenced class AMEs. The upper bound in a dependency with the host is 1 as it happened for the attribute AME. However the upper bound in a dependency with referenced classes is unlimited given that a class AME can be referenced as many times as desired. From the running example we note that the reference AME `Type` of `UML Parameter amount` depends on `UML Parameter amount` and also depends on the class AME `UML Primitive Type double`.

Figure 4 shows the AMEs of `Bank 1`, `Bank 2` and `Bank 3` and their corresponding dependencies using a directed graph visualisation. All the graphs shown in this paper are automatically generated by MoVa2PL. The direction of the edges are clockwise. We can see how the class AMEs (in black) are surrounded by the attribute AMEs (light color) and reference AMEs (dark color) that depend on each class AME. We can also see, as black edges, how some class AMEs depend on other class AMEs through a containment relation (given that a class depends on its parent class AME). The graph shows how attribute AMEs only depend on their corresponding host class AME. On the contrary, we can see the reference AMEs that, besides their host class AME, also depend on the referenced class AMEs.

2) Requires constraints discovery: For the discovery of structural constraints between features we reason on the dependencies of the AMEs. In order to avoid dangling elements in derived models after MSPL construction, we identify the *requires* constraints that will assure that all the model classes (except the root) will have a parent. We also identify the *requires* constraints between any pair of features where one feature needs the other in order to be successfully integrated in a system variant.

In Figure 5 we show the blocks that were identified by computing the interdependence between AMEs. The `BankCore` Feature, which corresponds to the first block (Block 0), comprises most of the AMEs. Specifically, it comprises those that are common for all model variants. On the contrary,



(a) Bank System variant 1 (b) Bank System variant 2 (c) Bank System variant 3

Fig. 4: Visualisation of the decomposition of three model variants into Atomic Model Elements

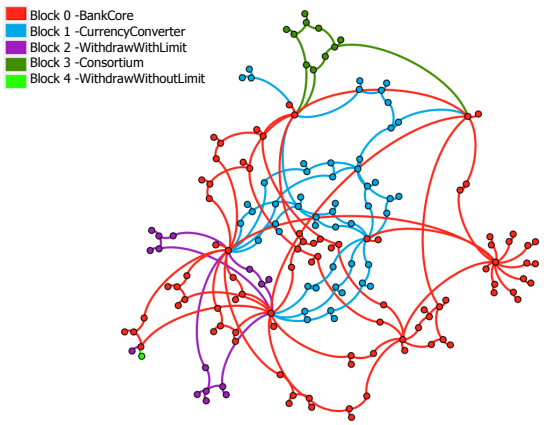


Fig. 5: Bank variants' identified Features. The dependencies among their associated Atomic Model Elements are used for constraints discovery

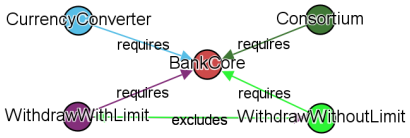


Fig. 6: Structural constraints discovered between the features of the Banking System

`WithdrawWithoutLimit` is only based on one AME that is the attribute AME of the `name` of an `Operation`. In this Figure, each AME shows also its dependencies with other AMEs through the edges. We can see how most of these dependencies are intra-feature structural dependencies given that they exist between AMEs corresponding to the same feature. However, we can see inter-feature structural dependencies that in this case are all in the direction to the Block 0 (i.e., `BankCore` feature). We identify the *requires* constraint between two features when at least one AME from one feature has a structural dependency to an AME of the other feature. Figure 6 thus shows the discovered *requires* constraints in the running example.

3) Mutual exclusion constraints discovery: Cardinality in model references plays an important role in defining a domain specific language. These cardinalities define constraints in the domain that must not be violated to obtain valid models. To avoid violation of upper bound cardinalities we identify in which cases two features cannot coexist in the same model (*mutual exclusion* constraint).

To illustrate upper bound cardinalities in real scenarios, and following with the context of the running example, we discuss the cardinalities of the widely used UML meta-model. In the case of the UML meta-model, as implemented in Eclipse UML2 ecore, we have 242 classes with a total of 3113 non-volatile, non-transient, non-derived references. 67.7% of these references have no upper bound. 32.2% of them have an upper bound of 1 maximum referenced model classes. The remaining 0.1% corresponds to the `DurationObservation` meta class which allows to model execution durations in UML. This class has the `event` reference with an upper bound of 2 UML `NamedElements`.

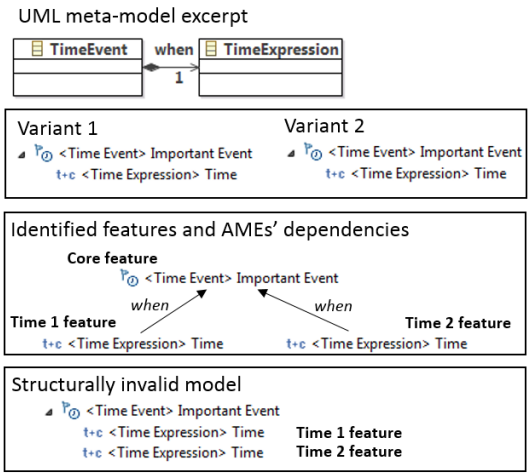


Fig. 7: Illustrative example of structurally invalid model as result of composing two features

If the upper bound of a containment reference is 1, that means that it is structurally invalid to try to reference 2 different containments at the same time. Figure 7 illustrates an example where a structural invalid model is created by combining two features. In the UML meta-model, the `TimeEvent` class has the `when` containment reference with an upper bound of 1 to a `TimeExpression` class. Therefore, it is not valid to reference two `TimeExpression` classes using this reference. Suppose we have a class AME `TimeEvent` that is part of the common feature and then two different class AME `TimeExpressions` belonging to two different features. Both class AME `TimeExpressions` will have a dependency to the same `TimeEvent` class AME. The dependency type of both dependencies correspond to the `when` dependency ID that has an upper bound of 1. Given such information we can identify *mutual exclusion* constraints between two features when the sum of the dependencies of a given dependency type that points to a given AME from one feature and the dependencies which points to the same AME with the same dependency type from the other feature, exceeds the defined upper bound of this dependency type.

This is applicable also for the reference AME and for the attribute AME. In our running example, we have two attribute AMEs that depend on the same `UML Operation` class AME with the `name` dependency ID. In one of them the value is `withdrawWithLimit` and in the other is `withdrawWithoutLimit`. These attribute AMEs correspond to two different features and only 1 can be used in a derived model. This discovered constraint is made visible in Figure 6 by an "excludes" link between the two features.

B. CVL models generation

Different strategies can be selected for implementing an MSPL ranging from additive, subtractive or hybrid approaches [20], [21]. The additive approach relies on a Minimum Base Model composed only by the Core of the family of models, i.e. the model elements that are common to all model variants. Then the approach requires Library Models which will contain the model fragments to be added to the Base Model. On the contrary, the subtractive strategy consists

in constructing the Maximum Base Model (also known as 150% model) and then removing the model elements related to each of the non selected features. Hybrid approaches use a mix between subtractive and additive strategies by providing to some extent Library models and still leaving the possibility to subtract from the Base Model. In MoVa2PL, we rely on a subtractive strategy. Notice that it is possible to construct a Maximum Base Model even if the resulting Base Model violates reference upper bounds. It will be the responsibility of the resolution model to operate in the Base model to bring it to a valid state. The following paragraphs explain how the CVL layers, described in Section II, are created. Using these layers, CVL implementations provide an engine to automatically transform the Base model in a resolved model.

Variability model: The variability model is created using information from the identified features and the discovered constraints. Figure 8 shows the CVL variability model created from the three model variants of the running example. The steps for its creation are as follows:

1) Identified features are added as well as their negations. Feature negations are needed to differentiate the actions of the resolution layer given that the negation will be the responsible to remove the model elements in this subtractive strategy.
2) Discovered structural constraints, as presented in Section III-A, are added as propositional logic formulas.
3) Mutual exclusion constraints are added to avoid selecting a feature and its own negation.
4) Configurations, in terms of features in the existing model variants, are added. These configurations are called resolution elements. The resolution elements will trigger the actions on the Base Model to regenerate the existing model variants.

In this example, three resolution elements are created in step 4 correspond to existing variants. However, with the features and constraints shown in Figure 8 as result of the steps 1, 2 and 3, there are eight possible valid resolution elements. Therefore five additional models can be derived using different combinations of feature selections.

Base model: The Base model creation is a realization of an n-way model merge, a technique that has proven to produce better results than pairwise merging [22]. We leverage the feature identification approach with its computation of Interdependent AMEs where the model matching is performed. Figure 5 already showed the hyper-set of all the AMEs from the Model variants. These AMEs are the result of an n-way matching. In order to create the Base Model, we start from the class AME that is marked as the initial resource (i.e. the root) and we automatically construct the Base Model from scratch with the information contained in each AME using: 1) an in-depth tree traversal of the containment dependencies creating the MOF classes and setting their attributes; and 2) a second phase where the references are set to the corresponding classes in the Base Model. In this process, there is neither need to consider if upper bound cardinalities are being violated, nor that there are attributes in the Base Model that are already set. In these cases, as discussed before, the resolution model is responsible for providing information to adjust it at resolution time. Figure 9 shows the Base Model obtained from the variants of our running example.

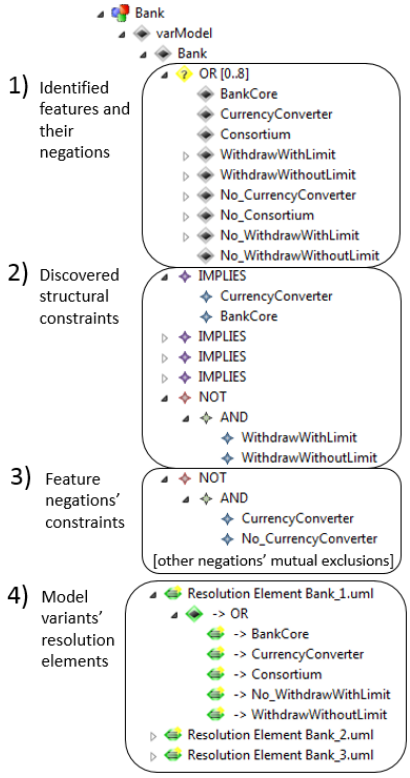


Fig. 8: CVL variability model for the Banking systems

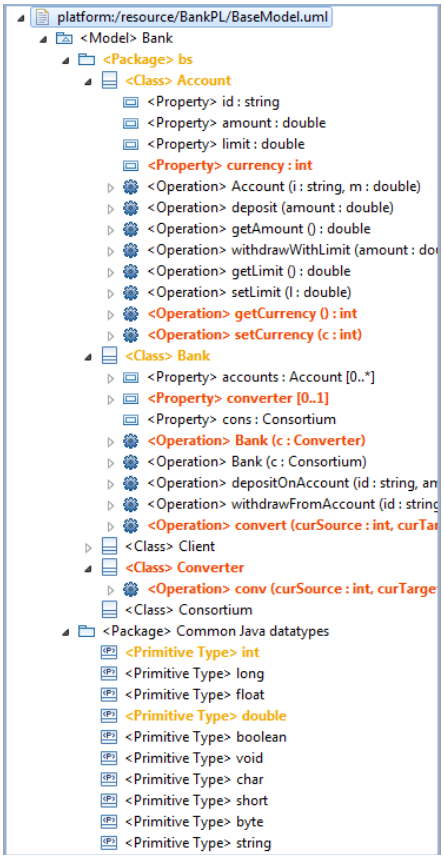


Fig. 9: Base Model for the Banking Systems Product Line

Resolution model: Given that we are following a subtractive strategy, each feature negation will be resolved by removing the feature's corresponding classes. For this, we create three CVL elements for each feature negation: A placement fragment, a replacement fragment and a fragment substitution element. The placement fragment defines the model elements from the Base Model that will be replaced with the replacement fragment. This replacement fragment consists actually of an empty fragment. The fragment substitution element just relates the placement and replacement fragments. Figure 10 shows the overall process to obtain a resolved model.

Figure 11 shows an excerpt of the resolution model for our running example. Notice that in the CVL implementation that we are using [23], the resolution layer is defined inside the variability model itself so the resolution information is contained in each of the features presented before in the variability layer. In the case of `No CurrencyConverter` we can see a Placement fragment that encompasses all the classes related to `CurrencyConverter`. In Figure 9, the classes that are inside the placement fragment of `No CurrencyConverter` are highlighted in dark color. The placement also includes the `FromPlacement`. In the `FromPlacement` we specify all the classes that are referenced from any class of the placement classes and from any class in the contents of the placement classes. These classes are highlighted in light color in Figure 9. This placement information will allow, at resolution time, the removal of the `CurrencyConverter` feature by its substitution with the empty replacement.

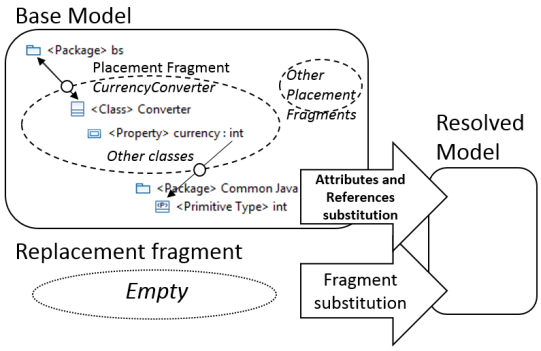


Fig. 10: Subtractive MSPL resolution process with CVL

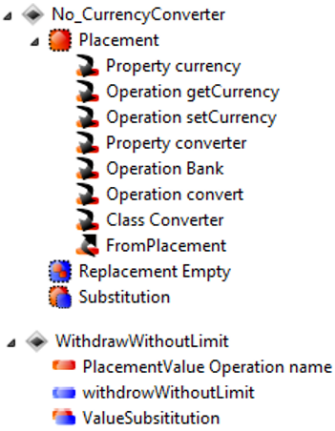


Fig. 11: Excerpt of the CVL resolution model

Regarding attribute and reference AMEs, if the class AME that hosts this AME is not in the same feature, we need to add a placement, replacement and substitution elements to the resolution information of the feature associated to the attribute or reference AME. For example, in the `WithdrawWithoutLimit` feature we add a Placement value in the `name` attribute of the corresponding `Operation class` and we add a replacement value with the String of the value of the attribute AME. At resolution time, if `WithdrawWithoutLimit` is selected, this attribute value will be assigned. For reference AMEs it is the same approach but Object placement, replacement and substitution are used.

IV. Experimental Assessment

In this section we discuss the assessment of MoVa2PL. First, we describe the case studies of real-world systems that we rely on to evaluate our approach. We then present characteristics of the extracted MSPL for the case studies. Finally, we summarize the evaluation of the MoVa2PL where we checked its efficiency to extract an MSPL for large models, and its effectiveness to derive back the existing models and new valid model variants.

A. Case Studies

ArgoUML Case Study: ArgoUML is an open source tool for UML modeling. Variants from this tool were created from its Java codebase by removing specific features [24]. These features are mainly related to the tool support for the edition of different kind of UML diagrams (i.e. Activity, Collaboration, Deployment, Sequence, State and UseCase diagrams). We reverse engineered the source-code of the original ArgoUML and the 6 variants related to diagram edition support as UML models in order to apply MoVa2PL. Concretely, the first column of Table I enumerates these variants. As example, the first one, `ActivityDisabled`, means that this model variant contains all the features related to UML diagrams' edition except Activity diagram. These models contain more than 50K classes. Table I shows the AMEs obtained after MoVa2PL decomposition of the models.

The decomposition in AMEs for the 7 model variants, including the dependencies, took an average of 15 seconds (i.e. around 2 seconds per variant) using a lap-top Dell Latitute E6330 with a processor Inter(R) Core(TM) i73540M CPU @3.00GHz 3.00GHz, 8GB RAM, with Windows 7 and 64-bit Operating System. The Interdependent AMEs identified 41 Blocks and took an average of 7 minutes. Table II shows the result of the identified blocks and their size in terms of AMEs.

We manually analysed the blocks in order to identify the features. The meaning of the blocks from 0 to 6 were easily recognisable by looking at the textual description of its associated AMEs. Block 0 corresponds to the `Core` of ArgoUML, Block 1 to `UseCase` diagrams edition, Block 2 to `Sequence`, Block 3 to `Collaboration`, Block 4 to `State`, Block 5 to `Deployment` and Block 6 to `Activity` diagrams. These blocks were the bigger ones in terms of number of AMEs because the rest of the Blocks only contain very few of them. We manually checked these small blocks and we realized that most of them contain reference AMEs that, in the UML meta-model, are defined as ordered (i.e. the order of the referenced elements is important). The applied

TABLE I: Number of Atomic Model Elements of ArgoUML UML model variants and number dependencies between them

	AMEs	Class	Attr	Ref	Depend
ActivityDisabled	157896	51235	77707	28954	180623
CollabDisabled	158535	51418	78046	29071	181338
DeployDisabled	157314	51033	77450	28831	179949
Original	159771	51820	78667	29284	182738
SequenceDisabled	155231	50349	76417	28465	177646
StateDisabled	156193	50699	76805	28689	178785
UsecaseDisabled	157504	51056	77547	28901	180184

TABLE II: Number of Atomic Model Elements of the blocks identified in the ArgoUML case study

	AMEs	Class	Attribute	Reference
Block 0 -Core	143894	46724	70696	26474
Block 1 -UseCase	2260	760	1117	383
Block 2 -Sequence	4509	1461	2233	815
Block 3 -Collaboration	1204	392	604	208
Block 4 -State	3499	1095	1818	586
Block 5 -Deployment	2457	787	1217	453
Block 6 -Activity	1796	559	916	321
Block 7	0	0	0	1
Block 8	0	0	0	1
⋮	⋮	⋮	⋮	⋮
Block 40	4	2	2	0

matching method takes into consideration the ordering of the referenced elements and therefore considered them as different. This issue, probably introduced by the Java to UML reverse engineering tool, makes more difficult the work of the domain expert that needs to manipulate and analyse these blocks. However, in terms of size of model elements, the main part of the features were successfully identified by MoVa2PL.

Figure 12 shows the graph visualisation of the discovered structural constraints. Concretely, for the ArgoUML case study 45 requires constraints and 13 mutual exclusion constraints were discovered. These discovered constraints reduce considerably the configuration space of the possible feature combinations. Figure 12 shows that the 6 identified features related to the diagrams depend on the `Core` feature. The other nodes of the graph correspond to the blocks that the domain expert will need to analyse. These automatically calculated relations between the blocks can also help during this manual process. For example, those blocks that exclude each other should be related and the scope of analysis is narrowed for those blocks that requires another block that is not the `Core` feature.

Once the constraints are discovered, the CVL MSPL extraction step generates the Variability model, the Base Model and the Realization layer. Figure 13 shows an excerpt of the realization layer related to the `UseCase` and `Sequence` features. We can see the Placement fragment containing the ToPlacement relations. The BaseModel was created as the Maximum model from the model variants. The time for the BaseModel creation took an average of 8 minutes and the Variability and Realization layer took 3 seconds.

While in the beginning we had model variants that only considered disabling one type of diagram each, with the extracted MSPL, we can generate ArgoUML UML model variants with any combination of diagrams. For example, we can derive a UML model that only considers ArgoUML Sequence diagrams edition.

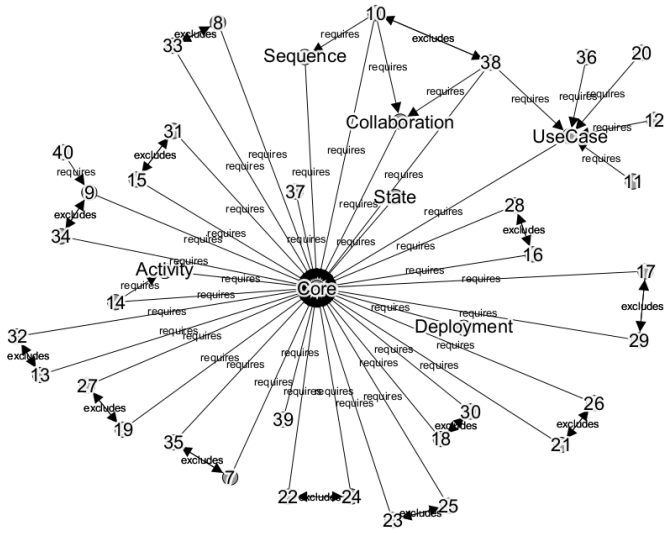


Fig. 12: Structural constraints discovered for the ArgoUML case study

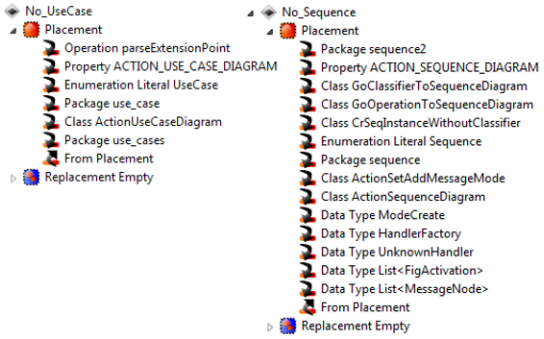


Fig. 13: Excerpt of the CVL Realization layer for the ArgoUML case study

The mentioned problem identified with the ordering of the references highlight the importance of the matching method during MoVa2PL. MoVa2PL is flexible to apply different matching methods. By providing a matching method that ignores the ordering of the references, 18 blocks were identified in the ArgoUML case study from which the first 7 blocks also correspond to the Core and diagrams' features. The trade-off is that, by ignoring the order, we will be assuming that the order was not important and this design decision on the matching policy requires a manual analysis by the domain expert on the models.

In-Flight Entertainment Systems Case Study: We consider the case study of an In-Flight Entertainment (IFE) System. The IFE system is responsible for providing entertainment services for the passengers, including movies, music, internet connection or games during a flight. The IFE system needs also to adjust its behavior in special circumstances (e.g. when the crew wants to communicate with the passengers or during take-off or landing).

For our experiments we consider an IFE system from the Thales Group. Their IFE system was modelled in Capella.[1]

[1]Capella and IFE system model example available at http://polarsys.org/capella/download.html

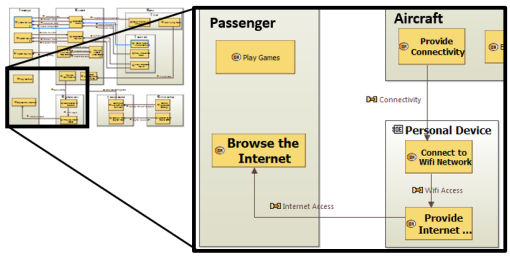


Fig. 14: Operational analysis diagram of the In-Flight Entertainment system variant showing some model elements related to Wi-Fi access for passengers

Capella is a systems engineering modeling tool which implements the Arcadia method for system, software and hardware architectural design [25]. A system modelled with Capella consists of 5 layers. The Operational analysis layer captures the stakeholders, their needs, as well as general information of the system's domain. The System analysis layer formalizes the system requirements. The Logical architecture layer defines how the system will fulfil its requirements. The Physical architecture layer defines how the system will technically be developed and built. Finally, the End-Product Breakdown Structure layer formalizes the component requirements definition to facilitate components' integration, validation, verification and qualification. The domain of Arcadia method is realized and tool-supported in Capella by defining a domain specific language consisting of 17 meta-models. These 17 meta-models are highly linked among themselves and they have a total of 411 meta-classes.

We consider in our study 3 model variants of the IFE system. The first one is the `Original` one and the other two are manually created independently. The second one, called `LowCost1` is a variant that does not include the feature for `Wi-Fi` access for passengers. Figure 14 shows an operational analysis diagram that contains some model elements related to Wi-Fi access. We can see how the Aircraft provides connectivity and the Personal device will allow the Passenger to Browse the internet during the flight. The capability of Wi-Fi access for passengers is propagated to the rest of the model layers such as the System Analysis and the Logical and Physical architecture. The third one, `LowCost2`, does not contain support for `ExteriorVideo` which allows the passengers to watch, in their personal screens, the exterior of the plane at any time during the flight. This variant is intended for aircrafts that may not have cameras outside the plane.

Following the MoVa2PL process as defined in Figure 3, first the IFE model variants were decomposed in AMEs and the dependencies between AMEs inside each variant were calculated. Figure 15 shows the decomposition of the Original IFE System variant. Table III shows the obtained number of AMEs for each variant.

Our Interdependent AMEs method lead to the identification of 3 blocks. The identified blocks where manually analysed and mapped to the features. Table IV summarizes the number of AMEs for the corresponding features.

In the end of this step MoVa2PL automatically discovers the structural constraints among the different features. Thus, we discovered that both `Wi-Fi` and `ExteriorVideo` requires the `Core` feature. With the gathered information the CVL models were automatically generated obtaining a MSPL for the IFE models. From this MSPL we are able to generate a new variant that does not contain `Wi-Fi` nor `ExteriorVideo`.

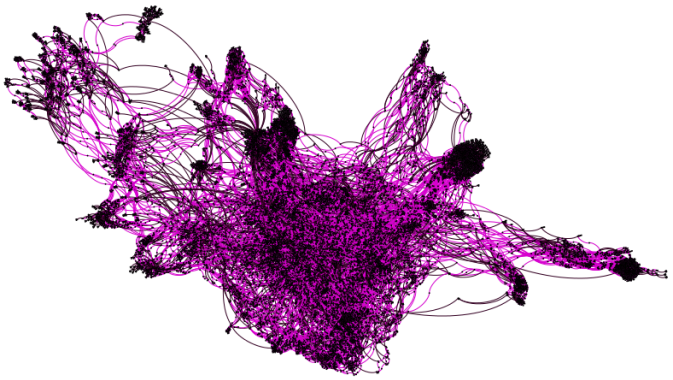


Fig. 15: In-Flight Entertainment system variant decomposition in Atomic Model Elements

TABLE III: Number of Atomic Model Elements of IFE system model variants and number dependencies between them

	AMEs	Class	Attr	Ref	Depend
Original	16624	5345	4081	7198	24205
LowCost1	16551	5321	4066	7164	24098
LowCost2	16594	5335	4075	7184	24161

TABLE IV: Number of Atomic Model Elements of the Features identified from the In-Flight Entertainment model variants

	Total AMEs	Class	Attribute	Reference
Block0 -Core	16521	5311	4060	7150
Block1 -Wi-Fi	73	24	15	34
Block2 -ExteriorVideo	30	10	6	14

B. *Summary evaluation of MoVa2PL*

For evaluating MoVa2PL, based on the case studies outlined above, we focused on checking that, in each case, the extracted MSPL is able not only to 1) re-generate the previously existing variants using our systematic reuse approach but also to 2) generate previously non-existing variants which are structurally valid. By realizing the experiments on different modeling meta-models, namely UML models and the Capella domain specific language, we demonstrated the flexibility and genericity of MoVa2PL as it worked for the two meta-models. We obtained an exact match of the re-generated models and we generated possible non-existing variants (RQ1 and RQ2). We also checked the structural validity manually and we found that we prevent invalid models (RQ1).

V. Limitations and Threats to Validity

Currently, our work presents a number of limitations and includes hypotheses which are threats to validity. We now summarize them before discussing related work.

Feature identification is a challenging task and can be complex for domain experts. Automatically identifying features using heuristics may lead to an output where a given marked

feature is actually a set of different features. This situation is likely when a set of features came always together in all the variants considered. In this case, the Interdependent AMEs approach cannot distinguish among them. Dealing with this kind of issue is out of the scope of the work presented in this paper. We, however, recommend the MoVa2PL approach to be used when in presence of a wide range of variants, increasing the probability that the minimum number of combinations were available to allow distinguishing all features.

In this work we have also eluded the question of semantic validity of the derived model variants. A semantically valid model is a structurally valid model which also makes sense in the domain (i.e. a combination of features that does not violate any semantic rule of the domain). In our work, the extracted MSPL will allow the creation of new models based on combinations of identified features. These new models may be semantically invalid. The challenging issue of assuring some notion of semantic validity has been addressed in other works such as Czarnecky *et al.* [26] or in our previous work proposing visualisation schemes [27].

Finally, MoVa2PL presents some limitation in ensuring the structural validity of models. Indeed, we are not considering constraints that could be defined using the Object Constraints Language (OCL) [26], [28], [29]. Also, the constraints discovery only check requires and excludes constraints between pair of features. More complex structural constraints involving more than two features are not considered.

VI. Related Work

MSPL Reengineering from variants: There are other extractive approaches that study reengineering of MSPLs from model variants. Zhang *et al*. [30] propose the CVL Compare process that combines EMF compare and the CVL framework to analyse a set of model variants and create a preliminary MSPL model. This approach is similar to our MoVa2PL, however, there are three differences: 1) while we automatically calculate the Base Model, CVL Compare relies on the SPL developer to choose it. 2) CVL Compare is based on EMF Compare two-way comparison mechanism. i.e.; model variants are only compared with each other but not simultaneously. 3) CVL Compare do not consider the identification of constraints. As mentioned below, the structural constraints represent the core aspect in our approach to construct valid MSPL.

Rubin [8] and Rubin *et al*. [31] propose the *merge-refactoring* framework. *merge-refactoring* is a formal framework to compare UML model variants using what is referred as n-way Model merging [22]. *Rubin et al.* also propose to refactor the input model variants and create an MSPL using the *compose* operator. While our approach is generic and can be applied to any MOF-based models, the *merge-refactoring* is only applied to UML models. In addition to the reengineering of the MSPL model, the *merge-refactoring* framework propose what is referred to as quality-based metrics that guide the refactoring process to ensure that all refactoring produced by the framework are semantically correct. As we presented in this paper, our approach is based on the structural constraints to generate coherent and correct CVL-compliant MSPLs. Koschke *et al*. [32] propose an automated approach for comparing specific model variants that only concern the static architectural view.

Feature identification: Some existing works only consider feature identification without any support for the reengineering step. Assunção *et al.* present a recent survey on these works [33]. For instance, Ryssel *et al* [34] compare model variants that are represented as function-blocks. Ziadi *et al*. [14] propose an approach to analyse the source code through the use of UML class diagrams of a set of software variants and identify commonality and variability between them. Apart from not covering the reengineering step, these approaches are not generic to any MOF-based scenario. The tool FeatureMapper [35] enables manual and automatic mapping of model elements to features. However, their automatic mapping is based on monitoring the developer while is modeling the realisation of a feature.

Feature constraints discovery: Bosco *et al.* [36] targeted the challenge of generating invalid models from MSPL to be used as counterexamples to refine variability models. Their assumption is that the MSPL exists while we focus on extracting them from variants. The work of Blanc *et al.* [13] also propose an approach for validating sub-models of a monolithic model. In Section V we already discussed Czarnecky *et al.* work regarding OCL constraints [26] as well as our approach to target constraints discovery through a semi-automatic approach using visualisation techniques [27]. These works can be complementary to MoVa2PL structural constraints discovery.

Model comparison and splitting: Model comparison have received a lot attention in the MDE community. However, this is often limited to a comparison between two versions of a model. Stephan *et al*. [37] and Kolovos *et al*. [38] analysed model comparison approaches and provided a classification of them. Model splitting [39] is used to reduce the complexity of monolithic models for human comprehension and team collaboration by hiding part of it. These approaches provide heuristics to modularize the model before splitting. However, these approaches focus in single models, not model variants. We consider that model splitting can be complementary in the feature identification process.

VII. Conclusion

We have presented MoVa2PL as a solution to MSPL adoption from existing model variants. First, the feature identification process considers structural constraints discovery in order to extract a variability model that ensures the validity of the MSPL configuration space. Secondly, a product realization layer is extracted with the information of the AMEs related to each of the features. This realization layer will operate on a base model extracted by an n-way model merge of the variants. We assessed MoVa2PL in two case studies that consider big-medium sized model variants with different meta-models. As further work we want to provide a better support for the semi-automatic step of MoVa2PL that is the analysis and naming of the identified features by using advanced information retrieval techniques.

Acknowledgment

Funded by FNR Luxembourg under the AFR grant agreement 7898764. The work of Tewfik Ziadi was also supported by the University of Luxembourg as a visiting researcher.

REFERENCES

[1] M. Voelter, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. http://dslbook.org.

[2] D. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, February 2006.

[3] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, July/August 2009.

[4] L. M. Northrop, P. C. Clements *et al.*, "A Framework for Software Product Line Practice, Version 5.0," http://www.sei.cmu.edu/productlines/framework.html, 2009.

[5] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 67–82, 2014.

[6] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, A. Cleve, F. Ricca, and M. Cerioli, Eds. IEEE Computer Society, 2013, pp. 25–34.

[7] R. E. Lopez-Herrejon, T. Ziadi, J. Martinez, and A. K. Thurimella, "Second international workshop on reverse variability engineering (REVE 2014)," in *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, S. Gnesi, A. Fantechi, P. Heymans, J. Rubin, K. Czarnecki, and D. Dhungana, Eds. ACM, 2014, p. 354.

[8] J. Rubin, "Cloned product variants: From ad-hoc to well-managed software reuse," Ph.D. dissertation, University of Toronto, 2014.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., 1990.

[10] Ø. Haugen, A. Wasowski, and K. Czarnecki, "CVL: common variability language," in *SPLC 2013*, 2013.

[11] J. Martinez, T. Ziadi, J. Klein, and Y. L. Traon, "Identifying and visualising commonality and variability in model variants," in *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, ser. Lecture Notes in Computer Science, J. Cabot and J. Rubin, Eds., vol. 8569. Springer, 2014, pp. 117–131.

[12] OMG, "Meta-Object Facility (MOF) Core Specification," 2006. [Online]. Available: http://www.omg.org/spec/MOF/2.0/

[13] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 511–520.

[14] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE Computer Society, 2012, pp. 417–422.

[15] T. Ziadi and J. Jézéquel, "Software product line engineering with the UML: deriving products," in *Software Product Lines - Research Issues in Engineering and Management*, T. Käkölä and J. C. Dueñas, Eds. Springer, 2006, pp. 557–588.

[16] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel, *Component-based Product Line Engineering with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[17] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool features and tough decisions: A comparison of variability modeling approaches," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '12. New York, NY, USA: ACM, 2012, pp. 173–182.

[18] S. K. Card, J. D. Mackinlay, and B. Shneiderman, Eds., *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[19] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Bottom-up adoption of software product lines: a generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, 2015, pp. 101–110.

[20] X. Zhang, "Developing model-driven software product lines," Ph.D. dissertation, University of Oslo, Norway, 2014.

[21] G. Perrouin, J. Klein, N. Guelfi, and J. Jézéquel, "Reconciling automation and flexibility in product derivation," in *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, 2008, pp. 339–348.

[22] J. Rubin and M. Chechik, "N-way model merging," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 301–311.

[23] SINTEF, "CVL Tool," 2015. [Online]. Available: http://www.omgwiki.org/variability/doku.php?id=cvl_tool_from_sintef

[24] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 191–200.

[25] Polarsys, "Capella," 2015. [Online]. Available: https://www.polarsys.org/capella/

[26] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, 2006, pp. 211–220.

[27] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Feature relations graphs: A visualisation paradigm for feature constraints in software product lines," in *VISSOFT*, 2014.

[28] OMG, "Object Constraint Language," 2014. [Online]. Available: http://www.omg.org/spec/OCL/

[29] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[30] X. Zhang, Ø. Haugen, and B. Møller-Pedersen, "Model comparison to synthesize a model-driven software product line," in *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, Eds. IEEE, 2011, pp. 90–99.

[31] J. Rubin and M. Chechik, "Combining related products into product lines," in *Fundamental Approaches to Software Engineering, FASE 2012, Tallinn, Estonia, 2012*, 2012.

[32] 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, 2009.

[33] W. K. G. Assunção and S. R. Vergilio, "Feature location for software product line migration: A mapping study," in *Proceedings of the 18th International Software Product Line Conference: Companion Volume 2*, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 52–59.

[34] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Automatic variation-point identification in function-block-based models," in *GPCE*, 2010, pp. 23–32.

[35] F. Heidenreich, J. Kopcsek, and C. Wende, "Featuremapper: mapping features to models," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, 2008, pp. 943–944.

[36] J. B. Ferreira Filho, O. Barais, M. Acher, J. Le Noir, A. Legay, and B. Baudry, "Generating Counterexamples of Model-based Software Product Lines," *Software Tools for Technology Transfer (STTT)*, Jul. 2014.

[37] M. Stephan and J. R. Cordy, "A survey of model comparison approaches and applications," in *MODELSWARD*, 2013, pp. 265–277.

[38] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, ser. CVSM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–6.

[39] D. Struber, J. Rubin, G. Taentzer, and M. Chechik, "Splitting models using information retrieval and model crawling techniques," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, S. Gnesi and A. Rensink, Eds. Springer Berlin Heidelberg, 2014, vol. 8411, pp. 47–62.