Generating Counterexamples of Model-based Software Product Lines: An Exploratory Study

Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Jérôme Le Noir, Benoit Baudry

Generating Counterexamples of Model-based Software Product Lines: An Exploratory Study*

João Bosco Ferreira Filho, Olivier Barais, Mathieu Acher
INRIA and IRISA
Université Rennes 1, France

Benoit Baudry
INRIA and SIMULA RESEARCH LAB
Rennes, France and Lysaker, Norway

Jérôme Le Noir
Thales Research & Technology
Palaiseau, France

ABSTRACT

Model-based Software Product Line (MSPL) engineering aims at deriving customized models corresponding to individual products of a family. MSPL approaches usually promote the joint use of a variability model, a base model expressed in a specific formalism, and a realization layer that maps variation points to model elements. The design space of an MSPL is extremely complex to manage for the engineer, since the number of variants may be exponential and the derived product models have to be conformant to numerous well-formedness and business rules. In this paper, the objective is to provide a way to generate MSPLs, called counterexamples, that can produce invalid product models despite a valid configuration in the variability model. We provide a systematic and automated process, based on the Common Variability Language (CVL), to randomly search the space of MSPLs for a specific formalism. We validate the effectiveness of this process for three formalisms at different scales (up to 247 metaclasses and 684 rules). We also explore and discuss how counterexamples could guide practitioners when customizing derivation engines, when implementing checking rules that prevent early incorrect CVL models, or simply when specifying an MSPL.

1. INTRODUCTION

In many domains, systems have to be efficiently extended, changed, customized or configured for use in a particular context (e.g., to respond to the specific expectations of a customer) [34, 10]. The challenge for practitioners is to develop and maintain multiple similar products (variants), exploiting what they have in common and managing what varies among them [5]. *Software Product Line (SPL)* engineering has emerged to address the problem [14, 31] involving both the research community and the industry.

Models, as high-level specifications of a system, are traditionally employed to automate the generation of products as well as their verifications [33]. A variety of models may be used for different development activities and artefacts of an SPL – ranging from requirements, architectural models, source codes, certifications and tests to user interfaces. Likewise, different stakeholders can express their expertise through specific modeling languages and environments, an important requirement in large companies like Thales [39].

Numerous *model-based SPL (MSPL)* techniques have been proposed (e.g., see [31, 29, 26, 15, 13, 18, 41, 38]). They usually consist in *i)* a variability model (e.g., a feature model or a decision model), *ii)* a model (e.g., a state machine, a class diagram) expressed in a specific modeling language (e.g., Unified Modeling Language (UML) [24]), and *iii)* a realization layer that maps and transforms variation points into model elements. Based on a selection of desired features in the variability model, a derivation engine can automatically synthesise customized models – each model corresponding to an individual product of the SPL. The *Common Variability Language (CVL)* [22] has recently emerged as an effort to standardize and promote MSPLs.

The *design space* (also called domain engineering) of an MSPL is extremely complex to manage for a developer. First, the number of possible products of an MSPL is exponential to the number of features or decisions expressed in the variability model. Second, the derived product models[1] have to be conformant to numerous well-formedness and business rules expressed in the modeling language (e.g., UML exhibits 684 validation rules in its EMF implementation). Consequently, a developer has to understand the intrinsic properties of the modeling language when designing an MSPL. Third, the realization model that connects a variability model and a set of design models, can be very expressive, especially in the case of CVL. Managing variability models or design models is a non trivial activity. Connecting both parts and therefore managing all the models is a daunting and error-prone task.

Specifically, *managing the design space of an MSPL* raises two key issues. First, the *realization model* specifies how to remove, add, substitute, modify (or a combination of these operations) model elements. Elaborating such a model is error-prone because, for example, it is easy for an SPL designer to specify instructions that both delete and add the same model element for a given combination of features [18]. Second, the *derivation engine* executes the realization model and produces a product model that has to be conformant to the syntax and the semantics of the modeling language. Assuring the correctness of the derivation engine for a given modeling language is still a theoretical and practical problem. Considering the aforementioned, deriving a valid product of an MSPL is not anymore just dependent on having a valid feature configuration. Additionally, this correctness is now dependent on *i)* the realization model and *ii)* the derivation engine, both participating to the synthesis of product

*This work was developed in the VaryMDE project, a bilateral collaboration between the Triskell team at INRIA and the Thales Research & Technology.

[1]CVL uses the term materialization to refer to the derivation of a model. Also, a selected/unselected feature corresponds to a positively/negatively decided VSpec. We adopt the well-known vocabulary of SPLE for the sake of understandability.

models.

We formulate the hypothesis that a one-size-fits-all support for deriving models is unlikely, since models are conformant to their own well-formedness (syntactic) rules and domain-specific (semantic) rules. Each time a new modeling language is used for developing an MSPL, the realization layer should be revised accordingly. We already observed this kind of situation in the context of prototyping the use of CVL with Thales on dedicated domain-specific modeling languages for systems engineering. For instance, in [21], we expose different strategies to customize the derivation engine since the one provided by default in CVL does not suit the needs. Ideally, an MSPL should derive safe product models for each authorized configuration. Our long term objective is to assist stakeholders on improving the development of derivation engines or the specification of realization models for arbitrary MSPLs.

In this paper, the objective is to provide a way to generate *counterexamples of MSPLs*, that is, examples of MSPLs that authorize the derivation of syntactically or semantically invalid product models despite a valid configuration in the variability model. These counterexamples aim at revealing errors or risks – either in the derivation engine or in the realization model – to stakeholders of MSPLs. On the one hand, counterexamples serve as testing "oracles" for increasing the robustness of checking mechanisms for the MSPL. Developers can use counterexamples to foresee boundary values and types of MSPLs that are likely to allow incorrect derivations. On the other hand, stakeholders may repeat the same kind of errors when specifying the mappings between a variability model and a base model. Counterexamples act as "antipatterns" that should avoid bad practices or decrease the amount of errors for a given modeling language.

We provide a systematic and automated process, based on CVL, to randomly search the space of MSPLs for a specific formalism. We validate the effectiveness of this process for three formalisms (UML, Ecore and a simple finite state machine) with different scales (up to 247 metaclasses and 684 rules) and different ways of expressing validation rules. We also explore the hypothesis exposed above, i.e., that a generic derivation engine or a basic support for managing the realization layer is likely to authorize incorrect MSPLs. We discuss how counterexamples could guide practitioners when customizing derivation engines, when implementing checking rules that prevent early incorrect CVL models, or simply when specifying an MSPL. Overall, the generative techniques and exploratory study call for solutions aware of the semantics of the targeted modeling languages when developing MSPLs.

2. BACKGROUND AND MOTIVATION

2.1 Model-based Software Product Lines

An SPL is a set of similar software products that share common features and assets in a particular domain. The process of constructing products from the SPL and domain assets is called *product derivation*. Depending on the form of implementation, there can be different automation levels of product derivation, from manual development effort to more sophisticated technology, including automated variant configuration and generation.

An MSPL has the same characteristics and objectives of an SPL, except that it extensively relies on *models*. In an MSPL, domain artefacts (requirements, tests, graphical interfaces, code) are represented as models conformant to a given modeling language, also called metamodel. (For instance, state machines can be used for specifying and testing the behavior of a system.) The goal of an MSPL is to derive customized models, corresponding to a final product, through a set of *automated transformations* [38, 16].

Numerous approaches, being annotative, compositional or transformational, have been proposed to develop MSPLs (see Section 5 for more details). We will use the *Common Variability Language (CVL)* throughout the paper. We chose CVL because many of the MSPL approaches are actually amenable to this language (CVL is an effort involving both academic and industry partners to promote standardization for MSPLs).

2.2 Common Variability Language

In this section, we briefly present the main concepts of CVL and introduce some formal definitions that are useful for the remainder of this paper. CVL is a domain-independent language for specifying and resolving variability over any instance of any MOF[2]-compliant metamodel. The overall principle of CVL is close to many MSPL approaches: (i) A variability model formally represents features/decisions and their constraints, and provides a high-level description of the SPL (domain space); (ii) a mapping with a set of models is established and describes how to change or combine the models to realize specific features (solution space); (iii) realizations of the chosen features are then applied to the models to derive the final product model.

CVL offers different constructs to develop an MSPL, and they can be distinguished in three parts:

- **Variability Abstraction Model (VAM)** expresses the variability in terms of a tree-based structure. Inspired by feature and decision modeling approaches [17], the main concepts of the VAM are the variability specifications, called *VSpecs*. The *VSpecs* are nodes of the VAM and can be divided into three kinds (Choices, Variables, or Classifiers). In the remainder of the paper, we only use the *Choices VSpecs*, making the VAM structure as close as possible to a Boolean feature model – the variant of feature models among the simplest and most popular in use [8]. These *Choices* can be decided to yes or no (through *ChoiceResolution*) in the configuration process.

- **Base Models (BMs)** a set of models, each conforming to a domain-specific modeling language (e.g., UML). The conformance of a model to a modeling language depends both on well-formedness rules (syntactic rules) and business, domain-specific rules (semantic rules). The Object Constraint Language (OCL) is typically used for specifying the static semantics. In CVL, a base model plays the role of an asset in the classical sense of SPL engineering. These models are then customized to derive a complete product.

- **Variability Realization Model (VRM)** contains a set of Variation Points (VP). They specify how *VSpecs* (i.e., *Choices*) are realized in the base model(s). An

[2]The Meta-Object Facility (MOF) is an OMG standard for modeling technologies. For instance, the Eclipse Modeling Framework is more or less aligned to OMG's MOF.

SPL designer defines in the VRM what elements of the base models are removed, added, substituted, modified (or a combination of these operations, see below) given a selection or a deselection of a *Choice* in the VAM.

Using CVL, the decision of a *Choice* will typically specify whether a condition of a model element, or a set of model elements, will change after the derivation process or not. In this way, these choices must be linked to the model elements, and the links must explicitly express what changes are going to be performed. The aforementioned links compose the VRM, determining what will be executed by the **derivation engine**. Therefore, these links contain their own meaning. We consider that these links can express three different types of semantics:

- **Existence**. It is the kind of VP in charge of expressing whether an object (*ObjectExistence* variation point) or a link (*LinkExistence* variation point) exists or not in the derived model.
- **Substitution**. This kind of VP expresses a substitution of a model object by another (*ObjectSubstitution* variation point) or of a fragment of the model by another (*FragmentSubstitution*)
- **Value Assignment**. This type of VP expresses that a given value is assigned to a given slot in a base model element (*SlotAssignment VP*) or a given link is assigned to an object (*LinkAssignment VP*).

Using the models provided by CVL, one can completely express the variability over any MOF-compliant BM. In addition, it is possible to derive a family of models that will compose an MSPL. Therefore, it is possible to properly define an MSPL in terms of CVL (see Definition 1).

Definition 1 (Model-based SPL). *An MSPL* $= \langle CVL, \delta \rangle$ *is defined as follows:*

- *A* $CVL = \langle VAM, VRM, BMS \rangle$ *model is a 3-tuple such that:*
  - VAM *is a tree-based structure of* VSpecs. *We denote* $\mathcal{C}_{VAM}$ *the set of possible valid configurations for* VAM *;*
  - VRM *is a model containing the set of mapping relationships between the* VAM *and the* BM [3]*;*
  - $BMS = \{BM_1, BM_2, \ldots, BM_n\}$ *is a set of models, each conforming to a modeling language ;*
- $\delta : CVL \times c \rightarrow DM$ *is a function that produces a derived model* DM *from a* CVL *model and a configuration* [4] $c \in \mathcal{C}_{VAM}$. *This function represents the derivation engine.*

2.3 Issues in Realizing Variability

We now introduce our running example to illustrate CVL and the issues raised when developing an MSPL.

Running Example. Let us consider the Finite-State Machine (FSM) modeling language. As shown in Figure 1, the FSM metamodel has three classes: *State*, *Transition*, and *FSM*. The metamodel defines some rules and constraints:

[3]realization layer in the current CVL specification
[4]resolution model in CVL specification

a finite state machine has necessary one initial state and a final state ; a transition is necessary associated to a state, etc. Some other rules may be expressed with OCL constraints (they are not depicted in Figure 1 for conciseness), for example, to specify that there are no *State*s with the same name.

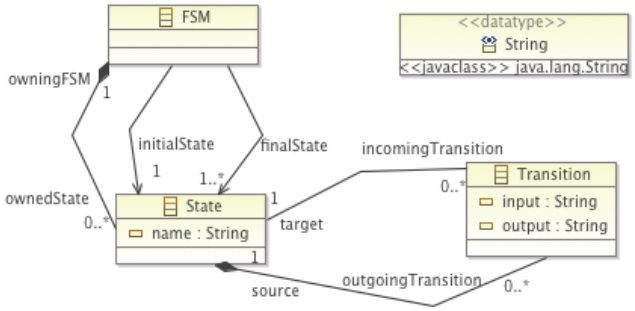

Figure 1: FSM metamodel.

Using CVL and the metamodel of Figure 1, we can define a family of finite state machines. As shown in Figure 2, the VAM is composed by a set of *VSpecs*, while the VRM is a list of variation points, binding the VAM to the BM. The BM is a set of states and transitions conforming to the metamodel presented in Figure 1. The schematic representation of Figure 2 depicts a VAM (left-hand side) with 6 boolean choices (e.g., VS_5 and VS_6 are mutually exclusive) as well as a VRM that maps VS_3, VS_2, VS_5 and VS_6 to transitions or states of a base model denoted BM.

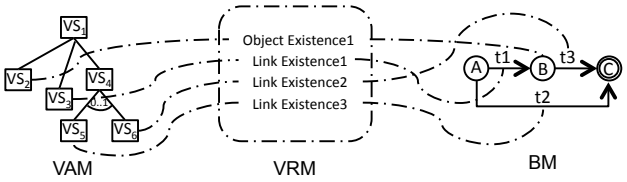

Figure 2: CVL model over an FSM base model.

Considering the MSPL of Figure 2, it is actually possible to derive incorrect FSM models even starting from a valid BM and valid configurations of VAM. This is illustrated in Figure 3. Configuration 1 generates a correct FSM model,

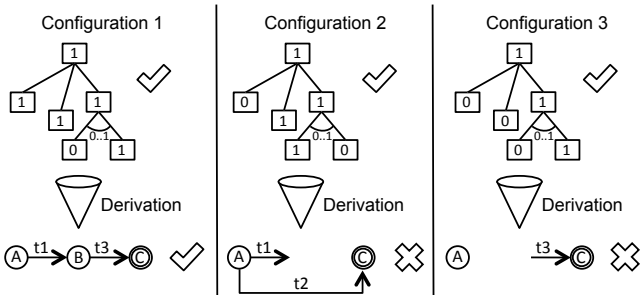

Figure 3: Configuration and derivation of FSMs.

i.e., conforming to its metamodel. *Configuration* 2 and *Configuration* 3, despite being valid configurations of the VAM, lead to two unsafe products. Indeed, the FSM model

generated from $Configuration$ 2 is not correct: according to the metamodel, an outgoing transition must have at least one target state, which does not hold for transition t1. In the case of $Configuration$ 3, the derived product model has the incoming transition t3 without a source state, which also is incorrect with respect to the metamodel.

Even for a very simple MSPL, several unsafe product models can be derived in contradiction to the intention of an MSPL designer. In practice, specifying a correct MSPL is a daunting and error-prone activity due to the fact that the number of choices in the VAM, the number of classes and rules in the metamodel and the size of the VRM can be bigger.

The problem of safely configuring a feature or a decision model is now well understood [8]. Moreover, several techniques exist for checking the conformance of a model for a given modeling language. The connection of both parts (the VAM and the set of base models) and the management of the realization layer are still crucial issues [6, 37, 35, 18, 13].

3. GENERATING COUNTEREXAMPLES

We argue that the realization layer may concern at least two kinds of users:

- designers of MSPLs in charge of specifying the VAM, the BMs, as well as the relationships between the VAM and the BMs (VRM)(see CVL of Definition 1) ;
- developers of derivation engines in charge of automating the synthesis of model products based on a selection of features (*Choices*) (function δ of Definition 1);

Incorrect derivation engines or realization models may authorize the building of unsafe products. The majority of the existing work target scenarios in which an existing MSPL has been designed and seeks to first check its consistency, then to generate unsafe product models – pointing out errors in the MSPL. These techniques are extremely useful but assume that a generic derivation engine exists and is correct for the targeted modeling language – which is hardly conceivable in our case. Moreover, designers of MSPLs are likely to perform typical errors for a given modeling language (e.g., FSM).

3.1 Counterexamples to the Rescue

We precisely want to provide support to the two kinds of users in their activities. Specifically, we are interested on finding MSPLs that apparently would derive models that respect the domain modeling language, as they have a correct variability model and a conforming base model, but however, either their VRM or their derivation engine were incorrectly designed. Definition 2 formalizes this kind of MSPL as *counterexamples*.

Definition 2 (Counterexample of MSPL). *A counterexample CE is an MSPL in which:*

- *CVL is well-formed ;*
- *There exists at least one valid configuration in VAM:* $\mathcal{C}_{VAM} \neq \emptyset$ *;*
- $\exists\ c \in \mathcal{C}_{VAM}, \delta\ (CVL,\ c\ , BM) =\ DM'$ *such that* DM' *does not conform to its modeling language.*

The expected benefits are as follows:

- SPL designers in charge of writing CVL models, can better understand the kinds of errors that should be avoided (Figure 3 gives two "antipatterns").
- developers of derivation engines can exploit counterexamples as testing oracles and anticipate the kinds of inputs that should be properly handled by their implementation. Furthermore, more specific error reports can be generated when an MSPL is incorrect, inspired by the catalogue of counterexamples.

3.2 Overview of the Generation

In order to systematically generate counterexamples of MSPLs, we have defined a set of activities that can be performed for this purpose. Figure 4 presents an overview of the process that generates a single counterexample, as well as the input and output for the different phases. We have divided the process into four phases. The first phase is in charge of setting up the input that will be taken into account, as different activities can be performed, depending on the input. The second and third phases are responsible to generate the CVL model, respectively generating the variability abstraction model with its resolutions and the variability realization model. The fourth and last phase is the detection of the counterexample. Following, we describe each phase of the proposed approach.

3.3 Set up input

3.3.1 Generate BM

Generally, companies that use or decide to set up a product line already have an initial set of core assets. In the case of MSPLs, if the models are not available, it is common to have the metamodel and the well-formedness rules of the modeling language. Considering this, the metamodel and the rules of the domain-specific modeling language are a starting point to generate a CVL model. Our approach is adaptable to work with both cases, whether the models are available or only their metamodel. In the case they are not available, we apply randomizations over the metamodel to create random models. These random instances populate the Base Model, and their correctness is checked against the metamodel and the well-formedness rules. If a created model is not correct, this instance is discarded. In the case of the FSM modeling language, the checked well-formedness rules are: if the initial state is different of the final, if the FSM is deterministic and if all the states are reachable. On the other hand, if we already have a set of models, mutation operators are applied on these models in order to increase the number of samples. Mutations operators are basic CRUD (Create, Read, Update, Delete) operations on the base model that are applied randomly.

3.3.2 Set up parameters

Besides the input, it is also necessary to set up parameters that will be used during the process. Although Figure 4 describes the process of generating one single counterexample, we iterate the process to produce a set of counterexamples. For this reason, the first parameter to be taken into account is the stopping criteria. The stopping criteria can be specified in two different ways. The first one defines a target number of counterexamples, making the process repeat until this number is reached. The second one is to set an amount of time, stopping the process after it has elapsed. For gen-

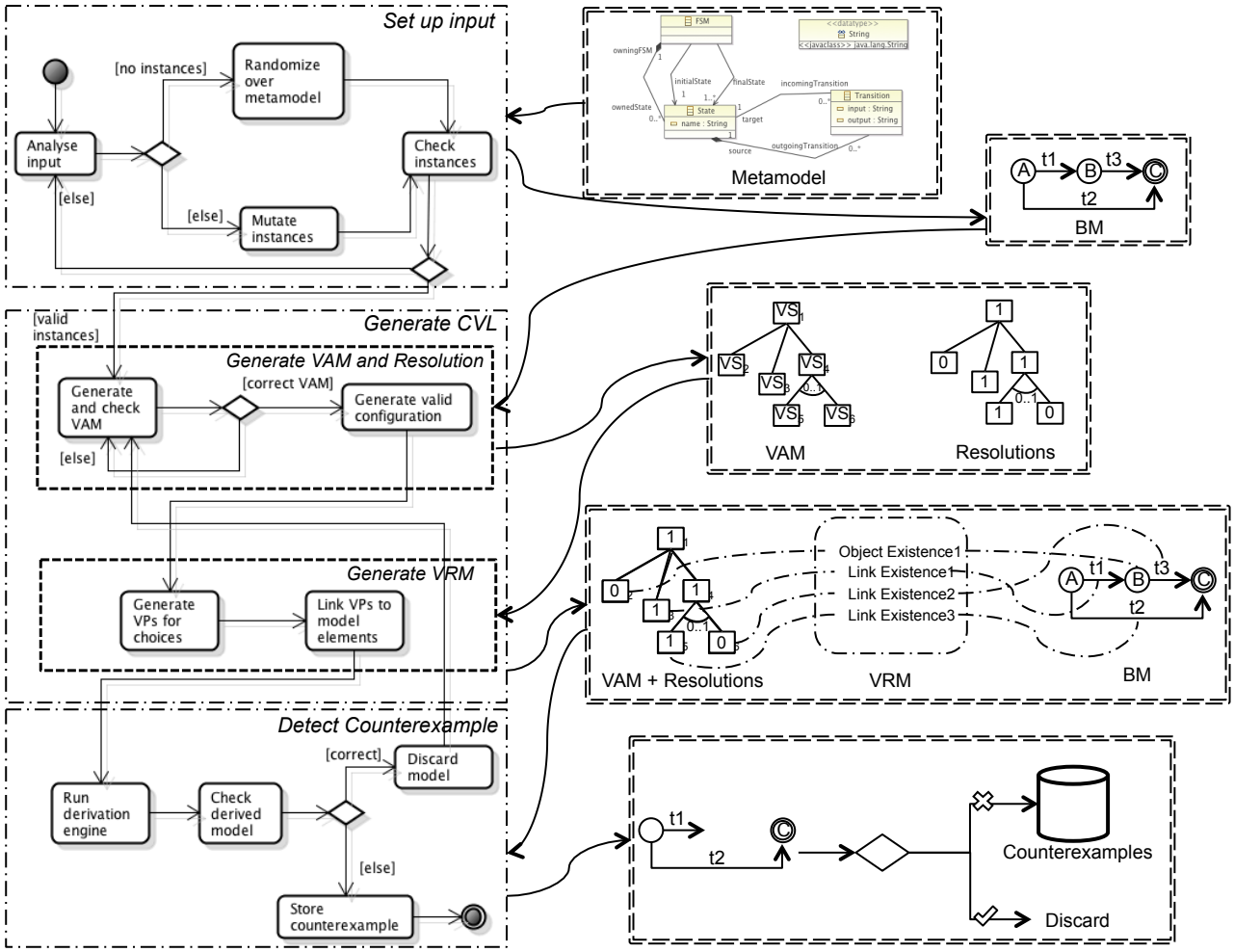

Figure 4: Overview.

erating the VAM and the VRM, the following parameters are required:

- The maximum depth of the VAM *(MAX_DEPTH)* and the maximum number of children for each $VSpec$ *(MAX_CHILDREN)*.
- The percentage of $VSpecs$ that will be linked to variation points *(LINK_PERCENT)*. For example, in Figure 4, the VAM was generated with a percentage of 66%, as four out of six $VSpecs$ are linked to VPs.

3.4 Generate VAM and Resolution

Once the BM is established and the parameters have been set, we take them as input to start the generation of the CVL model. First, we generate the VAM, creating a root $VSpec$ and its children. The number of children is decided randomly, ranging from 0 to *MAX_CHILDREN*. The $VSpec$ creation is repeated for each generated child until the *(MAX_DEPTH)* is reached or there are no more $VSpecs$ with children.

After generating the VAM, it is necessary to check its correctness, as we are not interested in wrong VAMs. For this reason, we translate the VAM to a language that can provide us a background for analysing it. The FAMILIAR language is executable and gives support to manipulate and reason about feature models [1] (we could also rely on existing frameworks like FaMa [8]). As stated in Section 2.2, the kinds of VAM we consider in this paper are amenable to boolean feature models supported by FAMILIAR. Using FAMILIAR, we check whether the variability model is valid or invalid. If it is an invalid model, we discard it and return to the VAM generation step. A resolution model is necessary in order to resolve the variability expressed in the VAM. To generate the configuration, we create the corresponding resolution CVL element for each $VSpec$. Meanwhile, random values (true or false) are set for each *ChoiceResolution* that has been created. We use standard satisfiability techniques to randomly generate a resolution, which is, by construction, a valid configuration of the VAM.

3.5 Generate VRM

Once we have a correct VAM and a correct BM, we can generate the VRM to link each other. To do this, we iterate over the set of choices in the VAM, deciding if the given choice is pointed or not by a Variation Point. This decision is done based on the *(LINK_PERCENT)* parameter. If the decision is true, we create the VP in the VRM. The type of the VP is also random. To finish the creation of the VP, we also randomize its target over the set of model elements of

the BM. Naturally, we restrict the set of the randomization with respect to the kind of VP, e.g., a *LinkExistence* has a random target randomized over the subset of BM references.

3.6 Detect Counterexample

After the aforementioned steps have been performed, we have a correct CVL model, composed by a correct VAM and a VRM created in conformance to the CVL metamodel. We also have a valid configuration c and a correct set of models composing the BM. The next step is to derive a product model using the CVL, c and the BM. If the derived model is incorrect, in other words, having $\delta\ (CVL,\ c\ , BM)$ incorrect, we have found a counterexample as states the Definition 2, and consequently, we add it to the oracle. If the model is correct then we discard it and generate a new VRM.

As we will discuss in Section 4, these counterexamples can be helpful to the domain experts in charge of designing the CVL model or developing their derivation engines for their domain.

4. EVALUATION

The goal of this evaluation is to verify the applicability and effectiveness of the proposed approach, as well as to assess important properties of the generated counterexamples. Regarding the effectiveness, we formulated the following question:

- RQ1. Can the approach generate counterexamples in a reasonable amount of time?

Then we seek to answer questions about the properties of the generated counterexamples, such as:

- RQ2. Does the number of counterexamples increase in a more complex domain?
- RQ3. With respect to the metamodel or the OCL rules, what errors are the most common in the counterexamples?
- RQ4. Is it possible to prevent the generation of counterexamples by the designer?

4.1 RQ1. (Applicability and Effectiveness)

Answering this question will allow us to know if the approach can actually generate counterexamples and how long it takes to generate a range of counterexamples.

Objects of Study. To answer RQ1, we need to apply the proposed approach to specific scenarios and verify if it effectively produces counterexamples. As a first scenario, we use the FSM modeling language that was presented in previous sections. As second and more complex scenario, we use the Ecore modeling language. We provide the corresponding metamodel and well-formedness rules as input for both scenarios. As previously mentioned, the FSM metamodel has 3 classes and 4 rules, while the Ecore metamodel has 20 metaclasses, 33 datatypes and 91 validation rules. Following the approach, we set up the parameters equally for both scenarios: the stopping criteria is set to the number of 100 counterexamples, the *MAX_DEPTH* is set to 5, the *MAX_CHILDREN* is set to 10 and the *LINK_PERCENT* is set to 30%.

Experimental Setup. Once the parameters and the input are ready, we start the automatic generation of the counterexamples. The generation was performed in a machine with a 2nd Generation Intel Core I7 processor - Extreme Edition and 16GB of 1333MHz RAM memory, running under a linux 64bit with a 3.8.0 kernel, Scala 2.9.3 and an oracle Java Runtime Environment 7.

Experimental Results. The times are shown in Figure 5, ranging from 0 to 12625 seconds. For both FSM and Ecore, we could successfully find and generate counterexamples in a reasonable time. The time for generating 10 counterexamples for the Ecore-based MSPL was approximately 15 minutes, which is acceptable, considering the complexity of the Ecore metamodel. Thus, as the target number of counterexample increases, we can confirm a linear growth of the time. Each time value is an average of 10 executions, this was done to minimize the random effect.

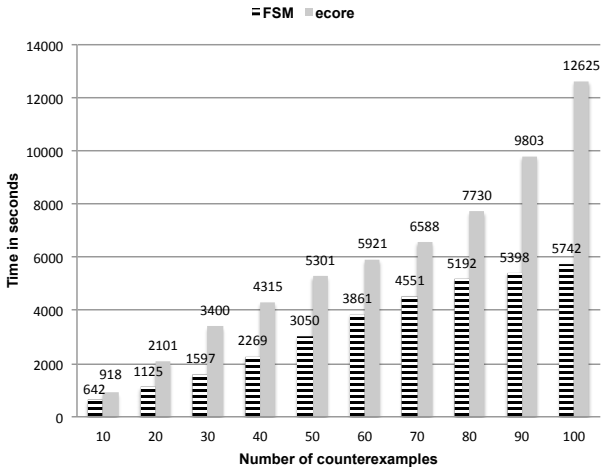

Figure 5: Counterexamples for FSM and Ecore.

4.2 RQ2. (Counterexamples vs Domain Complexity)

This research question aims at analysing the consequences of applying the approach in a more complex domain. Answering this question helps whether and to which extent it is more likely to design counterexamples (i.e., unsafe MSPLs) when the domain becomes more complex or not.

Objects of Study. To address RQ2, we compared the ratio between the number of invalid DMs and valid DMs. We made this comparison with three different modeling languages: FSM, Ecore (with the Eclipse Modeling Framework implementation) and UML (with the Eclipse UML2 project implementation. We classified these modeling languages in the following increasing sequence of complexity: FSM $<$ Ecore $<$ UML. Indeed, the FSM metamodel contains only 3 metaclasses 1 datatype and 4 validation rules. The Ecore metamodel contains 20 metaclasses, 33 datatypes and 91 validation rules. Finally, the UML contains 247 metaclasses, 17 datatypes and 684 validation rules.

Experimental Setup. For each modeling language, we applied our approach to obtain 100 counter examples, using the same parameters of the first experiment, and we collect the number of correct DMs we obtain. The evaluation was performed on the same computer of the previous experiment. For generating valid UML model, we do not create UML models from scratch, but we mutate existing UML models. We chose the footnote referred set of UML models to create

the BM[5].

Experimental Results. The experiment resulted in the generation of 469 correct DMs for 100 counterexamples for FSM, 292 correct DMs for 100 counterexamples for Ecore and 52 correct DMs for 100 counter examples for UML[6]. We can therefore verify the ratio of incorrect per correct derived models. In the case of FSM, the ratio is 1 incorrect DM to 5 correct DMs, while in the case of Ecore, this ratio is 1 to 3, and for UML the ratio is 1 to 0,5. These results provide evidence that, as the domain modeling language becomes more complex, the chance to get a correct DM becomes lower. In a sense, it confirms the relevance of our procedure for generating counterexamples. More importantly, the practical consequence is that the designer is likely to produce much more unsafe MSPLs when the targeted modeling language is complex.

4.3 RQ3. (Nature of the errors)

The purpose here is to evaluate whether the errors are a violation to the structural properties of the metamodel or to the validation rules (i.e., OCL rules). Answering this question can help to understand which part of the modeling language is more likely to reveal more errors. Hence, we conducted the following experiment to investigate the research question.

Objects of Study. To identify the nature of the errors in the counterexamples, we used the generation of the 100 counterexamples for the three modeling languages that were previously used to answer RQ2. Our object of study is the quantity of counterexamples with errors violating the metamodel or the OCL rules.

Experimental Setup For each modeling language, we applied our approach to obtain 100 counterexamples under the same parameters, and then we identify in which part of the modeling language definition is the error of the DM. The evaluation was performed using the same computer of the previous experiment.

Experimental Results For the FSM language, among the 100 counterexamples, we generate 10 models that do not conform to the metamodel and 90 models that violate one of the validation rules. For the Ecore modeling language, among the 100 counter examples, we generate 64 models that do not conform to the metamodel and we generate 36 models that violate one of the validation rules. For the UML modeling language, among the 100 counter examples, we generate 22 models that do not conform to the metamodel and we generate 78 models that violate one of the validation rules.

We now correlate these numbers with the properties of the modeling language. FSM contains only three structural rules (i.e., a state-machine must contain at least one state, one initial state and at least one final state). Most of the errors are the validation rules that are violated. Ecore contains much more structural rules (mainly lower case constraints for cardinality). Therefore lots of errors comes from structural inconsistencies. Finally UML contains so many validation rules that it is unfeasible to create a valid UML model randomly. (That is why we used *mutation* from a set of valid UML models.) For this case we obtained much more DMs that violate validation rules expressed in OCL.

[5]http://goo.gl/kC0sx

[6]Source code for the experiment is available at http://goo.gl/PgkrL

Yet, it is hard to draw definitive conclusions on whether structural or validation rules expressed in OCL participate the most in generating incorrect MSPLs. The results indicate that the kind of errors that are the most common in the counterexamples depend mainly on the domain modeling language (Ecore vs UML). It is well known, for instance, that some OCL rules can be refactored as structural constraints in the metamodel. In a sense, it partly confirms – in the context of CVL – some of the results exposed in [9] showing there exists different "styles" of expressing business or domain-specific rules within a metamodel.

4.4 RQ4. (Antipattern Detection)

The purpose of RQ4 is to evaluate the feasibility of expressing validation rules on the triplet VAM, BM, VRM to decrease the risk of creating invalid DMs from a a valid CVL model and a co BM, being $\mathcal{C}$ the set of possible valid configurations for a valid VAM. This question helps to know if it is possible for a domain designer to detect early "bad" CVL models (acting as "antipatterns") for a given domain.

Objects of Study. To evaluate this research question, we created two validation rules to detect antipattern for the FSM modeling language. These rules constrain the fact of having an object existence that target the initial state of an FSM, and also a substitution between a final state and an initial state. These rules have been implemented in Scala and can be written in few lines using an OCL writing style, as shown in Listing 1.

Listing 1: Antipattern rules for FSM

```
1  def checkVRM(f:FSM,vrm: VPackage):Boolean = {
2      vrm.asInstanceOf[VPackage].
           getPackageElement().foreach(e=> {
3          if (e.isInstanceOf[ObjectSubstitution])
               {
4          var p = e.asInstanceOf[
               ObjectSubstitution].
               getPlacementObject().getReference()
5          var p1 = e.asInstanceOf[
               ObjectSubstitution].
               getReplacementObject().getReference
               ()
6          if ((f.getFinalState().contains(p) && f.
               getInitialState().equals(p1))|| (f.
               getFinalState().contains(p1) && f.
               getInitialState().equals(p))) return
                false;
7           }else if (e.isInstanceOf[
                ObjectExistence]){
8             e.asInstanceOf[ObjectExistence].
                  getOptionalObject().foreach(p=> {
                  if (f.getInitialState().equals(p.
                  getReference())) return false;})
                  }})
9      return true}
```

Experimental Setup. For the FSM modeling language, we applied our approach to obtain 100 counterexamples and we compare the number of valid DMs we obtain either checking the antipatterns rules or not. The evaluation was performed on the same computer that the previous experiment, as well as with the same parameters.

Experimental Results. The experimental results show that we generate 1860 correct DMs for 100 counterexample for FSM when the antipattern rules for CVL are activated, against 469 correct DMs for 100 counter examples for FSM when the CVL validation rules for CVL are not activated. For this domain, writing only 2 rules on the triplet of VAM, VRM, BM allowed us to decrease 4 times the risk of generating an invalid DM. Therefore, it is feasible to detect

identified antipatterns using our approach, writing validation rules that detect *a priori* and therefore earlier these errors.

4.5 Discussion

Besides the checking operations, the time results presented in Figure 5 are mainly dependent on the following factors:

1. The time to generate a correct set of models to compose the BM ;
2. The time to generate a correct VAM ;
3. The time to generate a VRM ;

These three factors are resulting from the generality and the full automation of our approach that does not require any input models. The approach gives the ability of finding possible design errors without having yet designed the MSPL. This allows users to explore the design space of an MSPL, given a modeling language – this is the main scenario we initially target. However, it is possible to *predefine some inputs*. It could enhance the scalability of our generative process, since there is no need to spend time in generating these inputs. It may be the case when a designer of an MSPL already has a established BM. Another possible situation is when the VAM has been previously designed, as it is often one of the starting points of an MSPL. Therefore, we can claim that the conducted experiment address the *worst case* input for our approach. Consequently, our approach is sufficiently generic, as it does not assume that it is always the case of having a VAM or the BM as input. In addition, because it is fully automated, the approach does not demand a great effort to be used. Another benefit of predefining some inputs is that we could address other scenarios, like the debugging of an existing MSPL or the definition of various realization models given predefined BMs and VAMs.

By definition, an MSPL is a complex structure, composed by different connected models. This characteristic makes hard to design a correct MSPL, as errors can occur in any design phase. Given this great proneness to error, it is relevant to discuss the causes and to reason where is the lack of safety. For this purpose, we can analyse and give a rationale about two questions:

1. How a VAM and its analysis tools check and prevent configurations that result in incorrect *DMs*?
2. Is the fact of a derivation operator generate an incorrect DM fault of the own derivation operator (derivation engine) or is it fault of how it was invoked (realization model)?

Regarding the first question, it seems unfeasible to have a generic checker that, for any domain, could detect whether a configuration derives or not an incorrect model. It is rather needed to customize a derivation engine and/or a consistency checker (e.g., a simulator [40]) that takes into account the syntactic and semantic rules of the domain. Likewise, faulty configurations, currently not supported by the MSPL, could be better identified and located. From this aspect, counterexamples can help to devise such specific simulators and oracles. For the second question, we can argue that there is a trade-off between the expressiveness of the realization model and the safeness of the derivation. On the one hand, if more restrictions are applied to the derivation engine, we limit what could be generated. Also, a realization design can be wrong in one domain, but correct in another. On the other hand, if the derivation engine is not customized to address the specific meanings of a modeling language, then it is necessary to have checking mechanisms for the VRM that takes into account the syntax and semantics of the domain. More practical investigations are needed to determine when to customize the derivation engine or when to develop specific checking rules for the VRM. Counterexamples can be used for implementing both solutions.

5. RELATED WORK

MSPLs. Different variability modeling approaches have been proposed. *Annotative* approaches derive concrete product models by activating or removing parts of the model. Variant annotations define these parts with the help of, for example, UML stereotypes [41] or presence conditions [13, 18, 15]. *Compositional* approaches associate model fragments with product features that are then composed for a particular configuration (i.e., combination of features). For instance, Perrouin *et al.* offer means to automatically compose modeling assets based on a selection of desired features [29]. Apel *et al.* propose to revisit superimposition technique and analyze its feasibility as a model composition technique [4]. Dhungana *et al.* provide support to semi-automatically merge model fragments into complete product line models [19]. Annotative and compositional approaches have both pros and cons. Voelter and Groher illustrated how negative (i.e., annotative) and *positive* (i.e., compositional) variability [38] can be combined. *Delta modeling* [32, 11] promotes a modular approach to develop MSPL. The deltas are defined in separate models and a core model is transformed to a new variant by applying a set of deltas.

The variability realization layer of CVL, as exposed in Section 2.2, provides both the means to support annotative, compositional or transformational approaches [36, 25]. Therefore we believe our work is applicable to a wide range of existing MSPL approaches.

Verification of SPLs. Some techniques specifically address the problem of verifying SPL or MSPL. The objective is usually to guarantee the *safe composition* of an SPL, that is, all products of an SPL should be "safe" (syntactically or semantically). In [37], Batory *et al.* proposed reasoning techniques to guarantee that all programs in an SPL are type safe: i.e., absent of references to undefined elements (such as classes, methods, and variables). At the modeling level, Czarnecki *et al.* presented an automated verification procedure for ensuring that no ill-structured template instance (i.e., a derived model) will be generated from a correct configuration [18]. In [13, 12], the authors developed efficient model checking techniques to exhaustively verify a family of transition systems against temporal properties. Asirelli *et al.* proposed a framework for formally reasoning about modal transition systems with variability [6]. In [2], Alfeérez *et al.* applied VCC4RE (for Variability Consistency Checker for Requirements) to verify the relationships between a feature model and a set of use scenarios. Zhang *et al.* [40] developed a simulator for deriving product models as well as a consistency checker. Svendsen *et al.* present an approach for automatically generating a testing oracle for train stations expressed in CVL [35].

Some of this work generate counterexamples when the property of safe composition is violated, typically for pre-

senting to a developer an error in the specification of an SPL. In our approach, the goal is not to produce unsafe products of an *existing* MSPL, but to generate unsafe MSPLs. We do not assume variability models, models or configurations as inputs and the approach is fully automated. We thus target scenarios that go beyond debugging an existing MSPL. Our objective is rather to *prevent* the unsafe specification of realization models, i.e., generated counterexamples act here as "anti-patterns" that should prevent practitioners in specifying unsafe MSPLs. Another important difference is that verification techniques previously described assume that the derivation engine is correct. In our context, we cannot formulate the same hypothesis and have rather the crucial needs to implement new and robust derivation engines – each time a new modeling language is used in the MSPL. We provide quantitative evidence that the specificity of the modeling language should be taken into account. The generation of counterexamples aims at producing testing "oracles" and guide developers when building a derivation engine.

Techniques for combinatorial interaction testing of feature models (the VAM part of CVL) [27, 30, 23] have been proposed. As future work we plan to consider their use as part of our generation process.

Verification and debugging of models. Numerous techniques have been proposed for debugging or verifying consistency of models or model transformations (e.g., [28, 7, 20]. These works do not address specific issues of MSPL engineering, especially those related to the realization layer.

6. CONCLUSIONS AND FUTURE WORK

Because of the combinatorial explosion of possible derived variants, the great variety and complexity of its models, correctly designing a Model-based Software Product Line (MSPL) has proved to be challenging. It is easy for a developer to specify an incorrect set of mappings between the features/decisions and the modeling assets, thus authorizing the derivation of unsafe product models in the MSPL. In this paper, we have presented a systematic and fully automated approach to explore the design space of an MSPL. The main objective of the approach was to generate counterexamples of MSPLs, i.e., MSPLs that can produce invalid product models. This kind of MSPL can be used to test derivation engines or provide examples of invalid VRMs, which could serve as a basis to establish antipatterns for developers.

For this purpose, we have formalized the concepts of an MSPL, based on the Common Variability Language (CVL), as well as the concept of a counterexample. We explained in details each step of our generative approach and illustrated it with a running example. Afterwards, we performed experiments to assess the applicability and effectiveness of the approach. The conducted experiments allowed us to evaluate how efficiently the approach is when applied to different modeling languages. We could successfully generate counterexamples for each modeling language in a reasonable amount of time. In addition, we explored the natures of errors found in the counterexamples and our ability to detect antipatterns. We reported on our experience and findings that we cannot draw definitive conclusions for some aspects of our study, thus calling for more investigations.

In particular we will explore further the idea of exploiting counterexamples for various purposes (e.g., for developing and testing a derivation engine). It will necessarily involve users, i.e., either developers of tool support for CVL or practitioners in charge of specifying MSPLs. For this purpose, we plan to conduct user experiments in the context of prototyping the use of CVL with Thales on dedicated domain-specific modeling languages for system engineering.

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP) Special issue on programming languages*, page 22, 2013.

[2] M. Alférez, R. E. Lopez-Herrejon, A. Moreira, V. Amaral, and A. Egyed. Supporting consistency checking between features and software product line use scenarios. In K. Schmid, editor, *ICSR*, volume 6727 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2011.

[3] G. Antoniol, A. Bertolino, and Y. Labiche, editors. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012*. IEEE, 2012.

[4] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In R. F. Paige, editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2009.

[5] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July/August 2009.

[6] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Formal description of variability in product families. In E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors, *SPLC*, pages 130–139. IEEE, 2011.

[7] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

[8] D. Benavides, S. Segura, and A. Ruiz-cort. Automated Analysis of Feature Models 20 Years Later : A Literature Review. *Information Systems*, 35(6), 2010.

[9] J. J. Cadavid, B. Baudry, and H. A. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In Antoniol et al. [3], pages 131–140.

[10] L. Chen, M. A. Babar, and N. Ali. Variability management in software product lines: a systematic review. In *SPLC'09*, pages 81–90, 2009.

[11] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proceedings of the 9th GPCE'10 conference*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[12] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*, pages 335–344. ACM, 2010.

[14] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[15] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, volume 3676 of *LNCS*, pages 422–437, 2005.

[16] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek. Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 126–127, New York, NY, USA, 2005. ACM.

[17] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. 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*, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM.

[18] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06*, pages 211–220. ACM, 2006.

[19] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010.

[20] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Trans. Software Eng.*, 37(2):188–204, 2011.

[21] J. B. F. Filho, O. Barais, J. Le Noir, and J.-M. Jézéquel. Customizing the common variability language semantics for your domain models. In *Proceedings of the VARiability for You Workshop*, VARY '12, pages 3–8, New York, NY, USA, 2012. ACM.

[22] F. Fleurey, Ø. Haugen, B. Møller-Pedersen, A. Svendsen, and X. Zhang. Standardizing Variability - Challenges and Solutions. In *SDL Forum*, pages 233–246, 2011.

[23] A. Gotlieb, A. Hervieu, and B. Baudry. Minimum pairwise coverage using constraint programming techniques. In Antoniol et al. [3], pages 773–774.

[24] O. M. Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Technical report, Nov. 2007.

[25] O. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.

[26] F. Heidenreich, P. Sanchez, J. Santos, S. Zschaler, M. Alferez, J. Araujo, L. Fuentes, U. K. amd Ana Moreira, and A. Rashid. Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114, 2010.

[27] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In E. S. de Almeida, C. Schwanninger, and D. Benavides, editors, *SPLC (1)*, pages 46–55. ACM, 2012.

[28] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 128–137, Washington, DC, USA, 2008. IEEE Computer Society.

[29] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC'08*, pages 339–348. IEEE, 2008.

[30] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.

[31] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[32] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.

[34] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[35] A. Svendsen, Ø. Haugen, and B. Møller-Pedersen. Specifying a testing oracle for train stations - going beyond with product line technology. In J. Kienzle, editor, *MoDELS Workshops*, volume 7167 of *LNCS*, pages 187–201. Springer, 2011.

[36] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, Ø. Haugen, B. Møller-Pedersen, and G. K. Olsen. Developing a software product line for train control: A case study of cvl. In J. Bosch and J. Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2010.

[37] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07*, pages 95–104, New York, NY, USA, 2007. ACM.

[38] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC'07*, pages 233–242. IEEE, 2007.

[39] J.-L. Voirin. Method & tools to secure and support collaborative architecting of constrained systems. In 18^{th} *International Symposium of the INCOSE*, Utrecht, Netherlands, June 2008. International Council on Systems Engineering.

[40] X. Zhang and B. Møller-Pedersen. Towards correct product derivation in model-driven product lines. In Ø. Haugen, R. Reed, and R. Gotzhein, editors, *SAM*, volume 7744 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2012.

[41] T. Ziadi and J.-M. Jézéquel. Software product line engineering with the uml: Deriving products. In T. Käkölä and J. C. Dueñas, editors, *Software Product Lines*, pages 557–588. Springer, 2006.