Leveraging Variability Modeling to Address Metamodel Revisions in Model-Based Software Product Lines[☆]

Jaime Font[a,b,*], Lorena Arcega[a,b], Øystein Haugen[c], Carlos Cetina[a]

[a] *San Jorge University, SVIT Research Group, Spain*
[b] *University of Oslo, Department of Informatics, Norway*
[c] *Østfold University College, Department of Information Technology, Norway*

Abstract

Metamodels evolve over time, which can break the conformance between the models and the metamodel. Model migration strategies aim to co-evolve models and metamodels together, but their application is currently not fully automatizable and is thus cumbersome and error prone. We introduce the Variable MetaModel (VMM) strategy to address the evolution of the reusable model assets of a model-based Software Product Line. The VMM strategy applies variability modeling ideas to express the evolution of the metamodel in terms of commonalities and variabilities. When the metamodel evolves, changes are automatically formalized into the VMM and models that conform to previous versions of the metamodel continue to conform to the VMM, thus eliminating the need for migration. We have applied both the traditional migration strategy and the VMM strategy to a retrospective case study that includes 13 years of evolution of our industrial partner, an induction hobs manufacturer. The comparison between the two strategies shows better results for the VMM strategy in terms of model indirection, automation, and trust leak.

Keywords: Model-based Software Product Lines, Variability Modeling, Model and Metamodel Co-evolution

1. Introduction

Model-Driven Development aims to shift the focus of software development from coding to modeling. Metamodels are used to formalize a set of concepts and the relationships among those concepts. A model conforms to a metamodel if it is expressed by the terms that are encoded in the metamodel.

Model-based Software Product Lines enable a planned reuse of software components in products that are within the same scope [1]. Commonalities and variabilities among

[☆]This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R).

[*]Principal corresponding author

Email addresses: `jfont@usj.es` (Jaime Font), `larcega@usj.es` (Lorena Arcega), `oystein.haugen@hiof.no` (Øystein Haugen), `ccetina@usj.es` (Carlos Cetina)

the products are formalized into a set of models (and metamodels) using a variability language: either feature models [2, 3] (the de facto standard for variability modeling) or Common Variability Language (CVL) [4], (recommended for adoption as a standard by the Architectural Board of the Object Management Group). Although the details are different, all share the idea of modeling commonalities and variabilities among the different products.

Similar to other software components, metamodels evolve over time [5]; however, changes that are introduced in the new metamodel revision can invalidate the models that conform to the previous revision of the metamodel. To address this issue, migration strategies [6, 7, 8, 9, 10] propose co-evolving models and metamodels together in order to maintain consistency.

However, even though migration strategies have proven to be successful in model-based approaches, their application is not fully automatizable and can be cumbersome and error prone in large systems. Evolution is particularly critical for a successful adoption of model-based Software Product Lines (SPLs) [11].

We believe that the ideas of variability modeling can also be applied at the metamodel level to address the evolution of SPLs and at the same time avoid the issues involved with migration strategies. Our contribution is the Variable MetaModel (VMM) strategy, which enables the evolution of the metamodel without breaking model conformance. In VMM, each metamodel evolution is expressed in terms of metamodel commonalities and variabilities. As a result, already existing models continue to conform to the created VMM, thus eliminating the need for migration and its related issues.

First, we build a retrospective case study of the evolution undergone by our industrial partner (BSH) over the last 13 years regarding the evolution of their models and metamodels. BSH is the leading manufacturer of home appliances in Europe and its induction department produces induction hobs (explained in section 2) following an MDD approach [12].

We then apply a migration strategy to the case study, manually migrating the models (as described in section 4) whenever a metamodel change that breaks the conformance between models and metamodels arises. Migration strategies involve the following three issues: 1) model migration introduces indirection to the models; 2) some of the steps of the migration strategy need human assistance; and 3) the trust gained by models (over years of use) is lost when they are migrated.

Finally, we also apply the VMM strategy to the retrospective case study and compare both strategies (VMM and migration). The comparison shows that the VMM strategy achieves better results than migration in terms of the three issues related to migration: 1) VMM eliminates the need for migration (and the indirections introduced); 2) some of the steps of the migration strategy require human assistance while in the VMM strategy those steps are automatic; 3) the trust gained by models remains the same in the VMM strategy (since the model does not need to change).

This paper is an extended and revised version of our paper published at GPCE 2015 [13]. Apart from revisions throughout the article, in this version we have improved the motivation of the approach and included details about the core operations of the VMM approach (InitVMM and addGen). We have also added some lessons learned from the application of the approach to our industrial partner, information which may be valuable for practitioners that want to apply the ideas of VMM to manage metamodel revisions.

2. Background

This section presents the Domain Specific Language (DSL) used by our industrial partner to formalize their products, the IHDSL. It will be used throughout the rest of the paper to present a running example. Then, the Common Variability Language (CVL) is presented. CVL is the language used by our VMM approach to formalize the differences between metamodel revisions.

2.1. The Induction Hobs Domain

Induction cookers or hobs use electromagnetism to generate heat that is transferred to the cookware. Traditionally, stoves feature four rounded areas that become hot when turned on. Therefore, the first Induction Hobs (IHs) created provided similar capabilities. However, the induction hob domain is constantly evolving due to the possibilities provided by the induction phenomena and the programmable microcontrollers that are present in the IHs.

For instance, the newest IHs feature full cooking surfaces, where dynamic heating areas are automatically calculated and activated or deactivated depending on the shape, size, and position of the cookware placed on top. There has been an increase in the type of feedback provided to the user while cooking, such as the exact temperature of the cookware, the temperature of the food being cooked, or real-time measurements of the consumption of the IH.

The Domain Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 74 references among the meta-classes, and more than 180 properties. However, in order to gain legibility and due to intellectual property rights, in this paper we use a simplified subset of the IHDSL (the top-left corner of Figure 2 shows the metamodel for this DSL).

The bottom-right corner of Figure 1 shows an Induction Hob (Resolved Induction Hob) with the graphical representation of the IHDSL. It is composed of two power modules (vertical rectangles on both sides of the IH). Each of them holds two inverters (squares), which are in charge of providing the electrical supply required to generate the magnetic field. Inverters are connected to the inductors (circles), which are the elements where the magnetic field is generated. The number inside each inductor represents the diameter of the inductor. The line that connects inverters and inductors represents the power channel, which transfers energy from the inverter to the inductor. The user interface of an IH has controllers to configure the power level of each inductor (the horizontal rectangle at the bottom of the IH).

2.2. The Common Variability Language (CVL)

The Common Variability Language (CVL) is a DSL for modeling variability in any model of any DSL based on Meta-Object Facility (MOF, the OMG specification to define a universal metamodel for describing modeling languages). CVL defines variants of the base model by replacing parts of the base model with model replacements that are found in a library of models. The base model and the library of models must be specified using the same DSL.

Despite the fact that CVL is currently frozen by intellectual property rights issues, the proposed ideas have been recommended for adoption by the architectural board of the OMG and we decided to use it for technical reasons (given its expressiveness for realizing

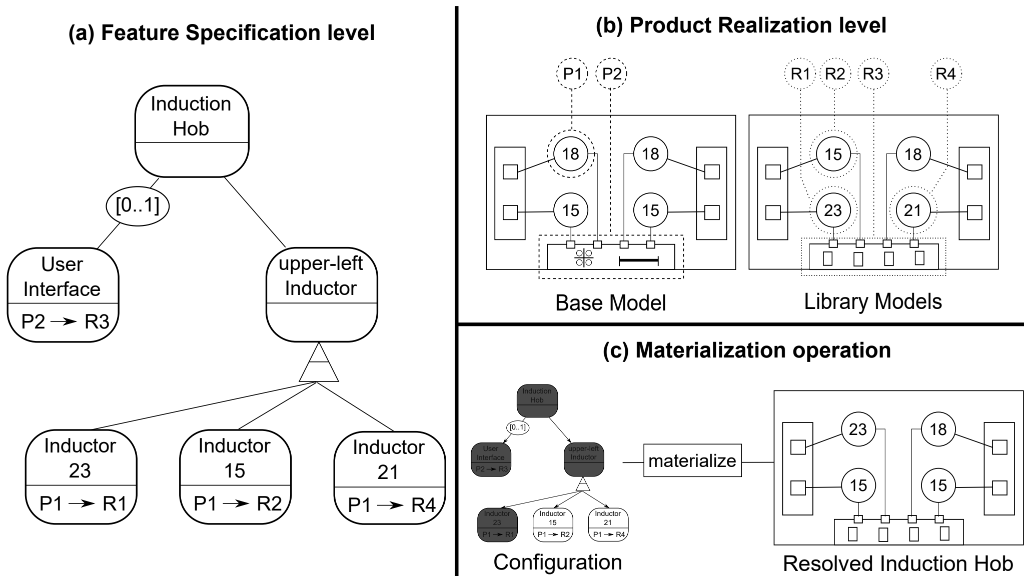

Figure 1: CVL applied to IH-DSL

the variability specification over models). However, any other variability specification mechanism with capabilities similar to the ones of CVL could be used to materialize our approach.

The variability specification in CVL is divided across two different levels[1]: the feature specification level (where variability is specified following a feature model syntax [2, 3]); and the product realization level (where the variability specified in terms of features is linked to the actual models in the form of variation points, replacements, and substitutions). Finally, the materialization operation takes a configuration of the feature specification and executes the substitutions defined by the product realization level to obtain a new variant.

Figure 1 shows an example of specification of variability through CVL applied to the induction hobs models. Figure 1 (a) shows the feature specification level, Figure 1 (b) shows the product realization level, and Figure 1 (c) shows the materialization operation.

In Figure 1 (a), the bubbles represent features which are organized hierarchically. Some features can be optional (denoted by the range [0..1] like the *User Interface*). In addition, an OR operation between multiple choices of features can be defined (denoted by the triangle above the three inductors features, which are children of the *upper-left inductor* feature). Some features are only used to improve the readability and structure of the tree (e.g., *the upper-left inductor*) while others are linked with the substitutions (defined in the product realization level) that must be performed when materializing those products.

In Figure 1 (b), the substitutions of variation points by replacements are defined. The **base model** is a model described by a given DSL (here, IHDSL) that serves as the

[1]layers in the CVL terminology

base for different model variants defined over it. The elements of the base model that are subject to variations are the placement fragments in CVL; however, we will refer to them with the more common term **variation points** (hereinafter VPs). It is important to remark that the term VP is also used in CVL and its definition includes not only the part of the model subject to variations but also the set of elements that can be used to substitute them. Therefore, the reader used to CVL terminology needs to be aware of the use of VP instead of placement in the context of this paper.

In the base model, there are two VPs defined over it: P1, which includes the top-left inductor; and P2, which includes the user interface. To define replacements for a VP, CVL uses a library of models described in the same DSL as the Base Model. Each alternative for a VP in the Base Model is a **replacement** fragment (hereinafter replacement). In the library model, there are four replacements defined: three inductor replacements (R1, R2, and R4) and a user interface replacement (R3).

Notice the correspondence between the substitutions linked to the features from the feature specification level in Figure 1 (a) and the VP and replacements defined in the product realization level in Figure 1 (b).

CVL defines variants of the base model by means of **fragment substitutions** (i.e., the substitution of VPs by replacements). Figure 1 (a) shows the Feature specification level with the substitutions linked to some features. First, the *Induction Hob* feature is the root and must be present in all IHs. Then, we can have buttons for the *User Interface* substituting P2 by R3 (this substitution is optional). We also have different alternatives for the *upper-left inductor* (sizes 23, 15, or 21) substituting P1 by R1, R2, or R4.

In order to specify a product model from the set of models described by the feature specification model, a **configuration**[2] of the feature specification model is used. In other words, the choices about the features that will or will not be included in the product are resolved by the user. Figure 1 (c) shows the materialization operation of CVL, which executes the substitutions that are linked to the features present in a configuration and produces a variant of the base model where the VPs have been substituted by the replacements selected. As a result of the materialization operation, a resolved induction hob is produced where P1 has been substituted by R1 and P2 has been substituted by R3 (following features from the configuration and the linked substitutions).

When VPs are defined through CVL, there are two characteristics that make the language really flexible: recursion and cardinality. Recursion enables the definition of VPs inside other VPs in a recursive manner. The cardinality enables the substitution of one VP by several replacements at the same time. CVL is currently frozen by non-technical reasons, the recursion and cardinality capabilities when defining VPs are leveraged by our approach, and, therefore, CVL is used by our approach as the variability specification language.

For simplicity throughout the rest of the paper, we will show the VPs superimposed on the base model, even though they are defined in a separate model. Also, the replacements defined in the library models will be shown separately from the rest of the model where they are defined.

[2]resolution model in the CVL terminology

3. SPL Evolution Formalized by CVL

This section presents the retrospective case study that was extracted from the evolution of our industrial partner's models and metamodels over the last 13 years. Although the evolution data provided involves all the elements present in the initial DSL, for simplicity and due to intellectual property rights, we are going to focus on the evolution related to the inductor concept.

Let MM be the set of all models that conform to the MOF language (i.e., the set of all metamodels) and let M be the set of all models. Let m_i be in M and let mm_i be in MM. Then, we say that a model (m_i) conforms to a metamodel (mm_i) if it is expressed by the terms that are encoded in the metamodel; this conformance is denoted as $C(m_i, mm_i)$.

Let $CVLSPL$ be the set of all CVL-based product lines. One such product line, $cvlspl_i$, is denoted as follows:

$$\begin{aligned} CVLSPL = &\ MM \times M \times M \\ cvlspl_i = &< mm_i\ ,\ b_i\ ,\ l_i > \end{aligned} \tag{1}$$

where mm_i is the metamodel of the DSL (conforming to MOF), b_i is the base model (over which VPs for the variable parts are defined), l_i is the library of replacements for those VPs, and the conformance between models $C(b_i, mm_i)$ and $C(l_i, mm_i)$ is fulfilled. In addition, let i be a consecutive index that is assigned based on when models and metamodels are created, i.e., we will refer to the *generation* i of the base model, the *generation* i of the metamodel, the *generation* i of the library, and the *generation* i of the CVLSPL. The use of the index i is only as an annotation to identify the generation. For each generation there may be several base models and libraries adhering to the same metamodel.

We perform a $CVLSPL$ evolution (a shift from one $cvlspl_i$ generation to the next generation, $cvlspl_{i+1}$) whenever there is a *breaking and unresolvable change* (hereinafter *breaking change*) [9] in the metamodel. Breaking changes break the conformance of models and the metamodel in a way that cannot be resolved by automatic means [9] (e.g., the addition of a mandatory meta-class or a restriction in the multiplicities). There are other metamodel changes that do not break the conformance of models and the metamodel (e.g., the addition of an optional class) or metamodel changes that can be resolved automatically by existing approaches [7, 8, 9, 10, 6] (e.g., eliminating a property). However, in this work we will focus on the evolution triggered by breaking changes.

Figure 2 shows a summary of the CVLSPL generations and the evolutions performed. Specifically, we present three CVLSPL generations: the first row shows $cvlspl_1$, which includes the concept of inductor; the second row shows $cvlspl_2$, which includes the concept of Hotplate; and the third row shows $cvlspl_3$, which includes the concept of cooking zone. This figure shows the breaking changes that were overcome by our industrial partner, such as the addition or removal of meta-elements. The first column shows the metamodel for each generation, the second column shows the base model, and the third column shows the replacements library. The full variability specification is not shown, but the shape of each VP in the base model and shape of each replacement in the replacements library are indicators of what VPs can be substituted by what replacements.

Evolution 1 (from $cvlspl_1$ to $cvlspl_2$) is triggered by a new concept called Hotplate (see the first and second rows of Figure 2). A Hotplate consists of a group of inductors

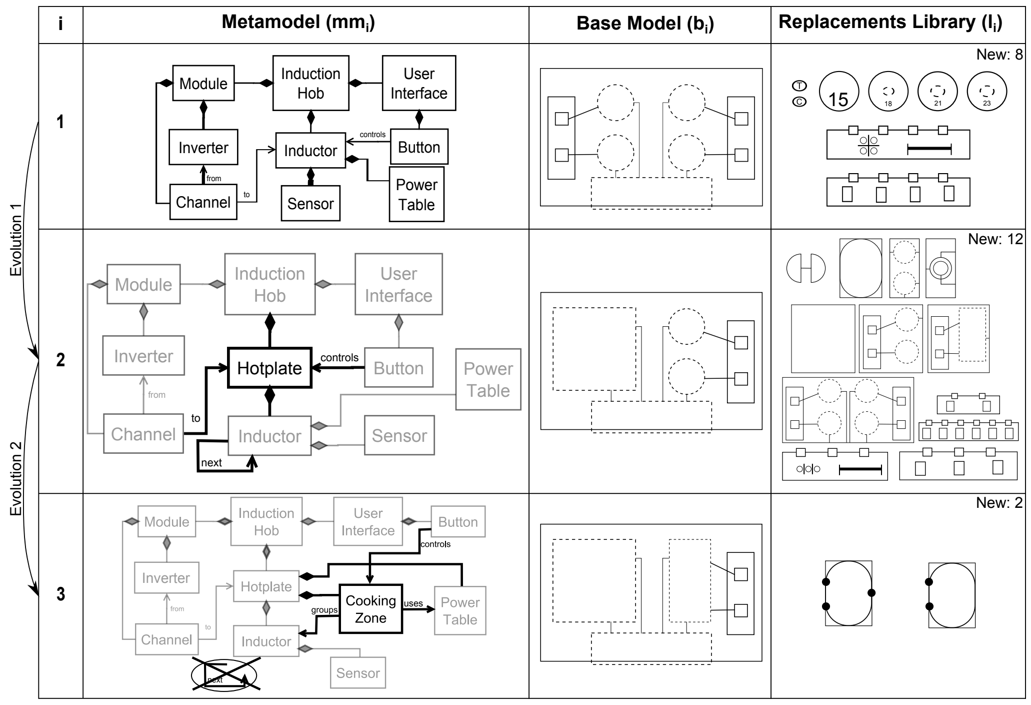

Figure 2: Model Generations of the CVLSPL

that can work together. There is a hierarchy (*next* relationship) among the inductors; some must be turned on before their subordinates are turned on. Since we need to control the whole Hotplate (two inductors) with just one user interface controller, the controller will now act over hotplates instead of inductors. This is reflected in the metamodel mm_2 (see the second row, first column).

There are also modifications at the model level. A new VP is created over the base model b_2 to enable substitutions of the new hotplate replacements. In addition, new replacements (l_2) that instantiate the hotplate concept are created; for example, the split hotplate (formed by two inductors, one main and one auxiliary) or the double hotplate (formed by two inductors, requiring twice the space and power as the rest of hotplates).

Evolution 2 (from $cvlspl_2$ to $cvlspl_3$) is triggered by a new concept called cooking zone (see the second and third rows of Figure 2). Cooking zones improve the hotplate by introducing the ability to heat two different pieces of cookware at the same time and with different power levels. Now each hotplate will have cooking zones, which will be controlled by the user interface controller. Since the number of combinations of inductors that are working at the same time increases, the power table is now aggregated by the hotplate, and the cooking zones use it. By means of this modification, several hotplates will share the same power tables (when the inductor configurations are equivalent). Furthermore, the hierarchy that is present among inductors is now controlled by the cooking zone (one cooking zone having the main inductor and another cooking zone having both inductors);

therefore, the relationship *next* is removed from the metamodel (mm_3).

A new VP to include hotplates on both sides is created over the base model b_3. Similarly, new replacements that exercise the new concept of cooking zone are created (l_3). For instance, the pool hotplate has four inductors that are divided into two different cooking zones, which are controlled by two different buttons.

As any other model change, the evolutions presented so far can be seen as graph transformations. For instance, the changes could be expressed as extensions and derivations like it is done in the work of Gonçalves et al. [14]. Although the presented changes could be formalized using graph transformations, the changes come from pragmatic development, and graph rewrites have not been the basis for the developments.

Sections 4 and 5 show the application of both Migration and VMM strategies to address the evolution presented in this section.

4. Motivation of the Approach

The evolution presented in Section 3 needs to be properly supported by the metamodels that are used by our industrial partner to formalize their SPL. Some of the changes presented can be addressed without breaking the conformance between the models and the metamodel, such as the creation of new model fragments or the addition of new optional elements to the metamodel. However, when we perform a breaking change to the metamodel (e.g., the hotplate and cooking zone concepts), the conformance between the models and the metamodel is lost.

Traditional migration strategies [7, 8, 9, 10, 6] propose migrating all of the models to conform to the new version of the metamodel. Given a metamodel change, the migration of the SPL can be achieved by the following steps: 1) the metamodel is upgraded to a new version introducing the new concept; 2) a model-to-model (M2M) transformation that migrates models from one version to another is created (manual specification [6], operator-based [7, 8], or metamodel matching [9, 10]); 3) existing replacements are migrated (by executing the M2M transformation obtained from Step 2) to conform to the new generation of the metamodel; 4) if some common parts that are present in the base model have become variable, the user creates VPs over the base model and extracts the model fragments as replacements; and 5) new replacements are created to instantiate the new concepts that have been incorporated into the metamodel.

Let E_{mig} be the operation used to evolve a $cvlspl_i$ from a given generation i to the next generation $(i+1)$ following the migration strategy. The operation is defined as follows:

$$\begin{array}{lcc} E_{mig}: \quad CVLSPL & \longrightarrow & CVLSPL \\ E_{mig}\,(< mm_i, b_i, l_i >) & = & < mm_{i+1}, b_{i+1}, l_{i+1} > \\ & & \text{where} \quad M2M(L_i) = L_{i+1} \end{array} \tag{2}$$

Figure 3 shows three *CVLSPL generations* that are managed using the migration strategy. We start from $cvlspl_1$, which is shown in the first column. The square depicts the metamodel (mm_1) (which includes the inductor concept), the diamond depicts the base model (b_1), and the circle depicts the library of replacements (l_1). Both b_1 and l_1 conform to mm_1.

Given a breaking change (hotplate concept), we perform the five steps of the migration strategy. In Step 1 (Figure 3, column one), we modify the metamodel to include the new

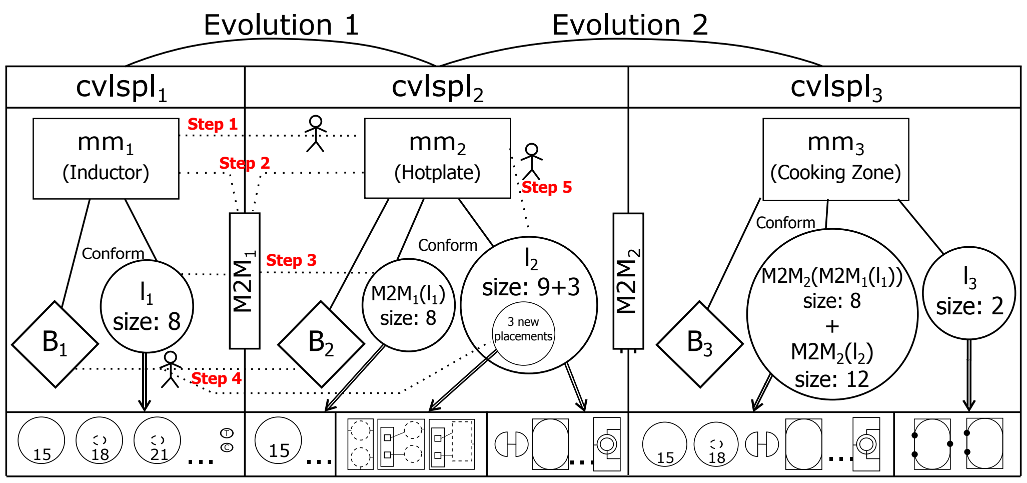

Figure 3: The Steps of the Migration Strategy

concept. Then, in Step 2, we define the $M2M_1$ transformation, which migrates models that conform to mm_1 into models that conform to mm_2. In Step 3, we apply the $M2M_1$ transformation to the library of replacements l_1 (8 replacements). This step requires assistance on the part of the user to apply the transformation associated to breaking changes. Step 4 consists of the extraction of common parts from the base model b_1 that have turned into variables parts (as required by the new generation). Since the changes of $cvlspl_2$ include bigger hotplates in the form of new replacements that cannot be placed in existing VPs, we create new VPs and substitutions over the base model b_2. In addition, we extract those old VPs from the base model and include them in new replacements (three new replacements in l_2). Finally, in Step 5, nine new replacements that instantiate the new concept are created in l_2.

We have created the $cvlspl_2$ of the SPL by following the above steps. We follow the same steps to create the $cvlspl_3$ with this strategy . This time, when the metamodel is edited (Step 1), the relationship *next*, (created in generation 2), is eliminated since its functionality is now provided by the cooking zone. Note that this time we need to migrate both replacement libraries, the library previously migrated from $cvlspl_1$ (8 migrated replacements) and the library created during $cvlspl_2$ generation (9 + 3 new replacements).

Figure 4 presents the evolution of a model fragment following a migration strategy. Each column shows the same fragment (Inductor 15) for each of the $cvlspl_i$ generations. Although its functionality remains the same, the model is augmented to conform to each generation metamodel. In **generation 1**, the replacement of an inductor of size 15 is represented by 2 metamodel classes (Inductor and Power Table) and can be connected to a channel and controlled by a button. In **generation 2**, the model fragment is migrated to conform to mm_2. Hotplate 1 now aggregates the inductor and is the one controlled by the button. In this generation, we need 3 classes (we add the Hotplate) to model the same functionality. In **generation 3**, we need to include a cooking zone (enabling groups inside the same hotplate), so the model is now composed of four model elements. The three versions of the model fragment represent the same functionality: a heating element of size 15 that is connected with a channel and controlled from a button. However, there is an increase in model complexity.

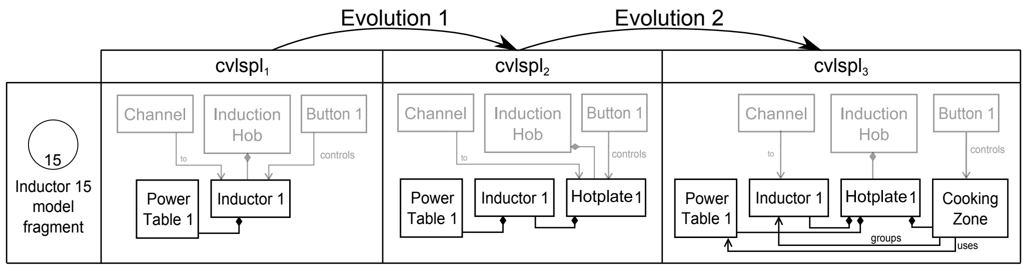

Figure 4: Evolution of Model Fragment through Migrations

Specifically, the migration of models from our industrial partner involves three related issues: 1) **indirection**, where there is an increase in the number of elements used to model the same element of the induction hob (as in this example); 2) **automation**, since the migration of the models cannot be performed automatically, an engineer needs to generate the M2M transformation and make decisions when applying it; and 3) **trust leak**, the modification of the model fragments (through the migrations) decreases the trust gained by those models during that generation. The fragments need to be modified to be adapted to the new metamodel (not to improve its functionality), and the modification is regarded as unnecessary and error prone. The domain of our industrial partner is constantly evolving, but the original elements are still present in new IHs. New kinds of heating elements or strategies may appear, but the simplest inductors (e.g., the inductor of size 15) are still an important part of modern IHs.

5. The Variable MetaModel (VMM)

In order to eliminate the need for migration when a new generation (metamodel revision) is created by the engineers, a new metamodel that supports both generations is automatically built: the Variable MetaModel (VMM). For instance, models that conform to generation 1 and models that conform to generation 2 will also conform to this VMM. A model that contains replacements from both generations will conform to the VMM. Since the VMM will be enhanced each time a new generation is created, a single VMM that includes all the generations of the CVLSPL will exist.

The VMM is the result of applying CVL at the metamodel level; we have a base model in a given DSL (in this case, MOF) with VPs defined over it and a library of replacements. VMM is defined as follows:

$$\begin{aligned} VMM &= MM \times MM \\ vmm_i &= < mmb_i , mml_i > \end{aligned} \tag{3}$$

where mmb_i is the base model at the metamodel level and mml_i is the library of replacements at the metamodel level.

The vmm_i hold all metamodel variations from starting generation (generation 1) to generation i. Similarly to CVL at the model level, we can materialize models that

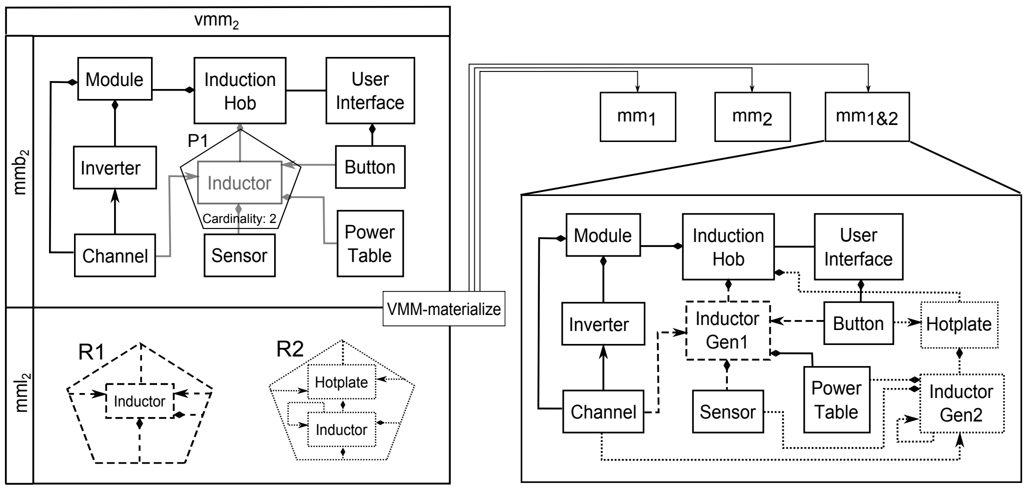

Figure 5: VMM and VMM-materialize

conform to the given DSL (in this case, MOF). Let G be the set of all generations and let $\mathcal{P}(G)$ be its power set. We define the $VMMmat$ (VMM Materialization) operation as follows:

$$\begin{aligned} VMMmat\colon \quad & VMM \quad \times\mathcal{P}(G) \longrightarrow MM \\ VMMmat(<mmb_i, mml_i>, \quad & g) \quad = mm_g \\ & where\ g \neq \emptyset \end{aligned} \tag{4}$$

That is, given a vmm_i where i generation is included in G and selecting a non-empty generation set g, $VMMmat$ retrieves the mm_g for the $cvlspl_g$ of the given generation set g.

Figure 5 (left) shows an example of VMM, the vmm_2 for generation 2. The top-left corner shows the base model (mmb_2). It is the metamodel from $cvlspl_1$, with a VP (P1) defined over the inductor. The bottom-left corner of Figure 5 shows the replacement library (mml_2), which contains two different replacements: R1 (in dashed lines) defined over the $cvlspl_1$ metamodel and R2 (in dotted lines) defined over the $cvlspl_2$ metamodel.

Figure 5 (right) shows the models produced with the vmm_2 presented. The materialization of CVL produces models that conform to the same language that the base model and replacements conform to; therefore, in this case the models produced will conform to MOF. With the library that is available (two replacements), we can produce three different models: 1) mm_1 (the metamodel of $cvlspl_1$) with a substitution of P1 by R1; 2) mm_2 (the metamodel of $cvlspl_2$) with a substitution of P1 by R2; and 3) $mm_{1\&2}$ (a new metamodel with the concepts from the mm_1 and the mm_2 metamodels) with the substitution of P1 by R1 and P1 by R2.

The cardinality property of VPs in CVL enables the creation of $mm_{1\&2}$. In other words, one VP can be substituted more than one time (the number of times can be specified). The first time that a VP is substituted, the existing references of the VP are replaced. The second time that the same VP is substituted, new references that are analogous to the existing ones need to be created. In Figure 5, the aggregation of

Inductors in vmm_2 is duplicated into an aggregation of Inductor Gen1 (in dashed lines), and an aggregation of Hotplate (in dotted lines) in the $mm_{1\&2}$. We have limited the substitution of the same replacement several times as the result will be the same as replacing it only once ($mm_{1\&1}$ produces the same metamodel as mm_1).

The $mm_{1\&2}$ metamodel contains concepts from both $cvlspl_1$ and $cvlspl_2$ at the same time. To achieve this, VMM renames the elements that conflict (e.g., Inductor from mm_1 and from mm_2). The advantages of this $mm_{1\&2}$ is that any model that conforms to mm_1 also conforms to $mm_{1\&2}$ and any model that conforms to mm_2 also conforms to $mm_{1\&2}$. In other words, $mm_{1\&2}$ is used when materializing IH models that contain replacements from both libraries (l_1 and l_2), and the resulting model conforms to $mm_{1\&2}$. When combining replacements from different generations into the same product, unexpected interactions between them might arise. However, dealing with feature interactions is not straightforward and there are several works focusing on this topic (such as [15]); thus, feature interactions will be left out of the scope of this paper.

The vmm_2 enables the materialization of mm_1 and mm_2 that are used directly by the engineers to create new replacements. By doing so, the replacements created will conform to a specific generation and will not include unnecessary indirection. If the functionality required for a particular replacement can be achieved with the expressiveness of a previous generation, that metamodel will be used.

Furthermore, if the engineers try to create new replacements using the $mm_{1\&2}$ directly, they could end up creating models that do not conform to either mm_1 or to mm_2. Therefore, we need to keep the original metamodels (mm_1 and mm_2) in order to enable the creation of new replacements.

6. VMM operations

There are two main operations in relation to the VMM: the initialization of the VMM and the addition of new metamodel revisions. Both operations are capable of spotting the commonalities and variabilities among metamodels and formalizing them in terms of CVL. The initialization is executed only one time, to generate the initial VMM. Then, the addition of new metamodel revisions is performed each time a new revision is created. The following subsections present both operations in detail.

6.1. InitVMM operation

Figure 6 shows an example of the initVMM operation. InitVMM receives two metamodel revisions (e.g., Metamodel A and Metamodel B) as input and produces a VMM that includes both generations as output. Either of the two metamodels can be used as the base model and will lead to valid CVL specification of the metamodels provided. Different base models result in different model fragments, which are used to specify the variability. This can be highly relevant when there are users that interact directly with the model fragments [12], but it is not important for the VMM approach since those model fragments will be managed automatically. Therefore, one revision is randomly selected as the base model (in this example, Metamodel A); we will refer to the other metamodel revision as the new revision (in this example, Metamodel B).

Then, the operation follows a five-step process to generate the VMM. The aim of this process is to formalize the commonalities and particularities of each metamodel revision

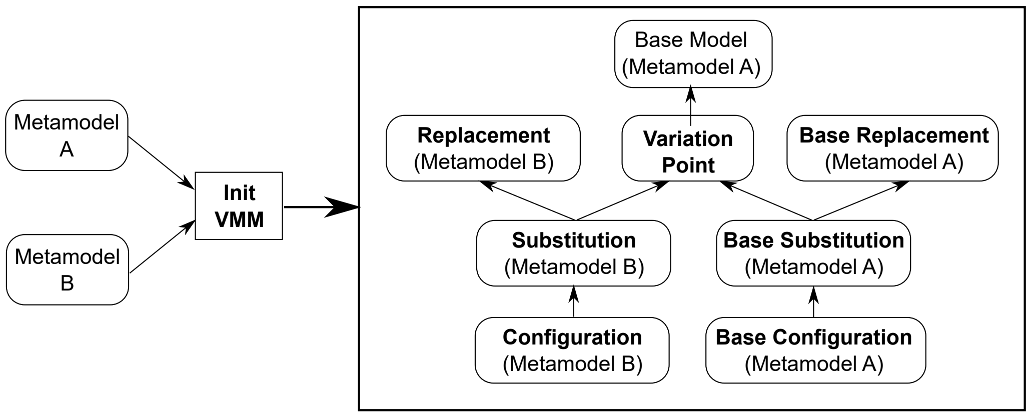
Metamodel A
Metamodel B
Init VMM
Base Model (Metamodel A)
Replacement (Metamodel B)
Variation Point
Base Replacement (Metamodel A)
Substitution (Metamodel B)
Base Substitution (Metamodel A)
Configuration (Metamodel B)
Base Configuration (Metamodel A)

Figure 6: InitVMM operation

in terms of CVL (VPs, replacements, substitutions, and configurations). To do this, the operation will perform comparisons between the Base Model (Metamodel A) and the new revision provided as input (Metamodel B):

1. **Compare**: Metamodel B is automatically compared with the base model. The result is a list of differences between the two revisions. Each difference is composed of two elements (the differing element from the base model and the differing element from the new revision (Metamodel B)). Then, each of the differences is processed and formalized as CVL elements as follows:
   (a) **Variation Point**: A VP is created over the base model (if that VP does not previously exist). Using the information from the comparison, the boundaries of the VP are generated accordingly.
   (b) **Replacement**: A replacement that formalizes the differences between the base model and the new revision must be created. A replacement holding the particularities of Metamodel B is created; this replacement will turn the base model into the Metamodel B. As with the VP, we use the information from the comparison to determine the boundaries of the replacement.
   (c) **Substitution**: Once a VP and a replacement have been created the process generates a substitution. That is, the boundaries of the VP and the replacement are mapped accordingly, so the VP can be substituted by the replacement.
2. **Configuration**: The process is repeated for all of the differences obtained in Step 1. Finally, a configuration is defined, specifying what substitutions need to be executed to turn the base model (Metamodel A) into the new metamodel (Metamodel B).

As a result of this process, the commonalities and variabilities among the new revision (Metamodel B) and the base model (Metamodel A) are formalized in terms of CVL and thus, there are replacements holding the particularities of the new revision. However, the VMM also needs to capture the particularities of the metamodel revision that is used as the base model in separate fragments. Therefore, each time a new VP is generated over the base model, Steps 1.(b), 1.(c) and 2. will be replicated to generate the CVL specification for the metamodel revision that is used as the base model:

(b) **Base replacement**: The process need a replacement that formalizes the particularities of the base model. Therefore, all the elements included in the VP will be included in a new replacement. This replacement holds the particularities of Metamodel A and will be necessary to generate combined metamodels (joining two revisions).

(c) **Base substitution**: As previously, we need to map the VP and the replacement boundaries so that the substitution can be properly executed. The execution of this substitution might seem unnecessary since the result would be the same base model (in this example, Metamodel A); however, the replacement and substitutions generated will be necessary when generating metamodel revisions that make use of different generations.

2. **Base configuration**: Finally, a new configuration describing the substitutions needed to generate that revision from the base model is generated. Again, this may seem redundant, but it is done this way to keep the consistency and explicitly formalize which replacements and substitutions belong to that particular revision (in spite of whether or not that revision is used as the base model).

In summary, the initVMM operation formalizes a metamodel revision in terms of CVL, generating VPs, replacements, substitutions, and configurations as needed. The first time it is executed, it also formalizes the base model in terms of CVL, so all of the metamodel revisions are formalized in terms of CVL independently of the revision used as the base model.

Figure 5 (left) shows an example of the result of initVMM applied to the induction hobs. Two different revisions, mm_1 and mm_2, were used as input. Then, mm_1 was selected as the base model (mmb_2), a new VP was created (P1), and then two replacements (mml_2) were generated to formalize the particularities of each revision (R1 to formalize mm_1 and R2 to formalize mm_2). In addition, the cardinality of the VP was updated accordingly as there were two substitutions using that VP (the configurations do not have a graphical representation in the figure).

6.2. AddGen operation

Once the VMM has been created, following the initVMM operation, it is necessary to have an operation to include new metamodel revisions in this VMM. This is accomplished by the addGen operation. The operation receives a VMM and a new metamodel revision as input and returns an extended VMM that includes the new revision.

The operation proceeds similarly to the initVMM operation; however, this time there is only one metamodel that will be compared with the base model (it is not necessary to capture the base model as separate fragments). Furthermore, the addition of new metamodel revisions can result in the reutilization of already existing VPs. TIn other words, when creating a new VP as part of Step 1.(b) Variation Point, the VP may already exist and there is no need to create a new one. The same VP will be used and its multiplicity will be increased. By doing so, we will enable the materialization of models that combine several generations.

As a result of this operation, new VPs, replacements, substitutions, and configurations are automatically created to formalize the new metamodel revision. The resulting VMM will now include the new metamodel and it will be possible to materialize it as a single generation metamodel or as part of a metamodel that combines several generations.

Both operations (InitVMM and AddGen) are automatic processes and there is no need for human assistance to run them. The first time that a new metamodel revision is generated, initVMM will be executed and the following times, addGen will be executed. The comparisons performed by the operations have been implemented based on the EMF Compare Framework [16]. This framework provides functionality to compare EMF (the implementation of MOF within the Eclipse environment) models and can be customized to perform the comparisons based on different criteria. In our case, we compared models at the finest level of granularity capturing any change from one revision to the next (e.g., the addition of a property or even a change in the name of a class).

7. Application of the VMM approach

The previous sections have presented the VMM approach and the operations needed to materialize it. In this section, we describe how to apply the VMM strategy to manage the metamodel revisions of a model-based SPL. Section 7.1 presents the steps necessary to build new generations. Section 7.2 shows the resulting VMM when applied to our case study. Finally, Section 7.3 shows the usage of the resulting SPL to derive new products from the SPL.

7.1. The Steps of the VMM Strategy

This section presents the steps performed to include a new CVLSPL generation into the VMM. Some steps need to be manually performed by the engineer while others are performed automatically by applying the operations described previously. In this example we show the inclusion of the first generation of CVLSPL, so the initVMM operation will be used (the addition of further generations follow the same steps). The evolution of a $cvlspl_i$ following the VMM-strategy is denoted as follows:

$$\begin{array}{ll} E_{VMM}: & CVLSPL \longrightarrow \quad VMM \\ & < mm_i, b_i, l_i > \longrightarrow < mmb_{i+1}, mml_{i+1} > \end{array} \quad (5)$$

$VMMmat$ is used with the generated vmm_i to retrieve the different CVLSPL generations needed by the company.

Figure 7 shows the method for performing Evolution 1 from $cvlspl_1$ to $cvlspl_2$. Each of the columns of the tables represents one step in the application of the VMM strategy. The top part shows $cvlspl_1$ with its base model b_1 (depicted as a diamond), its metamodel mm_1 (depicted as a rectangle), and its fragment library l_i (depicted as a circle). The bottom part shows $cvlspl_2$: the first row shows vmm_2, and the second row shows $cvlspl_2$.

Step 1 shows the edition of the metamodel by the user. The mm_1 metamodel is edited to include the new concepts of the next generation, resulting in the mm_2 metamodel. In this example the engineer has modified the mm_1 (see first row first column of Figure 2) into the mm_2 (see second row first column of Figure 2), including the concept of Hotplate.

Step 2 shows our initVMM operation, which is used to spot the differences between the two metamodels, to describe them in terms of a base model and replacements, and to initialize the VMM. The common parts of the two metamodels (mm_1 and mm_2) are included in the mmb_2 and VPs are created over it for the differences between mm_1 and mm_2. Furthermore, replacements that contain these differences are created and included

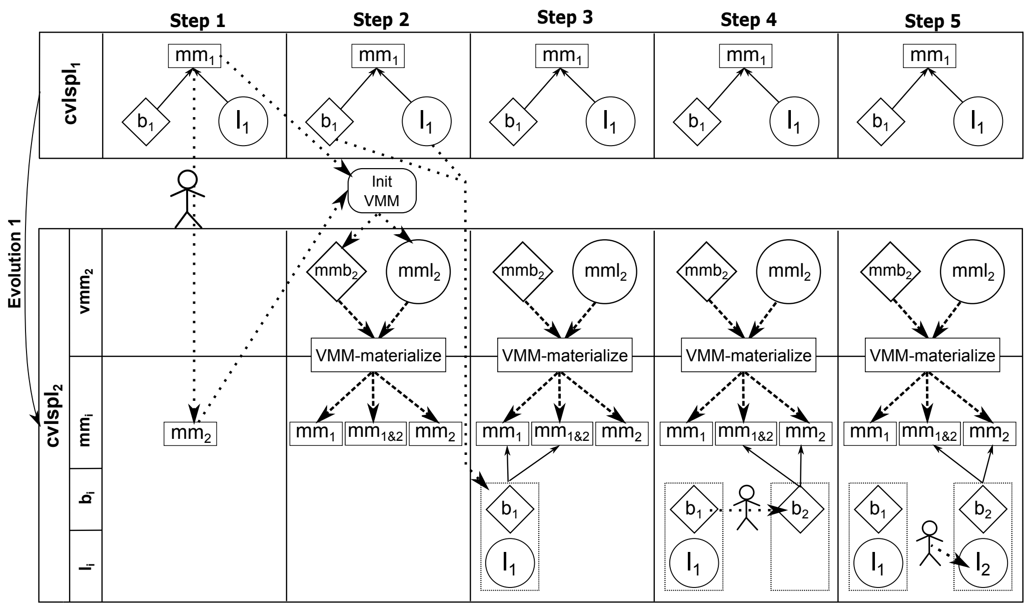

Figure 7: The Steps of the VMM Strategy

in the mml_2. This operation has been explained in deatil in section 6.1. Then, the $VMMmat$ operation can be applied to vmm_i to obtain mm_1, mm_2, and $mm_{1\&2}$ as explained in section 5.

In Step 3, the l_1 and b_1 from $cvlspl_1$ are copied without any modification to be used in $cvlspl_2$. Both conform to the materialized mm_1, and they also conform to the materialized $mm_{1\&2}$. That is, model replacements from $cvlspl_1$ (see first row third column of Figure 2) are copied to $cvlspl_2$.

In Step 4, some common parts of the base model (b_1) may become variable because of the new concepts introduced in generation 2. In that case, the engineer edits the base model b_1 (that has been copied in the previous step) from the $cvlspl_2$ to extract the variable parts as replacements. Notice the modifications performed to b_1 (see first row second column of Figure 2) in order to obtain b_2 (see second row second column of Figure 2)

In Step 5, the engineer creates new replacements that instantiate the new concepts of this generation (see second row third column of Figure 2) and includes them in l_2. These new replacements conform to mm_2, and they also conform to $mm_{1\&2}$.

Following the above steps, we can evolve the SPL from one generation to the next, while eliminating the need for migrating existing fragments. Then, when the engineer wants to create new replacements, the engineer will be able to use the metamodel of just one generation and not the $mm_{1\&2}$. As a result, the engineer can create replacements for the most recent generation (using mm_2) to instantiate the new concepts of that generation. In contrast, the engineer can use the previous generation metamodel (mm_1) to create replacements that do not exercise the expressiveness provided by the new generation, thus avoiding the overcharge of the model (as in the case of the motivating example, see Section 4). When materializing an IH model containing replacements from both generations (l_1 and l_2), the resulting IH model will conform to $mm_{1\&2}$.

In addition, the recursion capabilities of CVL enable us to create VPs inside a replacement and hence apply the VMM strategy to further generations. In other words, when creating the next generation, Step 2 of the process could end up in the creation of a new replacement that includes previously defined VPs (if the replacement is not common for both metamodels).

7.2. The Resulting Models after Applying the VMM Strategy

Figure 8 shows an overview of our industrial partner's $CVLSPL$ models (rows) after applying the VMM strategy to manage Evolution 1 and Evolution 2 (columns).

In **Evolution 1**, a new concept (hotplate) is introduced (see the first and second columns). This concept affects the inductor, which is now aggregated by the hotplate; therefore, we apply the method explained above to perform Evolution 1. InitVMM produces a base model (mmb_2) that contains a VP (P1) with cardinality 2 (i.e., it can be replaced up to two times). InitVMM also produces mml_2, which contains two replacements: R1 (which holds the particularities of Gen1) and R2 (which holds the particularities of Gen2). The $VMMmat$ operation can be applied to those models to produce three different metamodels: 1) the substitution of P1 by R1 produces mm_1; 2) the substitution of P1 by R2 produces mm_2; and 3) the substitution of P1 twice (by R1 and by R2) produces $mm_{1\&2}$.

When creating new fragments, the engineer must stick to only one generation in order to create a valid fragment. In other words, fragments must conform to a specific metamodel generation, either mm_1 or mm_2. As a result, the engineer can create replacements using only concepts from mm_1, thereby eliminating the indirection introduced by the migration strategy (see Section 4).

When materializing an IH model that contains replacements from both generations (l_1 and l_2), the resulting IH model conforms to $mm_{1\&2}$. Overall, vmm_2 enables the materialization of IH models with replacements from both generations (l_1 and l_2), while at the same time allowing the creation of fragments pertaining to one generation (either conforming to mm_1 or to mm_2).

In **Evolution 2**, a new *breaking change* that introduces the concept of cooking zones occurs (see the second and third columns). Similarly to Evolution 1, we apply the method to perform Evolution 2 (from generation 2 to generation 3).

The CVL capabilities of recursion (placements inside replacements) and cardinalities over the VPs applied to the metamodel level have proven to provide enough expressiveness to overcome all of the evolution situations of our industrial partner over 13 years.

7.3. The Derivation of SPL Products after Applying VMM

The VMM strategy has been tooled within the Eclipse environment and integrated into our industrial partner's SPL. The resulting tool is used by our industrial partner (BSH, the leading manufacturer of home appliances in Europe) to generate the firmware of their Induction Hobs (sold under the brands of Bosh and Siemens). An example of the resulting tool in action can be seen here [3]. This section presents an example of using

[3] http://www.carloscetina.com/variablemetamodel.htm

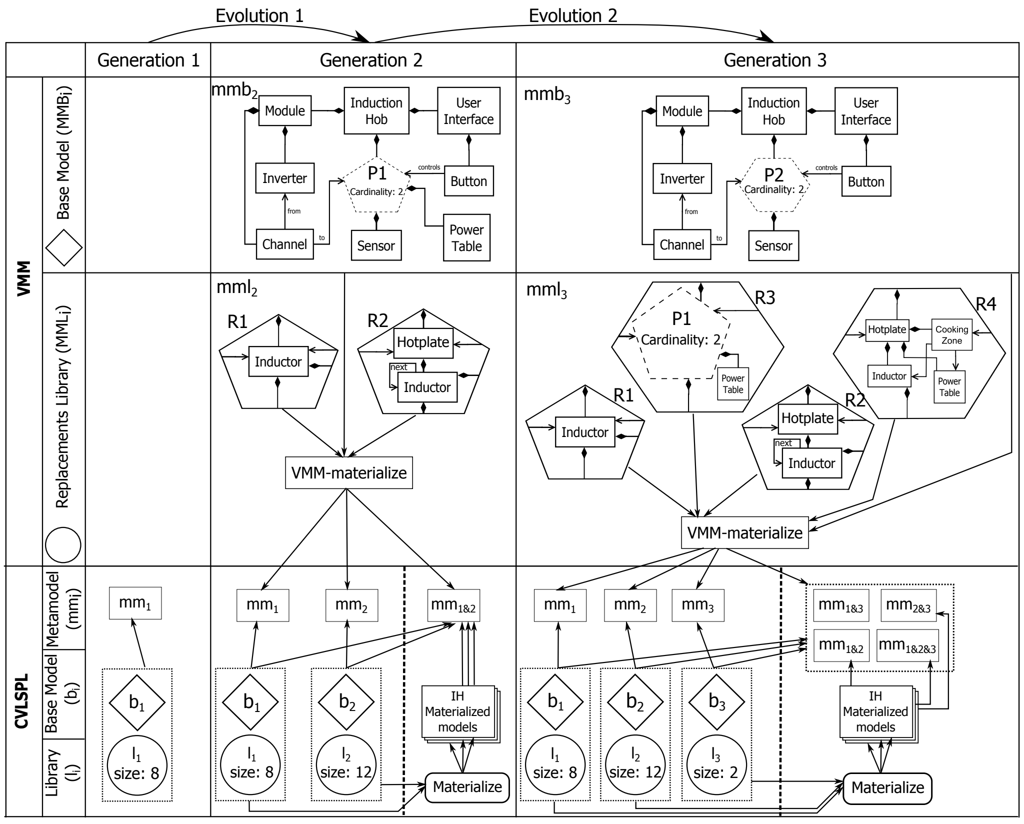

Figure 8: The IHDSL Metamodel level for each generation of the CVLSPL

the SPL evolved with the VMM strategy: an engineer of our industrial partner deriving a new product.

The engineer will act at the model level, choosing which replacements should be substituted in the base model and building the Induction Hob model; in the meantime, at the metamodel level, the metamodel is built up automatically reflecting those model level substitutions. Each time a replacement is chosen by the engineer (at the model level), the replacement (at the metamodel level) corresponding to the replacement chosen by the engineer at the model level will be automatically substituted in the base model metamodel (only if it is the first occurrence of that generation).

Figure 9 shows an example of the derivation when the SPL is in generation 3. At the model level (the first and second columns), the engineer chooses the replacements (the first column) for the VPs of the base model (the second column); at the same time, at the metamodel level (the third and fourth columns), the metamodel replacements (the third column) are automatically substituted for the VPs of the base model (the fourth column). Note that the metamodel level elements presented in Figure 9 (the third and fourth columns) and the metamodel level elements presented in Figure 8 (the third column) are the same.

The first row in Figure 9 shows the first substitution of the product derivation: the

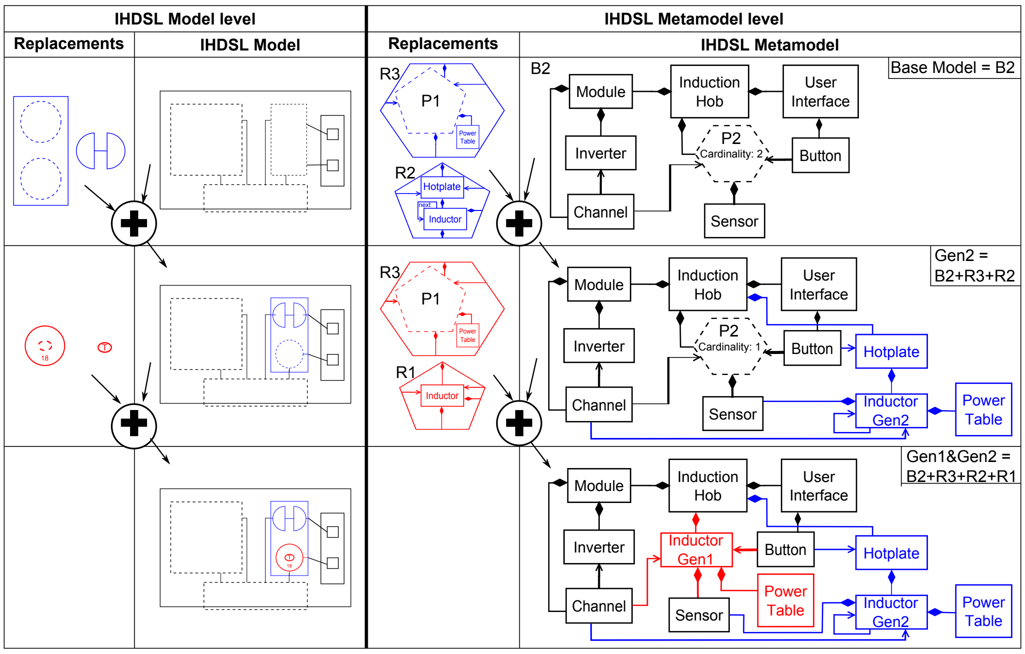

Figure 9: Fragment substitutions for deriving SPL products

engineer can use replacements from the three different generations available. In this case, the engineer is going to use replacements from the second generation (the first column). The base model of the current generation (the second column) is used. The metamodel level has the replacements R2 and R3 (third column) that correspond to the model level replacements, and the metamodel base B2 (fourth column) with all the common elements from all of the generations.

The second row in Figure 9 shows the result of the first fragment substitution. The fragments chosen by the engineer have been substituted at the model level (the second column). At the metamodel level, corresponding fragments have been automatically substituted (the fourth column), resulting in the generation 2 metamodel (Gen2). Now, if more model level replacements from generation 2 are added, the metamodel does not vary (it only varies the first time that a generation is used). We repeat the operation with more replacements: this time they belong to generation 1. At the metamodel level, the corresponding metamodel level replacements R1 and R3 are used.

The third row shows the results of the second fragment substitution. The model now has elements from two SPL generations; therefore, the metamodel has automatically been increased to be the combination of those two generations (Gen1&Gen2) maintaining the conformance between the model (the second column) and the metamodel (the fourth column). The engineer then performs more fragment substitutions until all the VPs of the IH model are substituted; the metamodel is automatically increased as necessary.

The VMM strategy of this work enables our industrial partner's engineers to derive products by means of replacements from any generation, while avoiding the disadvantages of migrating the replacements after each evolution. Section 8 discusses the advantages and disadvantages of each of the strategies, taking into account the experience acquired from our industrial partner.

8. Discussion

We have applied both strategies to the retrospective of 13 years of our industrial partner's SPL models. In this paper, we only show a simplification of the evolution related to the inductor concept even though we have applied it to all of the concepts. This involves about 32 different IH models composed of approximately 72 different model replacements (each of which is composed of multiple model elements). The average number of model elements of a fragment replacement is 43, while the average number of elements of an IH model is about 470. Figure 10 shows a summary of the comparison obtained from the collaboration with our industrial partner of both the migration strategy and the VMM strategy in terms of three dimensions: (a) indirection, (b) automation, and (c) trust leak.

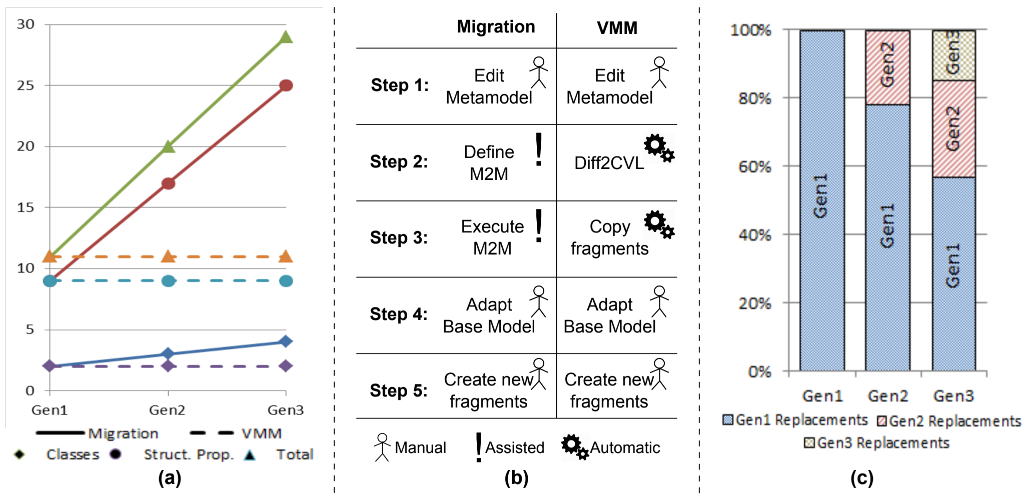

Figure 10: Comparison between Migration and VMM Strategy

8.1. *Indirection*

Indirection refers to an increase in model elements in order to conform to an evolved metamodel while keeping the same functionality, for instance, the inductor that migrates into a hotplate and then into a cooking zone (see Section 4).

Figure 10 (a) shows the comparison of both strategies in terms of the indirection that is present in the replacements. The graph shows the number of model elements (classes and structural properties) used in each generation to represent an inductor. In the migration strategy (solid lines), the inductor grows from a total of 11 elements in Gen 1 to a total of 29 elements in Gen 3. This growth trend is common for all of the concepts studied in this work. Although it is out of the scope of this paper, there are transformations based on the metamodel to transform IHDSL models into code, and this indirection requires modifications and produces an increase in the complexity of the transformations and the code generated. In contrast, the VMM strategy (dashed lines) avoids the migration of replacements, and the number of elements needed to represent the inductor concept (11) remains the same over all of the generations.

8.2. Automation

Depending on the degree of involvement of the user, the execution of the steps of both strategies can be either manual, assisted, or automatic. A step is automatic when it is done without user intervention; it is assisted when the user must help in the process; and it is manual when the whole process is performed by the user.

Figure 10 (b) shows the comparison of the two strategies in terms of automation for each of the steps of the strategies. Step 1 (Edit Metamodel) is the same for both strategies and must be performed manually. Step 2 is different; the migration strategy requires the definition of a M2M transformation. With the options that are available (manual [6], operator-based [7, 8], or metamodel matching [9, 10]), the process is, at best, assisted [17, 9]. In contrast, in the VMM Strategy Step 2 (InitVMM and addGen) is fully automatizable, (CVL applied to the model and the metamodel level enabled us to resolve all kinds of changes presented by [9] in an automatic way). Step 3 in the migration strategy is the execution of the M2M transformation. Breaking changes (e.g., the addition of obligatory properties) are not automatically resolvable ([17, 9]), so the step needs to be assisted. In contrast, in the VMM strategy replacements are used "as is" (i.e., no migration is required and only an automatic copy is performed). Finally Steps 4 (Adapt base model) and 5 (Create new replacements) are performed manually in both strategies.

8.3. Trust Leak

Models are used to produce code; once they have been used repeatedly on many IHs, they gain the trust of our industrial partner's engineers. However, when the replacements are modified, there is a loss of this trust on the part of the engineers, which has been reported as *trust leak*.

Figure 10 (c) shows the evolution of the replacements being used in each generation, regarding the generation when they were created. That is, the graph shows the weight of the replacements originated in each generation in relation to the total number of products created with the SPL (i.e., the average percentage of replacements originating from each generation present in the induction hobs taking into account all of the IHs derived from the SPL for that generation). This is highly relevant for the *trust leak* phenomena, as it is related to the number of migrations that the replacements overcome.

In generation 1, all the fragments used to build the products were originated in that generation. However, only 22% of the replacements used by products in generation 2 are originated in that generation. The rest 78% of replacements were created in generation 1 and if not using the VMM strategy need to be migrated to conform to generation 2 metamodel (resulting in a decrease in the trust, as the model elements are modified). In generation 3 the effect is increased, as only a 17% of the replacements are created in that generation. The rest of the fragments have been created in previous generation but are still being used by products of generation 3. Therefore, if we apply a migration strategy 83% of the fragments needed by products of that generation will need to be migrated from previous generations (58% of them twice, from Gen1 to Gen2 and then to Gen3).

It turns out that the replacements from generation 1 are the ones that are most frequently used to build IHs (in all generations), and they are also the ones that require more migrations when following the migration strategy. Therefore, those are the replacements that have the highest level of trust leak as the trust is reduced each time that the replacement needs to be modified.

9. Lessons Learned

This section presents three lessons learned from the adoption of the presented VMM approach as part of the SPL of our industrial partner. After a period of usage of the approach by our industrial partner, we reviewed the VMM created to determine whether it was working properly. As part of this review process, we learned some lessons that enabled us to improve the approach. The first lesson id related to the creation of false revisions, the second lesson is related to the folding of revisions, and the third lesson is related to isolated revisions.

9.1. False Revisions

We designed the presented VMM to automatically include new metamodel revisions. In other words, each time a new metamodel was created, the addGen operation was triggered and the new revision was formalized in terms of CVL (when needed due to a breaking and unresolvable change). However, when reviewing the VMM generated by our industrial partner after the period of usage, we realized that some false revisions were being created in the VMM.

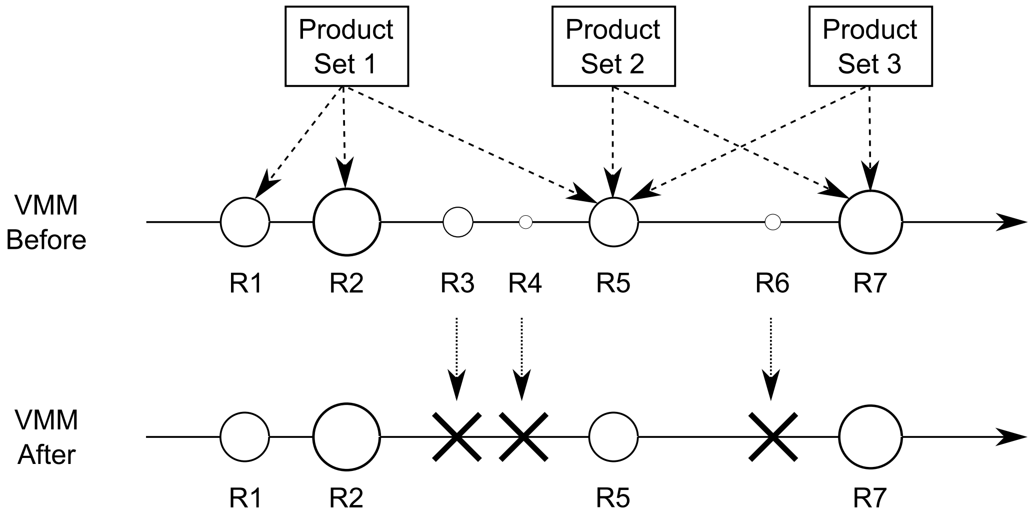

Figure 11: False Revisions

Figure 11 shows an example of false revisions. The horizontal arrows represent the *VMM before* and after addressing the false revision issue. The *VMM before* shows 7 different revisions (circles). The number of model fragments generated for each revision is represented by the size of the circle. In addition, there are some products that were

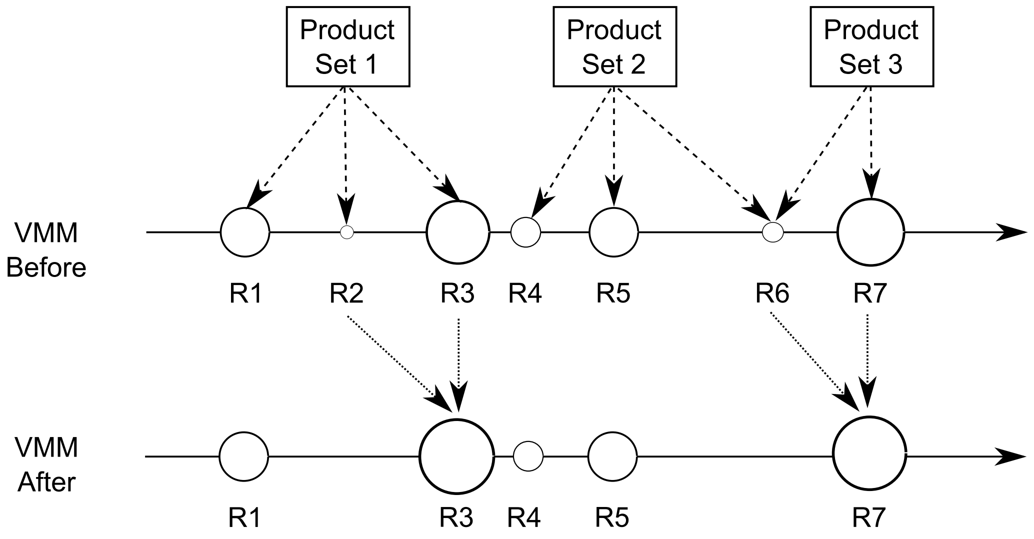

Figure 12: Revision Folding

built based on the model fragments from the revisions. For instance, Product Set 1 is composed of model fragments from three different revisions (R1, R2, and R5). However, there are some revisions that were not used to build any of the products (R3, R4, and R6). We discussed this situation with our industrial partner. It turns out that those revisions where tests that were discarded and not used to build real products.

Therefore, we decided to remove those revisions from the VMM (as in the *VMM after*) and thereby reduce the complexity of the VMM. It turns out that what defines a new generation is not just the creation of a new metamodel revision or the creation of new model fragments for that revision. Those tasks (the creation of a revision and the addition of model fragments) are common for testing purposes. What defines the creation of a new generation is the usage of model fragments (that belong to the new revisions) to build new products. Therefore, we decided to postpone the addition of new metamodel revisions until they are used for the creation of new products (as in Section 7.3).

However, the false revisions (R1, R2, and R5) are not deleted as they might be used to create products in the future. Therefore, we store them into an auxiliary VMM, a copy of the 'main' VMM that is used only for storing purposes (not to build new products). Then, the user can create new replacements using those metamodel revisions in the auxiliary VMM. When the user uses a replacement created with one of those revisions to build a product, the revision (that is stored into an auxiliary VMM) is added to the 'main' VMM and is not considered anymore a false revision.

9.2. Revision Folding

Some situations also suggested the need for removing a particular revision from the VMM even though they are not false revisions (i.e., being used by some products). In other words, a new metamodel revision that includes a concept is created, model fragments for that revision are developed, and products using those model fragments are created. Then, an issue with the revision is found and a new revision (fix revision) that properly represents the concept and addresses the issue discovered needs to be created. After the fix revision is created, the old revision is not used anymore, but it is not possible to remove it directly (as there are products using it). To manage situations of this kind, we introduced revision folding.

Figure 12 shows an example of revision folding. In the VMM *before*, an issue is discovered in R2 after some products from Product Set 1 have already been created. Then, our industrial partner created revision R3 to address the issues discovered in R2 and started using it. R2 is no longer needed, but some of the model fragments (which were not affected by the issue discovered) are still in use. To address this kind of situation, we propose migrating the model fragments from R2 to R3 and folding both revisions into a single one. The VMM after shows how the R2 and R3 revisions have been folded (into R3). The same situation occurs with R6 and R7.

As a result, the products previously using model fragments from R2 now are using the migrated fragments from R3. This migration usually only affects a small set of fragments, and the lifespan of those fragments is short. Therefore, the disadvantages of migration are outweighed by having a clearer and smaller set of revisions under the VMM. In other words, when the engineer considers that two metamodel revisions are mutually exclusive and the later revision is a direct fix of the previous one, the engineer can fold both revisions, migrating the fragments that belong to the unused metamodel revision.

When the engineer decides to fold two revisions, the traditional migration strategy is followed. That is, the steps presented in section 4 are followed to migrate the fragments from the fault revision to the fix revision. The engineer is guided through the process that can be fully automated if there are not breaking changes among the two revisions.

9.3. Isolated Revisions

When reviewing the VMM generated by our industrial partner, we also discovered some isolated revisions (i.e., some revisions are only used to build products that do not include other revisions). Therefore, the products conform to that particular metamodel revision and it is not necessary to combine it with other metamodel revisions. As a result, that revision can be extracted from the VMM, decreasing the number of revisions managed and the complexity.

Figure 13 shows an example of an isolated revision. The VMM *before* shows four products sets built with model fragments from six different revisions. However, Product Set 3 is built only with model fragments from R4. In addition, R4 model fragments are not used to build any other product. As a result, R4 can be extracted from the VMM since it is not used in combination with any other revision. Product Set 2 is also built only with model fragments from a single revision (R3). However, R3 model fragments are also used to build Product Set 1, where R3 is combined with R1 and R2. Therefore it is not possible to extract R3 from the VMM. Only revisions that are not combined with other revisions can be extracted from the VMM.

When isolated revisions are extracted from the 'main' VMM, they are stored into an auxiliary VMM. It is important to notice that, although at that point in time the revision is isolated, it could stop being isolated if the engineer creates a product that combines replacements from the isolated revision and other revisions. Therefore, in that

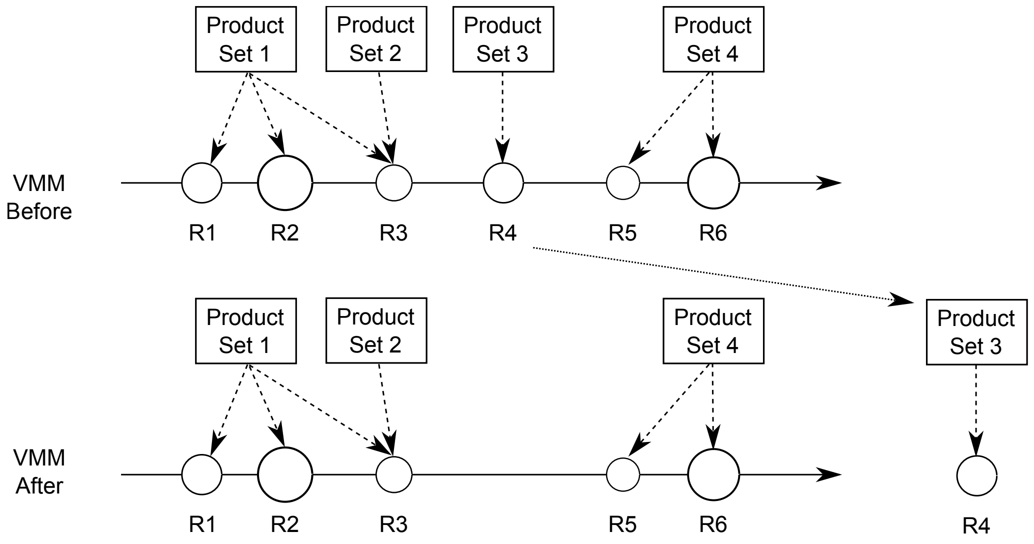

Figure 13: Isolated Revisions

event, the isolated revision that is stored into the auxiliary VMM is moved to the 'main' VMM.

The VMM strategy eliminates the need for migration and properly manages different metamodel revisions. However, the inclusion of the VMM strategy also entails the need to properly manage the generations. As indicated by these lessons, in order to reduce the complexity of the VMM, the creation of false revisions must be avoided, the means for folding revisions must be provided, and isolated revisions must be properly extracted.

10. Related Work

To the best of our knowledge, there are no works that address the evolution of SPLs using variability modeling ideas at the metamodel level. However, there are research efforts on SPL evolution that can complement model-based SPL evolution.

There are some approaches that rely on model comparisons to formalize the variability that exists among a set of models [18, 19, 20, 21]. However, those operations are applied at the model level (not at the metamodel level) and the aim of the operations is to be capable of recreating the products used as input. In other words, they produce the fragments necessary to recreate the models, but they do not capture each model's particularities in separate fragments. In contrast, the initVMM operation from our approach is designed to capture the particularities of each metamodel revision provided as input in separate fragments, thus enabling the possibility of materializing metamodels that combine several generations.

In [22], Batory et al. present the AHEAD model, which is based on the step-wise refinement paradigm and enables the synthesization of multiple complex programs from a simple program. In AHEAD, the software is expressed as nested sets of equations that describe feature refinements. The composition function (which is specific for each kind of asset) is used to stack the refinements applied to the base program to produce the different variants. However, we do not focus on how to specify variants of the base product; the main focus in our approach is to avoid the migration of the models from one generation to the next by applying variability at the metamodel level.

Dhungana et al. [23] present an approach that is based on model fragments that are applied at the model level. The tool support for the automated detection of changes facilitates metamodel evolution and the propagation of changes in the domain to already existing variability models. However, in contrast to our approach, they do not use fragments at the metamodel level, requiring their fragments to be updated when changes occur at the metamodel level.

Deng et al. [24] argue that adding new requirements to a model-based Product Line Architecture (PLA) often causes invasive modifications to the PLA‘s component frameworks and DSLs. To address these modifications, they show how structural-based model transformations help maintain the stability of domain evolution by automatically transforming domain models. Although the details are different, their approach is similar to the migration strategy with the support of model transformations. However, our work shows that, in the case of a CVLSPL, the *VMM strategy* offers better results.

Creff et al. [25] propose an incremental evolution by extension of the product line. They aim to benefit from the investments made during the product derivation and *reinvest* them into the SPL models. Specifically, they introduce an assisted feedback algorithm to extend the SPL to emerging product derivation requirements. We believe that their feedback algorithm could be tailored to help in the detection of the need for new metamodel changes (new SPL generations) when product derivations occur, triggering our *VMM strategy* to address the evolution at the metamodel level.

All of these works are based on address the evolution of a software. However, these approaches do not take into account the problems that can arise from the migration of the deprecated software of the system (in our case model fragments). We focus our work on the evolution that incorporates new versions of products without damaging the rest of the product that are already developed and in use. Anyway, all of them are using different strategies and, although we base our evolution strategy in the ideas of CVL, we can adapt their ideas to use them with our evolution strategy.

There are much more research efforts in categorization and analysis of changes that can trigger the need of evolving a system. These works combines empirical studies and analysis in order to obtain a better understanding of the changes that occurs in the software life cycle.

Lotufo et al. [26] provide empirical evidence of how a large real-world variability model evolves. They present their study using 21 versions of the Linux kernel over five years. Their entire development process is feature driven. They analyze how a number of characteristics, such as number of features, height of the tree and depth of the leaves, using the feature models of those versions. Based on this investigation, they identify six categories of reasons for changes. Although we don't use feature models, we can consider their categories to categorize the changes that occur in the evolution of our CVLSPL.

Passos et al. [27] developed a vision of software evolution that is based on a feature-oriented perspective. They provided a feature-oriented project management and system development platform that supports traceability and analyses. In our work, the SPL is specified by means of base models, fragment substitution, and metamodel expressiveness. However, since we can represent the variability model of our industrial partner's SPL by means of a feature model, our strategy can benefit from the analysis and traceability of the work of Passos et al. [27].

Similar to us, they are focus on software evolution in the real-world. Although they studies are based on different techniques, some of the ideas and categorizations could be applied to the evolution of our CVLSPL to obtain a more accurate evolution strategy.

11. Conclusions

The CVL capabilities of recursion (placements inside replacements) and cardinalities over the VPs applied to the metamodel level have proven to provide enough expressiveness to overcome all the evolution situations of our industrial partner over 13 years. In addition, the VMM strategy of this work enables our industrial partner's engineers to derive products by means of replacements from any generation, while avoiding the disadvantages of migrating the replacements after each evolution.

This work indicates that the VMM achieves better results than the migration strategy in domains like the domain of our industrial partner in terms of indirection, automation, and trust leak. Furthermore, using already existing variability management approaches (like CVL) enables us to bring efforts from the variability research community to address the evolution challenge. Nevertheless, there are still open issues (e.g., evolutions that turn variabilities into commonalities) that will be addressed in our work in the future.

The VMM strategy avoids the need for migration of model fragments when new metamodel revisions arise, but at the expense of introducing the need of managing the VMM. Such management is needed to avoid creation of false revisions, provide means for folding revisions and properly extract isolated revisions to reduce the complexity of the VMM. However, the management of the VMM is performed at a higher abstraction level than the migration tasks that have been replaced.

12. References

[1] K. Pohl, G. Böckle, F. Van Der Linden, Software product line engineering: foundations, principles, and techniques, Springer, 2005.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. rep., DTIC Document (1990).

[3] D. Benavides, S. Segura, A. Ruiz-Corts, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (6) (2010) 615 – 636. doi:http://dx.doi.org/10.1016/j.is.2010.01.001.
URL http://www.sciencedirect.com/science/article/pii/S0306437910000025

[4] F. Fleurey, Ø. Haugen, B. Møller-Pedersen, G. K. Olsen, A. Svendsen, X. Zhang, A generic language and tool for variability modeling, Technical Report SINTEF A13505.

[5] J.-M. Favre, Meta-model and model co-evolution within the 3d software space, in: In Procedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM, Amsterdam, Netherlands, 2003, pp. 98–109.

[6] L. M. Rose, R. F. Paige, D. S. Kolovos, F. A. Polack, An analysis of approaches to model migration, in: Proc. Joint MoDSE-MCCM Workshop, 2009, pp. 6–15.

[7] M. Herrmannsdoerfer, S. Benz, E. Juergens, Cope - automating coupled evolution of metamodels and models, in: S. Drossopoulou (Ed.), ECOOP 2009 Object-Oriented Programming, Vol. 5653 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 52–76. doi:10.1007/978-3-642-03013-0_4.
URL http://dx.doi.org/10.1007/978-3-642-03013-0_4

[8] G. Wachsmuth, Metamodel adaptation and model co-adaptation, in: E. Ernst (Ed.), ECOOP 2007 Object-Oriented Programming, Vol. 4609 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 600–624. doi:10.1007/978-3-540-73589-2_28.
URL http://dx.doi.org/10.1007/978-3-540-73589-2_28

[9] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE, 2008, pp. 222–231. doi:10.1109/EDOC.2008.44.

[10] K. Garcés, F. Jouault, P. Cointe, J. Bézivin, Managing model adaptation by precise detection of metamodel changes, in: R. F. Paige, A. Hartman, A. Rensink (Eds.), Model Driven Architecture - Foundations and Applications, Vol. 5562 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 34–49. doi:10.1007/978-3-642-02674-4_4.
URL `http://dx.doi.org/10.1007/978-3-642-02674-4_4`

[11] M. Svahnberg, J. Bosch, Evolution in software product lines: Two cases, Journal of Software Maintenance 11 (6) (1999) 391–422. doi:10.1002/(SICI)1096-908X(199911/12)11:6¡391::AID-SMR199¿3.0.CO;2-8.

[12] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Building software product lines from conceptualized model patterns, in: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, 2015, pp. 46–55. doi:10.1145/2791060.2791085.
URL `http://doi.acm.org/10.1145/2791060.2791085`

[13] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Addressing metamodel revisions in model-based software product lines, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015, 2015, pp. 161–170. doi:10.1145/2814204.2814214.
URL `http://dx.doi.org/10.1145/2814204.2814214`

[14] R. C. Gonçalves, D. Batory, J. L. Sobral, T. L. Riché, From software extensions to product lines of dataflow programs, Software & Systems Modeling (2015) 1–19doi:10.1007/s10270-015-0495-8.
URL `http://dx.doi.org/10.1007/s10270-015-0495-8`

[15] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, B. Garvin, Exploring feature interactions in the wild: The new feature-interaction challenge, in: Proceedings of the 5th International Workshop on Feature-Oriented Software Development, FOSD '13, ACM, New York, NY, USA, 2013, pp. 1–8. doi:10.1145/2528265.2528267.
URL `http://doi.acm.org/10.1145/2528265.2528267`

[16] E. Foundation, Eclipse modeling framework compare (emfcompare) website, http://wiki.eclipse.org/index.php/EMF$Compare$(2008).

[17] B. Gruschko, D. Kolovos, R. Paige, Towards synchronizing models with evolving metamodels, in: Proceedings of the International Workshop on Model-Driven Software Evolution, 2007.

[18] X. Zhang, Ø. Haugen, B. Moller-Pedersen, Model comparison to synthesize a model-driven software product line, in: Proceedings of the 2011 15th International Software Product Line Conference, SPLC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 90–99. doi:10.1109/SPLC.2011.24.
URL `http://dx.doi.org/10.1109/SPLC.2011.24`

[19] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, 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. doi:10.1145/2791060.2791086.
URL `http://dx.doi.org/10.1145/2791060.2791086`

[20] J. Font, M. Ballarín, Ø. Haugen, C. Cetina, Automating the variability formalization of a model family by means of common variability language, in: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, 2015, pp. 411–418. doi:10.1145/2791060.2793678.
URL `http://dx.doi.org/10.1145/2791060.2793678`

[21] J. Rubin, M. Chechik, Combining related products into product lines, in: J. de Lara, A. Zisman (Eds.), Fundamental Approaches to Software Engineering, Vol. 7212 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 285–300. doi:10.1007/978-3-642-28872-2_20.
URL `http://dx.doi.org/10.1007/978-3-642-28872-2_20`

[22] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 187–197.
URL `http://dl.acm.org/citation.cfm?id=776816.776839`

[23] D. Dhungana, P. Grünbacher, R. Rabiser, T. Neumayer, Structuring the modeling space and

supporting evolution in software product line engineering, Journal of Systems and Software 83 (7) (2010) 1108–1122. doi:http://dx.doi.org/10.1016/j.jss.2010.02.018.
URL http://www.sciencedirect.com/science/article/pii/S0164121210000506

[24] G. Deng, D. C. Schmidt, A. Gokhale, J. Gray, Y. Lin, G. Lenz, Evolution in model-driven software product-line architectures, Designing Software-Intensive Systems: Methods and Principles.

[25] S. Creff, J. Champeau, J.-M. Jézéquel, A. Monégier, Model-based product line evolution: an incremental growing by extension, in: 16th International Software Product Line Conference, SPLC '12, ACM, NY, USA, 2012. doi:10.1145/2364412.2364430.
URL http://doi.acm.org/10.1145/2364412.2364430

[26] R. Lotufo, S. She, T. Berger, K. Czarnecki, A. Wasowski, Evolution of the linux kernel variability model, in: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 136–150.
URL http://dl.acm.org/citation.cfm?id=1885639.1885653

[27] L. Passos, K. Czarnecki, S. Apel, A. Wasowski, C. Kästner, J. Guo, C. Hunsen, Feature-oriented software evolution, in: 7th International Workshop on Variability Modelling of Software-intensive Systems, ACM, Italy, 2013.