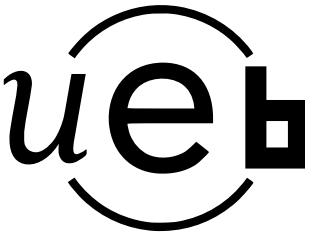
Leveraging model-based product lines for systems engineering

João Bosco Ferreira Filho

ANNÉE 2014



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

João Bosco FERREIRA FILHO

préparée à l'unité de recherche INRIA
Rennes Bretagne Atlantique

Leveraging model-based product lines for systems engineering

Thèse soutenue à Rennes
le 3 Décembre 2014

devant le jury composé de :

Øystein HAUGEN
Professor, SINTEF and Department of Informatics, University of Oslo / *Rapporteur*

Philippe COLLET
Professeur, Université Nice Sophia Antipolis / *Rapporteur*

Mathieu ACHER
Maître de conférences, Université Rennes 1 / *Examinateur*

Claude JARD
Professeur, Université de Nantes / *Examinateur*

Benoit BAUDRY
Chercheur, INRIA Rennes - Bretagne Atlantique / *Directeur de thèse*

Olivier BARAIS
Maître de conférences, Université Rennes 1 / *Co-directeur de thèse*

Résumé en Français

Les applications et les systèmes logiciels sont devenus omniprésents dans notre quotidien, qu'il s'agisse des véhicules de transport, sur nos routes, à la maison dans nos appareils multimédias mais aussi notre électroménager voire les bâtiments eux-mêmes, etc. Ces systèmes cyber-physiques [COM09] (i.e., les systèmes avec des éléments computationnels collaboratifs qui contrôlent des éléments physiques) imposent aux ingénieurs de travailler avec des logiciels de taille très importante -- une nouvelle voiture a environ 100 millions de lignes de code. Pour faciliter le développement de tels systèmes, les ingénieurs adoptent l'approche *diviser pour régner* et de séparation des préoccupations. Le système est donc développé en utilisant de nombreux et différents langages dédiés et impliquant des différentes parties prenantes.

Cette observation est valable pour la plupart des acteurs industriels, quelque soit leur secteur d'activité, et est particulièrement vérifiée dans le cas de Thales: une entreprise multinationale qui opère dans les domaines complexes, tels que l'aéronautique, l'espace, les systèmes de défense ou encore le transport. Leurs ingénieurs utilisent différents langages dédiés à fin de développer des ensembles intégrés de systèmes. Ces langages sont construits dans un ensemble de représentations dédiées à l'analyse de problèmes spécifiques à un domaine et s'appuient sur l'ingénierie dirigée par les modèles (MDE) [Sch06].

Ces entreprises ont également besoin de construire des versions\variantes légèrement différentes d'un même système. Ces versions partagent des points communs et des différences, le tout pouvant être géré à l'aide d'une approche ligne de produits (SPL – Software Product Line, même si on pourrait dorénavant remplacer la spécificité du logiciel par une encapsulation plus large, à savoir le système lui même) [CN01, PBvdL05]. L'objectif principal d'une SPL est d'exploiter la personnalisation de masse, dans laquelle les produits sont réalisés pour répondre aux besoins spécifiques de chaque client [BSRc10]. Pour répondre à ce besoin de personnalisation, les systèmes doivent être étendus de manière efficace, ou modifiés, configurés pour être utilisé dans un contexte particulier [SvGB05, CBA09].

Une approche encourageante consiste à connecter l'approche MDE (séparation des préoccupations, modélisation et langages dédiés) à l'approche SPL (gestion de la variabilité) – les SPL basées sur les modèles (MSPL). Les produits de la ligne de produits sont ainsi exprimés sous la forme de modèles conformes à un méta-modèle et ses règles de bonne formation. De nombreuses techniques de MSPL ont été proposées ([PBvdL05, PKGJ08, HSS+10, CA05a, CHS+10b, CP06, ZJ06, VG07]). Ces approches sont composées généralement par **i)** un modèle de variabilité (e.g., un modèle de caractéristiques ou un modèle de décision), **ii)** un modèle de base (e.g., une machine à

états, un diagramme de classes) exprimé dans un langage de modélisation spécifique (par exemple, le langage de modélisation UML de l'OMG (Unified Modeling Language [Gro07]), et **iii)** une couche de réalisation qui met en correspondance les points de variations et les éléments d'un modèle de base.

Basée sur une sélection de caractéristiques souhaitées dans le modèle de variabilité, un moteur de dérivation peut synthétiser automatiquement des modèles personnalisés – chaque modèle correspondant à un produit individuel de la ligne de produits. Dans ce contexte de MSPLs, le langage \emph {CVL (Common Variability Language} [FHMP+11a] a récemment émergé comme un effort de standardisation et la promotion des MSPLs. Dans cette thèse, nous adoptons CVL comme langage de construction de MSPLs.

Challenges

L'espace de conception, l'environnement du système logiciel que l'on construit (i.e., l'ingénierie du domaine) d'une MSPL est extrêmement complexe à gérer pour un ingénieur. Tout d'abord, **le nombre possible des produits d'une MSPL est exponentielle** au nombre d'éléments ou de décisions exprimé dans le modèle de variabilité. Ensuite, **les modèles de produits dérivés doivent être conformes à de nombreuses règles liées au domaine métier mais aussi aux langages de modélisation utilisés**. Par exemple, UML présente 684 règles de validation dans l'implémentation EMF. Par conséquent, un développeur doit comprendre les propriétés intrinsèques du langage de modélisation pour concevoir une MSPL. Troisièmement, **le modèle de réalisation qui relie un modèle de variabilité et un modèle de base peut être très expressif**, spécialement dans le cas de CVL. La gestion des modèles de variabilité et de modèles de conception est une activité non triviale, relier les deux parties et par conséquent l'ensemble des modèles est une tâche non négligeable et susceptible à erreurs.

En plus de ces défis intrinsèques aux MSPLs, il faut ajouter que les ingénieurs système utilisent différents langages de modélisation dédiés dans le cadre de projets pour la réalisation de systèmes critiques. Comme les modèles sont conformes à leurs propres règles de bonne formation et des règles spécifiques du domaine, chaque utilisation d'un nouveau langage de modélisation pour l'élaboration d'une MSPL implique la révision de la couche de réalisation.

Nous pouvons résumer ces défis autour de cinq questions de recherche que nous abordons dans cette thèse. Ces questions sont la conséquence de l'effort de gestion de la variabilité, en ingénierie système chez Thales, autour d'un scénario utilisant plusieurs langages et avec une nécessité d'avoir des modèles corrects.

1) Comment fournir une aide dès la conception de nouvelles MSPLs avec de nouveaux langages dédiés ?

2) Comment personnaliser le support de dérivation de produit de CVL ?

3) Comment amener la séparation des préoccupations en matière de modélisation de la variabilité en ingénierie système ?

4) Comment intégrer la gestion de la variabilité en ingénierie système d'une manière transparente et non intrusive ?

5) Comment exploiter la génération valide des produits ?

Contributions

Nos contributions sont basées sur le fait qu'une solution générique, pour tous les domaines, et qui dérive des modèles corrects n'est pas réaliste, surtout si on prend en considération le contexte des systèmes complexes décrits précédemment. Par conséquent, au lieu d'essayer de trouver la solution miracle (par exemple, un vérificateur générique de modèles de ligne de produits ou une notation unifiée pour exprimer les MSPLs en ingénierie système), nous prenons le parti inverse et proposons une approche indépendante du domaine pour générer des contre-exemples de MSPLs, révélant des erreurs de conceptions de modèles et supportant les parties prenantes à construire de meilleures MSPLs et des mécanismes de dérivation plus efficaces.

Plus précisément, **la première et principale contribution de la thèse est un processus systématique et automatisé, basé sur CVL, pour la recherche aléatoire de contre-exemples de MSPL dans un langage donné.** *Les contre-exemples* sont des exemples de MSPLs qui autorisent la dérivation de modèles invalides, syntaxiquement ou sémantiquement, malgré une configuration valide dans le modèle de variabilité.

Ces contre-exemples visent à révéler des erreurs ou des risques – soit dans le moteur de dérivation ou dans le modèle de réalisation – aux parties prenantes de MSPLs. D'une part, ces contre-exemples font office d'oracles de tests pour augmenter la robustesse des mécanismes de vérification de la MSPL. Les développeurs peuvent alors utiliser des contre-exemples pour prévoir des valeurs limites et les types de MSPLs qui sont susceptibles de permettre des dérivations erronées. D'autre part, les parties prenantes peuvent répéter le même type d'erreurs lors de la spécification des correspondances entre un modèle de variabilité et d'un modèle de base. Les contre-exemples agissent alors comme des anti-patrons qui devraient éviter de mauvaises pratiques ou diminuer le nombre d'erreurs pour un langage de modélisation dédié.

Nous validons l'efficacité de ce processus autour de trois formalismes (UML, Ecore et une simple machine à états finis) à différentes échelles (jusqu'à 247 méta-classes et 684 règles) et différentes façons d'exprimer les règles de validation. De plus, nous l'appliquons

dans un scénario industriel réel (en partenariat avec Thales Research & Technology) démontrant comment notre approche s'exécute dans la pratique.

Nous explorons aussi l'hypothèse exposée ci-dessus : qu'un moteur de dérivation générique ou un support de base pour la gestion de la couche de réalisation est susceptible d'autoriser MSPLs incorrectes. Nous discutons dans quelle façon les contre-exemples pourraient guider les praticiens lors de: la personnalisation des moteurs de dérivation, la mise en œuvre des règles de vérification qui empêchent la création d'un modèle CVL incorrect, ou tout simplement lors de la spécification d'un MSPL. Les techniques génératives et l'étude exploratoire suggèrent l'utilisation de solutions au courant de la sémantique des langages de modélisation ciblés lors de l'élaboration des MSPLs.

La seconde contribution de la thèse est un étude sur les mécanismes pour étendre la sémantique des moteurs de dérivation, offrant une approche basée sur des modèles à fin de personnaliser leurs sémantique opérationnelle. Cette approche facilite la définition des opérateurs de réalisation corrects par l'ingénieur du domaine. Il s'agit d'une étape naturelle après les preuves empiriques obtenues par le procédé automatisé mentionné comme première contribution.

Dans la troisième contribution de la thèse , nous extrapolons les limites de langages de modélisation. Nous présentons une étude empirique à large échelle sur le langage Java qui comprend : une évaluation automatisée de tous ses éléments de langage et comment les opérateurs de réalisation de CVL sont pertinents ou non à modifier et faire varier les programmes Java. Nous proposons une classification complète des transformations de variabilité. Nos données statistiques aident à caractériser quelles constructions du langage sont susceptibles de varier ou nécessitent des transformations spécifiques. D'un point de vue qualitatif, nous examinons et analysons les variantes de programmes Java générées à l'aide d'outils dédiés. Cette expérience sert aussi à démontrer les premières étapes après avoir généré les contre-exemples: l'analyse, l'ordonnancement et la classification.

La quatrième et dernière contribution de la thèse est une méthodologie pour intégrer notre travail dans une organisation qui cherche à mettre en œuvre les lignes de produit logiciels basées sur des modèles pour l'ingénierie des systèmes. Nous nous concentrons sur la réponse à l'événement de devoir concevoir une MSPL pour un nouveau langage dédié, montrant les différentes activités, comment elles se succèdent et les rôles impliqués dans chacune.

Contents

Introduction

Context

Software and Systems are becoming increasingly essential for daily life; they are omnipresent in the different transportation, home-appliances, civil infrastructures, entertainment or healthcare devices. These Cyber-Physical Systems [COM09] (i.e., systems of collaborating computational elements controlling physical elements) impose engineers to deal with massive pieces of software – a typical new car has about 100 million lines of code [1]. To ease the development of such systems, engineers adopt a *divide and conquer* approach: each concern of the system is engineered separately, with several domain specific languages and stakeholders.

This is the case in many companies in different industry sectors, being also true in Thales [2]: a large company actuating in complex domains, such as aerospace, space, defence and transportation. Their stakeholders use numerous domain specific modeling languages to develop integrated sets of systems. These languages are built within a set of dedicated representations to analyze domain-specific problems and they rely on the Model-driven Engineering (MDE) [Sch06] paradigm.

On the other hand, these companies also need to construct slightly different versions/variants of a same system; these variants share commonalities and variabilities that can be managed using a Software Product Line (SPL) [CN01, PBvdL05] approach. The main goal of an SPL is to leverage mass customization, in which products are made systematically to meet individual customer's needs [BSRc10]. To meet this need for customization, systems have to be efficiently extended, changed or configured for use in a particular context [SvGB05, CBA09].

A promising approach is to ally MDE with SPL –*Model-based SPLs (MSPL)*– in a way that the products of the SPL are expressed as models conforming to a metamodel and well-formedness rules. Numerous MSPL techniques have been proposed (e.g., see [PBvdL05, PKGJ08, HSS+10, CA05a, CHS+10b, CP06, ZJ06, VG07]). 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) [Gro07]), and *iii)* a realization layer that maps and transforms variation points into model elements. Based on a selection of desired

[1] `http://www.wired.com/2012/12/automotive-os-war/`

[2] `http://www.thalesgroup.com/`

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)* [FHMP+11a] has recently emerged as an effort to standardize and promote MSPLs; it is our adopted language for constructing MSPL.

Challenges

The *design space* (i.e., domain engineering) of an MSPL is extremely complex to manage for an engineer. 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 have to conform 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**, specially in the case of CVL. Managing variability models and design models is a non-trivial activity. Connecting both parts and therefore managing all the models is a daunting and error-prone task.

Added to these intrinsic MSPL challenges, we have the fact that the field of systems engineering uses many different modeling languages and is often critical. Since models conform to their own well-formedness rules and domain-specific rules. Each time a new modeling language is used for developing an MSPL, the realization layer should be revised accordingly.

Overall, we can summarize the challenges we address in this thesis in five research questions. They are consequence of leveraging variability management for systems engineering in a scenario with multiple languages and with the need for safe models.

1. How can we provide early assistance for designing MSPLs for new languages?
2. How can we customize the derivation support of CVL?
3. How can we provide separation of concerns in variability modeling for systems engineering?
4. How can we integrate variability management in systems engineering in a non-intrusive and seamless way?
5. How can we leverage safe generation of products?

Contributions

Our contributions are based in the fact that a *one-size-fits-all* support for deriving correct models in systems engineering is unfeasible, specially when considering such complex context previously described. Therefore, instead of trying to come up with the silver bullet (e.g., a generic checker of product line models or a unified notation to express

MSPLs in systems engineering), we provide a domain-independent approach to generate counterexamples of MSPLs, revealing dangerous designs and assisting stakeholders to construct better MSPLs and derivation mechanisms.

Specifically, **the first and core contribution is a systematic and automated process, based on CVL, to randomly search for counterexamples of MSPL of a given language.**

Counterexamples are examples of MSPLs that authorize the derivation of syntactically or semantically invalid product models despite of 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 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. In addition, we apply it in a real industry scenario (a partnership with Thales Research & Technology), demonstrating how our approach performs in practice. 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. 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.

Our second contribution is a study on the mechanisms to extend the semantics of derivation engines, providing a model-based approach to customize their operational semantics. This approach eases the definition of safe realization operators by the domain engineer. It is a natural step after the empirical evidences obtained by the automated process mentioned as first contribution.

As a third contribution, we extrapolate the limits of modelling languages. We perform a substantial empirical study on Java: an automated assessment of all its language constructs and how CVL-based realization operators adequate or not to vary Java programs. We provide a comprehensive classification of variability transformations. Statistical data help to characterize which language constructs are likely to vary or require specific transformations. From a qualitative perspective, we review and analyze the resulting Java variants with the help of dedicated tools. This experiment also serves to demonstrate the first steps after having generated counterexamples: analysing, ordering and categorizing them.

Our fourth and final contribution is a methodology to integrate our work into any organization that seeks to implement model-based software product

lines for systems engineering. We focus on the response to the event of having to engineer an MSPL for a new DSL, showing the different activities, how they follow each other and the roles involved in each of them.

Plan

The remainder of this thesis is organized as follows.

Chapter 1 contextualizes this thesis, situating it inside the research of systems engineering. We show the main challenges of leveraging model-based variability management in a context of multiple domain specific languages, which describes the scenario of our industry partner.

Chapter 2 presents the foundations of Software Product Lines and Model-driven Engineering, both fields serving as basis for understanding the state of the art of Model-based Software Product Lines. We then explore the state of the art of variability modeling and analysis, identifying how the approaches tackle the raised challenges and which challenges need further investigation.

Chapter 3 presents the core contribution: our exploratory approach to find counterexamples of MSPLs. We discuss its validity on practical examples and its applicability in an industrial case. We also present the tooling support that implements the approach.

Chapter 4 shows how one can customize realization operators of MSPLs to build safer derivation engines, showing the different ways of adding new semantics obtained from the knowledge after applying the approach of Chapter 3.

Chapter 5 is an empirical study on applying CVL to Java, using the realization operators to transform programs and studying their feasibility. This chapter can be seen as an extended validation of Chapter 3 and of CVL variation points applied to fine-grained program elements and, finally, as a valuable study on the variability of the Java language constructs itself.

Chapter 6 presents the roles and activities of a methodology to integrate our work into an organization. The starting point of the methodology is the need for engineering an MSPL to a new DSL, embracing the main contributions of the thesis.

Chapter 7 concludes the thesis by synthesizing the advances that it brings to leverage product line in systems engineering. It also discusses the long- and short-term perspectives of future research related to the thesis.

Part I

Context and State of the Art

Chapter 1

Context

In this chapter, we contextualize the thesis by presenting the scenario of Systems Engineering: a complex development activity that involves various stakeholders and technologies. In Section 1.1, we show that the diversity of languages is an important factor in this scenario, exemplifying with the case of the Thales group (see Section 1.1.2). We motivate that systems engineering could benefit from a product line approach once the variability inside these languages is externally handled (Section 1.2), by using the Common Variability Language (CVL).

Continuing, we present the challenges of managing variability in such a complex context, justifying and illustrating the current practice of specializing the derivation semantics of CVL (Section 1.3), which makes the challenge of deriving correct products even harder (Section 1.4). We then conclude that engineering software product lines for systems engineering needs special assistance (Section 1.5) and, finally, we synthesize the challenges in Section 1.6.

1.1 Systems Engineering: a tale of many languages and variants

The system of a typical new car has about 100 million lines of code[1]; it is inconceivable to make, manage or evolve such a big and complex piece of software by considering it as a monolithic system – a *divide and conquer* approach is imperative to succeed. The need for this division outperforms the problem of modularizing and separating concerns of the system and its software, it also involves the issue of dealing with several different stakeholders in charge of conceiving and engineering each part [SR09].

In the Systems Engineering field, stakeholders are often domain specialists (e.g, in the automotive domain there are engineers responsible for: safety, vehicle dynamics, performance, drivability, etc.); each one of them needs to deal with particular information of their particular domain, and they are not necessarily concerned by the other domains. Having one general language to design all parts and reason about all domains

[1] `http://www.wired.com/2012/12/automotive-os-war/`

is inefficient and hard to manage. Therefore, they use many specialized languages to build systems for their field. The use of Domain-Specific Languages (DSL) allows to abstract from unneeded complexity and to have efficient dedicated tooling.

1.1.1 Model-driven Engineering in the state of practice of DSLs

One way to implement DSLs is using Model-driven Engineering (MDE) [Sch06]. Instead of having a grammar specifying the abstract syntax of a language, a Domain-Specific Modeling Language (DSML) uses a metamodel to express the concepts of a domain. Metamodels are the central artifact of a DSML; they can be complemented by constraints expressed in a declarative language like OCL (Object Constraint Language), by a concrete syntax, by an execution semantics, etc.

Recent investigations of the state of practice in industry reveal that the use of MDE is widespread and that the majority of MDE usage examples follows domain-specific modeling paradigms. Whittle and others [WHR14] report that most of the companies that succeeded to adopt MDE in their development process created or used languages specialized for their own domains; also, they confirmed that the number of DSMLs in a company tends to be large and that, sometimes, several "mini"-DSMLs are used within a single project [HWRK11]. Therefore, DSLs make part of the current practice in Systems Engineering, being used extensively and having metamodels as primary artifacts; we attest this in the next section by presenting this scenario in a large company.

1.1.2 DSLs in the wild: the Thales scenario

Thales[2] is a large company involved in different industry sectors (aerospace, space, defence and transportation areas, etc.); they produce software intensive systems, extensively using and evolving the ARCADIA method [Voi08b]. This method is a viewpoint-based architectural description, defining five different abstraction levels (further explained as phases) of a system, following the ISO/IEC 42010, Systems and Software Engineering - Architecture Description [ISO10].

Thales' engineers use numerous domain specific modeling languages to develop integrated sets of systems according to ARCADIA. These languages are built within a set of dedicated representations to analyze domain-specific problems. The language workbench provides a set of highly dynamic (changes are often) notations working seamlessly together on top of models; in addition, they can be combined and customized according to the concept of Viewpoints. Views, dedicated to a specific Viewpoint, can adapt both their display and behavior depending on the model state and on the current concern. The same information can also be simultaneously represented through diagram, table or tree editors. We enumerate following three examples of Viewpoints.

- **Performance.** It is the view in charge of analysing properties such as CPU usage, bus overload and latency.

[2] `http://www.thalesgroup.com/`

- **Fault Propagation.** It is the view in charge of analysing and act over system failures, handling the system exceptions.
- **Spatial Arrangement.** It is the view in charge of analysing the spatial constraints impacting the physical architecture.

ARCADIA also defines five main phases of Thales' systems engineering, which we briefly explain following.

- **Customer Operational Need Analysis.** This step focuses on analyzing the customer needs, goals, expected missions and activities; it ensures that the system definition is adequate to its real operational use. This phase also defines the IVVQ (Integration, Verification, Validation, Qualification) conditions.
- **System Need Analysis.** This step focuses on defining how the system can satisfy the operational need, along with its expected behavior and qualities.
- **System Logical Architecture Definition.** This step focuses on identifying the system parts, their contents, relationships and properties, excluding implementation or technical/technological concerns. In order to assure that these parts are stable and safe to the further steps, all major non-functional constraints, such as safety, security and performance, are taken into account to find the best trade-off among them.
- **System Physical Architecture Definition.** This step also focuses on defining the system architecture, but in the physical components level, making them ready to be developed in a low-level engineering.
- **Definition of Components Development & IVVQ technical ContractEPBS.** This step focuses on supporting the construction of the EPBS (End-Product Breakdown Structure), benefiting from the previous architectural definition, defining components requirements, and preparing a safe IVVQ (Integration, Verification, Validation, Qualification).

Each of these Viewpoints and phases contributes to the number of DSLs used in Thales. These languages are defined as a set of 20 metamodels with about 400 metaclasses and about 200 validation rules; they model the ARCADIA method in an eclipse-based environment. Besides, this workbench is extensible and new languages can be defined to design novel systems. Therefore, those numbers are constantly growing; the diversity of DSLs in Thales is a first and critical aspect of its development process.

1.2 Managing variability in DSLs

Given a DSL definition, whether it is expressed with metamodels, grammars or even code, we can know the different combinations of models/programs that can be made for that given domain. Therefore, a DSML expresses all possible models in the domain, and

in this set, there are different valid products that fit the needs of clients. Orthogonally to this, the main goal of Model-driven/based Software Product Lines (MSPL) [CAK+05] is to manage the commonalities and variations among the possible products, in this case, the possible models.

Managing this variability from a product line perspective allows to raise the level of abstraction (even more), gaining productivity by exploring reuse and all the other benefits that comes along with using Software Product Line Engineering (SPLE) [CN01]. However, there is a price to pay: the variability of the DSL must be captured and expressed in a model – the variability model – and it must work seamlessly with all the machinery of the given domain (e.g., language workbenches, frameworks, editors, etc).

1.2.1 Externalizing variability and standardizing it for any DSL

From a practical point of view, such a variability model could be amalgamated to the DSL itself, having the metamodel of the variability language together with the metamodel of the DSL. On the one hand, this amalgamated approach can save effort on the integration of the variability and the domain-specific language machinery. On the other hand, it can add complexity to something that should be simple – *"DSLs are often made simple and should stay simple"* [HMlPO+08]. Another drawback of an amalgamated approach is that it does not scale in real scenarios. For example, in a company that uses several DSLs and can invent and implement a new one in around two weeks [3], the cost and redundancy of such approach undermine its adoption.

In these amalgamated approaches, variability can be modelled in existing models, such as requirement or design models. Some approaches, as those proposed by Gomaa [Gom06], Gomaa and Shin [GS02], Ziadi and others [ZHJ03], propose including commonality and variability in the UML models, opposed to orthogonal modeling. However, this strategy has some limitations: the mixing of variability and other modelling concepts [HPS08, PM06] and the fact that variability is implicitly spread across the product line software artifacts.

In the opposite direction to the amalgamated approach, the *separated-languages* approach [HMlPO+08] aims at having the variability modeling language as an independent part of the whole. In this way, the constructs of a variability model simply refer to the constructs of the DSL, which makes it orthogonal to the concerns of the domain. This idea gave birth to the initiative of the Common Variability Language[4] (CVL): a domain-independent language to specify and resolve variability over any instance of any language defined based on MOF (Meta-Object Facility[5]). CVL results from a large consortium involving both industry and academia, being an emerging standard and continuously increasing its adoption.

A software product line defined using CVL has four main parts: a variability model, a base model (i.e., an instance of a DSL), a realization model (contains relationships

[3] According to [WHR14], DSLs can be used extensively and in a "quick and dirty" way, with DSLs and their generators being developed in as little as two weeks.

[4] `http://www.omgwiki.org/variability`

[5] `http://www.omg.org/mof/`

between the variability and the base models) and a derivation engine (i.e., the semantics of the relationships in the realization model and the algorithm that executes them). Having separate models for each concern favors modularization and reusability; this is a step towards externalizing variability from the domain language and standardizing it for any DSL. More details on CVL can be found in Section 2.5.

1.2.2 Fragile – please do not break

As stated in Section 1.1.2, Thales already has an established and efficient model-based method in their systems development processes, their goal is to leverage this process from single systems to family of systems, maintaining their safety and quality standards. A well-defined and -implemented model-driven development process has good properties, such as reliability, traceability and automation. These properties are explicit and can be verified in the case of Thales approach. In their process, we can state that a composition of model elements that will be part of a given system, is effectively consistent, due to their model verification techniques.

Integrating SPL into this well-defined process may lead to some complex challenges. If we integrate the variability model with their current models, we can imagine to occur that decisions made in the variability model may be inconsistent with selected elements in the models and artifacts level. The consistency of variability models and their configurations can be checked with current variability management tools (satisfiability algorithms can be used to check whether a selection of features is valid or not [FAM]). In the same way, consistency in well-defined models, such as the ones used by Thales, can also be checked [Voi08a]. One issue is: how to assure the consistency of a feature selection with respect to the semantics of the artifacts related to these features?

In other words, it is difficult to assure that constructing their systems in an SPL fashion (configuring a set of features, pressing a button and having an end-product) is at least as safe as constructing them in the traditional way. Reaching such a level of confidence is an open problem even in academia, so in industry, there is still a long way road. Therefore, leveraging product line engineering for each of these languages and domains is very expensive and error-prone; it has to be supported by automated tools.

Several stakeholders have to work during the design and development process of the tool chain for supporting SPLE in the Thales context:

- Product-line engineers who have to identify the commonalities and the variants and in charge of designing the VAM and the VRM.
- Product engineers who have to create specific products, focusing on creating valid products regarding a set of requirements.
- DSL designers who are in charge of creating or extending existing DSLs (base metamodel).

The ultimate goal of such a tool chain is to enable the generation of systems by configuring options in a variability model; CVL has been prototyped to leverage this,

however there are some open issues around its generality and safety, which we discuss in the next two sections, respectively.

1.3 One language to rule them all?

Product Derivation is a key activity in Software Product Line Engineering. During this process, the core assets (i.e., the base model) are customized and selected according to a given configuration of the variability model. Derivation engines have been designed and implemented to automate Product Derivation. These engines mainly work by removing, replacing or adding elements from the set of core assets in order to derive a concrete product; in the case of an MSPL defined with CVL, we would obtain as output of the derivation an instance model of the used DSL.

Ideally, CVL derivation should be adequate in any working domain, however, in the practice of Systems Engineering, we observe some challenges on using CVL. The derivation operators of CVL can be too generic and thus ignorant from the domain knowledge; **their semantics can be specialized for better fitting a domain.** To illustrate this, we present following the different scenarios in which the semantics of a simple derivation operator (ObjectExistence) can vary according to different factors.

1.3.1 Semantics variation scenarios

The primary semantics of the ObjectExistence variation point is to determine whether a model element exists or not in the materialized model. Considering a negative derivation, this is done by checking the resolution of a feature, if it is set to yes, nothing is done, if it is set to no, the referred model element is excluded. However, we have to consider that excluding a model element may lead to secondary operations to complement the primary semantics of the variation point. This semantics complements, or secondary operations, may vary according to different scenarios.

The first scenario to be considered is that this materialization semantics can vary within the same metamodel. In Figure 1.1 (a), we have a base model that conforms to the UML Class Diagram metamodel. This model has three classes: *Garage*, *Car* and *Sedan*. The class *Sedan* is a subclass of the class *Car*, which represents a car that can be parked in a garage. Therefore, the class *Garage* represents a place that can accommodate cars. To exemplify the semantics variation of excluding a model element, we remove each class of this model and observe the possible outcome for each class.

Although we exclude the same type of element (Class) in this example, we can see that the semantics of the exclusion operation leads to different possible secondary operations in the model. A possible result from excluding the class *Garage* is shown in Figure 1.1(b), which is to exclude the class *Garage* and all its relationships. This outcome is reasonable, considering that the other classes are not depending on the class *Garage* to exist. This same scenario can be observed in Figure 1.1(c). After removing the class *Sedan* , the inheritance relationship is removed and the other classes remain in the model. On the other hand, as illustrated in Figure 1.1(d), removing the class *Car*

leads to exclude not only itself and its relationships, but also removing its subclasses, in this case, the class *Sedan*.

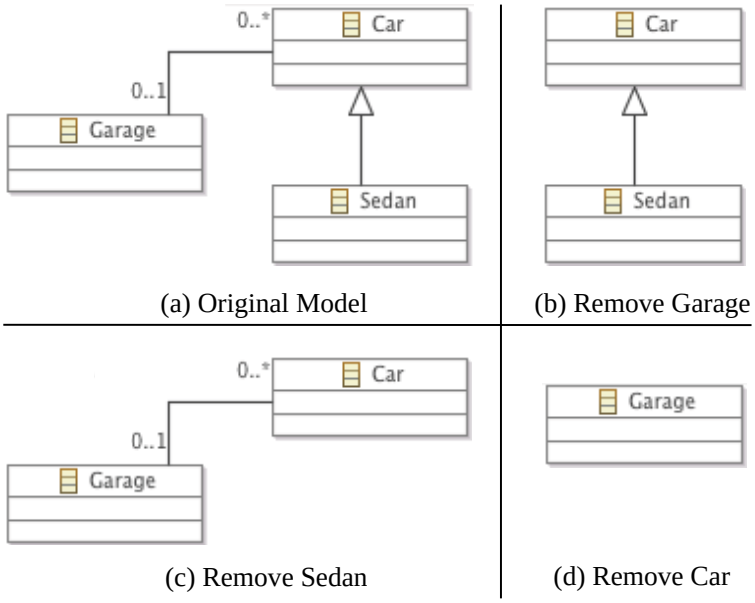


Figure 1.1: Different semantics for removing a class.

The semantics of excluding an element can vary for the same type of model elements, but can also vary for different types. Excluding a package of a class diagram can imply on removing all its classes. However, excluding a class attribute may not lead to any further operations.

Another scenario to consider is related to the other kinds of base models, such as the activity diagram of the UML, in which we can observe that the materialization semantics can vary even more. In Figure 1.2, excluding the *Fasten Seat Belt* activity can imply on four operations: remove *Fasten Seat Belt* model element, remove the income link, remove the outcome link and create a new link from the *Get in* activity to the *Start Engine* activity. This is also discussed in [CA05b].

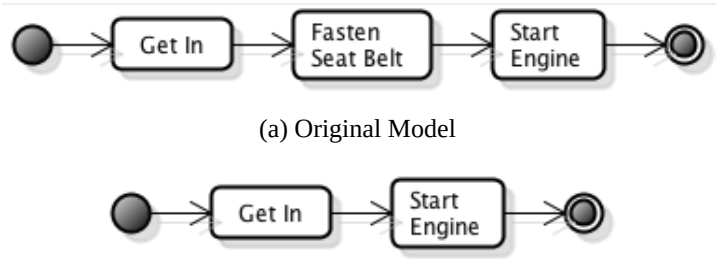


Figure 1.2: Removing an activity.

The need for customizing the operational semantics of CVL for a specific domain does not mean that CVL is inadequate; it rather confirms that CVL is extensible and can be used as basis. In fact, one can also argue that this semantics variation can be expressed with the CVL standard semantics, by composing variation points. However, we noticed from practical experience that such customization is important to leverage abstraction over the base model semantics, encapsulating secondary operations. Therefore, the engineer in charge of designing a CVL realization model can, for example, abstract over tedious operations, such as removing dangling references or excluding contained elements.

Figure 1.3 illustrates the semantics variations of derivation operators. For example, D_1 is an operator that can assume three different meanings, according to the base model element it is pointing, whether they are in the same DSL or not. This Figure gives us the big picture that one single semantics for an operator does not suit the needs and that is why **practitioners tend to customize the CVL semantics for their own DSLs.**

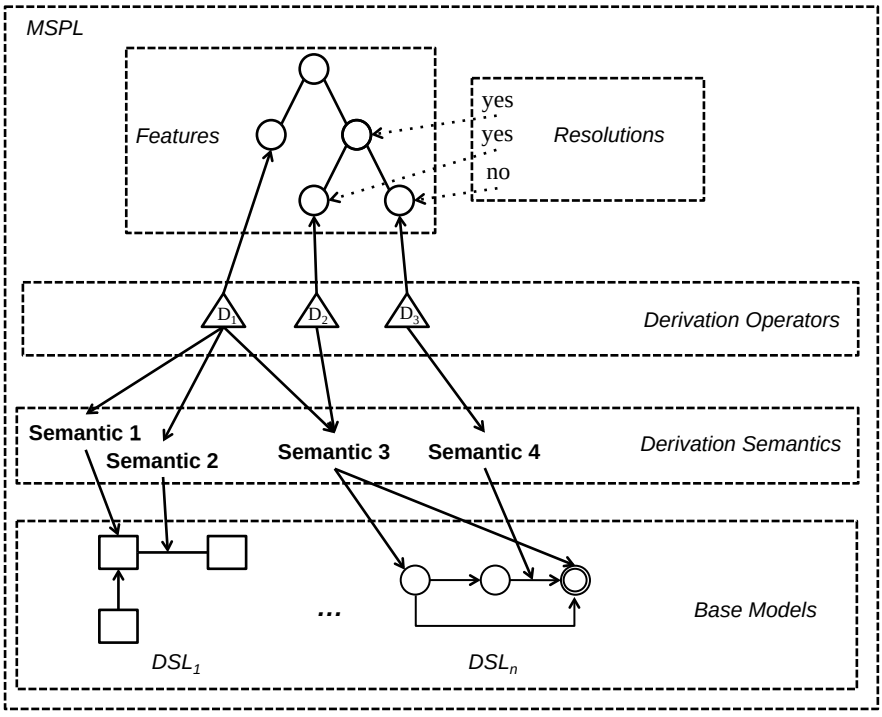


Figure 1.3: Semantics variation of derivation operators.

1.3.2 Semantics specialization mechanisms

CVL provides means to be specialized: it has a semantically customizable derivation operator called Opaque Variation Point (OVP). An OVP works like an extension point, allowing the implementation of "home-made" semantics as a black box that can define

an arbitrary behaviour to be executed during derivation.

Besides the OVP, practitioners can also introduce new derivation semantics statically inside the code of the CVL derivation engine and the second one is using the strategy pattern, but also inside the derivation semantics. We have analysed the different mechanisms used to customize the CVL semantics in 4, as well as their advantages and disadvantages.

1.4 And after all, is it safe?

The *design space* (also called domain engineering) of an MSPL defined with CVL 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[6] have to conform to numerous well-formedness and business rules expressed in the modeling language (e.g., UML exhibits 684 validation rules in its EMF implementation). The number of derived models can be infinite while only part of the models are safe and conforming to numerous well-formedness and business rules. Consequently, the engineer has to understand the intrinsic properties of the modeling language when designing an MSPL.

The two modeling spaces (variability and working domain) should be properly connected so that all valid combinations of features (configurations) lead to the derivation of a safe model. In CVL, as in many MSPL approaches, the realization layer is crucial and should be properly managed. 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 delete model elements that are dependent on others (e.g., deleting a super class of a class without deleting also the class) for a given combination of features [CP06], or perhaps to forget a constraint between features in a variability model and allow a "valid" configuration despite the derivation of an unsafe product (this is illustrated in detail in Section 2.6 of Chapter 2).

Second, the *derivation engine* executes the realization model and produces a product model that has to conform 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.

1.5 Engineering MSPL in industry needs special assistance

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

[6] 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.

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. At the tooling level, CVL leads engineers to customize its semantics to better tackle their specific domains; it is hard to know whether these modifications will only produce safe products or not.

In this thesis, we want to provide assistance for both levels: the modeling level, helping designers of CVL to make safer models; and at the tooling level, assisting engineers of derivation engines to better implement custom operators. Our approach must differentiate from others in the literature and consider the challenges previously presented in this chapter.

The majority of standard Verification & Validation (V&V) techniques for SPL relies on assumptions that do not hold in our context. **Our approach cannot assume**:

- a single formalism in which all base models are expressed, like in model checking techniques (e.g., [CHS+10b, CHSL11, AtBGF11]), as we must consider a multi-DSL scenario.
- existing variability, realization or even base models; the ideal would be to provide assistance even before starting to create the MSPL's artifacts, assuming only the DSL definition in a metamodel as input.
- that the derivation engine is correct. CVL derivation engine is not yet safe, assuring the derivation of a correct model is still a theoretical and practical problem; besides, engineers tend to customize CVL's derivation semantics to adequate it to a domain, as already discussed, such customizations may introduce many errors.

1.6 Synthesis

In Figure 1.4, we synthesize the specific challenges of leveraging variability management in a systems engineering context. They will guide the remainder of this thesis; our goal is to address the five challenges. The challenges are extracted from the Thales scenario, where multiple languages and dimensions are used to engineer a system. These languages are supported by model-driven workbenches, which are also important to leverage automation in the development process.

The five challenges can also be seen as research questions:

1. How can we provide early assistance for designing SPLs for new languages?
2. How can we customize the derivation support of CVL?
3. How can we provide separation of concerns in variability modeling for systems engineering?
4. How can we integrate variability management in systems engineering in a non-intrusive and seamless way?
5. How can we leverage consistent generation of products?

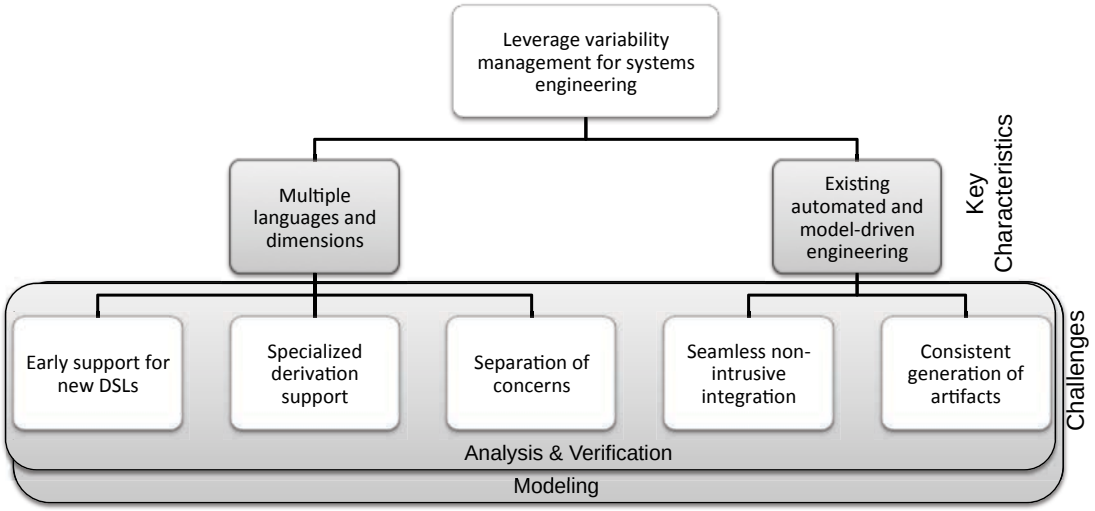


Figure 1.4: Synthesis of the challenges.

Concluding, the main objective of this thesis is to leverage assistance for variability management in systems engineering, helping stakeholders to construct their derivation mechanisms and to design better SPL models. Knowing this objective and having identified the specific challenges synthesised in Figure 1.4, we provide as contribution an automated approach to analyse and assist the creation of MSPLs for any DSL. We do not assume any other artifacts than a metamodel as input, however the approach is flexible enough to also use variability and base models. The approach works by generating examples and counterexamples of MSPLs for a given DSL, revealing potential errors even before a variability model being defined.

Chapter 2

State of the Art

In this chapter, we present the State of the Art of Variability Management and Analysis. First, we present how variability has been evolving as a core concept on software and systems development, wrapped in a trending paradigm called Software Product Lines 2.1. After, we introduce Model-driven Engineering, a strong ally to catalyse the benefits promoted by SPL (Section 2.2), also explaining the conjunct concept of Model-based Software Product Lines in Section 2.3.

Once the foundations are presented, we review the literature of variability modeling approaches (Section 2.4), assessing them according to the issues raised in the previous chapter. The literature review of variability modeling approaches supports the use of the Common Variability Language (CVL), which we then explain in Section 2.5.

We dedicate the rest of the State of the Art to the issues on realizing variability in SPL (Section 2.6) and the existing analysis techniques for this purpose (Section 2.7). Finally, we make a synthesis of the issues addressed by the approaches of the state of the art and the open challenges that we seek to address in this thesis (Section 2.8).

2.1 Software Product Lines

As in any engineering branch, constructing software is subject to constant evolution. The way we build systems has been changing over the last decades; innovative methods, tools, languages and paradigms shape the future to increasingly more efficient software design and development. A notorious evolution is concerning automation. Just like in the industrial revolution, software engineering has been progressively increasing automation in its processes, quiting handcrafting and entering industrialization.

We can think about this automation in a broader sense than just the execution of tasks by the machine. In the case of software development, automation begins with the need to reuse knowledge and code, to simply avoid unneeded repetition, which obviously promotes time and economic advantage. To achieve this reuse, once again software engineering reproduces methods from other engineering disciplines, adapting them to the software reality. It is the case of assembly lines; perfected by Henry Ford in the beginning of the twentieth century [FC22], they were responsible for leveraging

the mass production of cars.

What if we could make software and systems like we make cars? This vision leads us to think that software production can be systematized and automated – we could assemble different pre-built parts of a software product in a systematic way. *If only software artifacts were as easy to handle as car parts or Lego bricks.* One key and challenging difference is that software is far more customizable than standard goods. Instead of producing one single standardized product for every customer (mass production), industrialized software production is closer to mass customization, in which products are made to meet individual customer's needs [BSRc10].

To meet this need for customization, systems have to be efficiently extended, changed or configured for use in a particular context [SvGB05, CBA09]. 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 [AK09]. *Software Product Line (SPL)* engineering has emerged to address the problem [CN01, PBvdL05], involving both the research community and the industry.

2.1.1 Domain Engineering and Application Engineering

Constructing software and systems following a product line approach requires to think about these artifacts in a broader sense: instead of dealing only with the tasks to build a single product, engineers must also take into account the family of similar products in a domain. This last concept is the fundamental idea of *Domain Engineering*; its main motivation is to leverage the development of parts of software **for reuse** in a family of applications. These parts can be seen as *features* of the system. Many definitions of *feature* have been proposed in the SPL literature, it can be roughly defined as an end-user visible functionality of the system [CE00]. These features can be common to many products of a domain or vary among them; they are often implemented as reusable artifacts.

Complementary to the Domain Engineering, the *Application Engineering* of an SPL is the activity of developing products **with reuse**, by exploiting the reusable artifacts, composing and adapting them to the specific needs of a single product. Figure 2.1 shows an overview of the Application Engineering Activity; when this activity is automated, a derivation engine is responsible to transform the core assets into a product corresponding to a configuration.

Ideally, the functionalities in the final product can be mapped to the features of the family of applications anticipated during the Domain Engineering phase. As Domain and Application Engineering are continuous processes in an SPL, it is common to have feedback from one to the other. For example, features that appear in an individual product during its development and that were not thought before in terms of an entire domain can be promoted as reusable assets among other products. Clearly, managing all the commonalities and variabilities of a family of products is challenging; however, it can be facilitated with variability modeling (see next section 2.1.2).

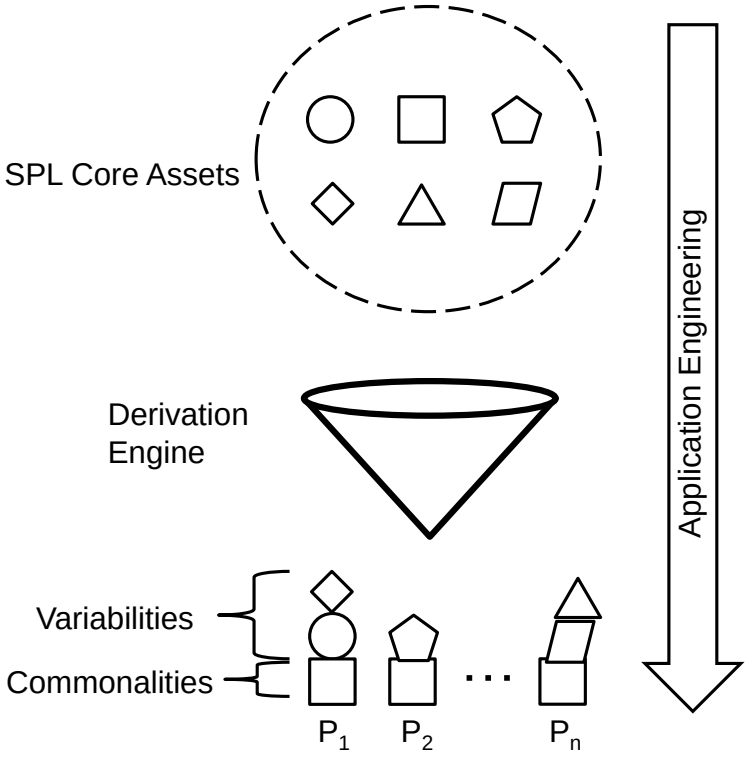


Figure 2.1: Application Engineering.

2.1.2 Variability modeling

Variability modeling is a key activity in SPL, it crosscuts both domain and application engineering. Over the past twenty years, a number of variability modelling approaches have been proposed. Roughly, they can be grouped in two main categories: Feature modeling and decision modeling.

Decision modelling focuses on decisions rather than domain description. Decisions were first introduced by Campbell and others [CFW90] as "actions" which can be taken by application engineers to resolve the variations for a product of a system in the domain. The *Synthesis* method [Bur93] drives the majority of existing decision modeling approaches. A decision model is defined as *"a set of requirements and engineering decisions that determine the variety of work products in the domain, and must be resolved by an application engineer to define and construct work products"*. Schmid and John [SJ04], Forster and others [FMP08], Dhungana and others [DRGN07], amongst others, use decision models as variability modelling language.

Feature modeling is the most popular notation and has gained attention of both research and industry. The first feature model was proposed by Kang and others [KCH+90], in 1990, as part of the method Feature-Oriented Domain Analysis (FODA). Since then, several other feature modeling approaches were proposed based on FODA [CBUE02, CHE04, FFB02, GFA98, KTS+09, KKL+98, KLD02, RFS08, SHTB07]. They usually

focus on describing the product line domain and their key idea is to capture in a feature model the set of possible products of a product line.

A feature model represents a hierarchy of characteristics of a software/system that are visible to end-users (i.e., features). Figure 2.2 shows an example of feature model: each box represents a feature, which can mandatory, optional, mutually exclusive (Xor-group) or alternative (Or-group) children – this is how variability is modeled in feature models. They can also have cross-tree constraints, expressing that one feature requires or excludes another.

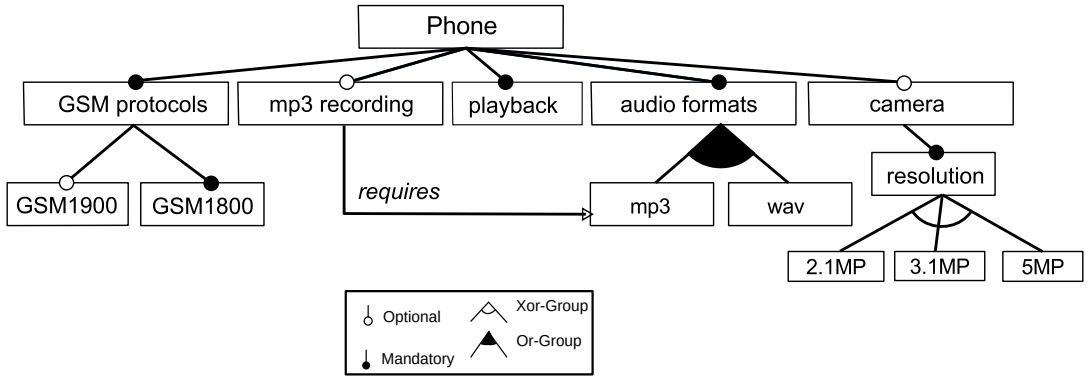


Figure 2.2: Example of a feature model adapted from [CGR+12].

Variability modeling has some important characteristics; some works try to capture them, for example, Czarnecki and others [CGR+12] have identified ten dimensions of variability modeling approaches: applications, unit of variability, orthogonality, data types, hierarchy, dependencies and constraints, mapping to artifacts, binding time and mode, modularity and tool aspects. Istoan and others [IKP11] propose four different categories of variability modeling approaches. Following, we briefly discuss some characteristics of variability models that we believe to be important to the remainder of this thesis.

Unit of variability. These are the key concepts of a variability model. These units are features in the case of Feature Modeling and decisions in the case of Decision Modeling. The unit of variability defines how grained are the variable and common parts expressed in the variability model.

Orthogonality. The goal of this dimension is to introduce explicitly documentation of the variability in software product lines into a separate model. They provide a cross-sectional view of the variability of the product line across all software development artifacts. One-way references to the base model describe how the base model elements can vary. Bachmann and others [BGdPL+03] proposed the use of orthogonal variability models to provide this separate view of the variability by documenting explicitly the variation points. The Orthogonal Variability Model (OVM) was initially proposed in [PBL05]. This approach proposes to document variability in a separate model, and interrelates variability on the base product line models. OVM is mainly characterised by considering variation points as first-class citizens. Another approach proposed to make

variability models orthogonal to the product line models is the Common Variability Language (CVL) [FHMP+11b].

Mapping to Artifacts. This is an important characteristic of variability modeling techniques that can handle product derivation. In order to reflect decisions made in the variability model into the actual artifacts, we need to map both levels. Section 2.1.3 discuss the different ways of realizing variability after a configuration and a mapping to domain artifacts.

Modularity. This is a key characteristic to enable reuse and separation of concerns in variability modeling. As variability models may express an entire product line of a complex organization, it must provide means to encapsulate and abstract parts of its information [KAO11].

2.1.3 Realization/Derivation techniques in SPL

There are different approaches to manage variation at the assets level of the SPL; they differ in the way they transform or compose the core assets to generate the product.

Annotative approaches derive concrete variants by activating or removing parts of a model or a program. Variant annotations define these parts with the help of, for example, UML stereotypes [ZJ06] or presence conditions [CHS+10b, CP06, CA05a]. The directives of the C preprocessor (#if, #else, #elif, etc.) conditionally include parts of files and can be used to activate or deactivate a portion of code [KAK08, LAL+10].

Compositional approaches associate reusable fragments (e.g., feature modules or model fragments) with features that are then composed for a particular configuration. For instance, Perrouin *et al.* offer means to automatically compose modeling assets based on a selection of desired features [PKGJ08]. Superimposition is a generic composition mechanism to produce new variants, being programs (written in C, C++, C#, Haskell, Java, etc.), HTML pages, Makefiles, or UML models [AJTK09, TBKC07]. Software artefacts are composed through the merging of their corresponding substructures [AKL13].

Dhungana *et al.* provide support to semi-automatically merge model fragments into complete product line models [DGRN10]. Annotative and compositional approaches have both pros and cons. Voelter and Groher *et al.* illustrated how negative (i.e., annotative) and *positive* (i.e., compositional) variability [VG07] can be combined. *Delta modeling* [SBDT10, CHS10a] promotes a modular approach to develop software product lines. The deltas are defined in separate models and a core model is transformed to a new variant by applying a set of deltas. At the foundation level, the *Choice Calculus* [EW11, Wal13] provides a theoretical framework for representing variations (being annotative or compositional).

The Common Variability Language (CVL) has emerged to provide a solution for managing variability in any domain-specific modeling languages [SZLT+10, HMlPO+08]. CVL provides both the means to support annotative, compositional, or transformational mechanisms. CVL thus shares similarities with other variability approaches.

2.2 Model-driven Engineering

MDE is a software development paradigm originated at the beginning of the 2000s, with its first concepts designed and presented in the Model-driven Architecture approach, an OMG standard [S+00]. MDE advocates the use of models as the main artifacts in the software and systems development. Models help engineers to abstract from unneeded details by allowing to represent structural or behavioural problems in a simpler way [Sch06].

In MDE, each kind of model can be specific to represent problems of a distinct domain of knowledge. Instead of having one single kind of model to serve as unified language to represent any problem, Domain Specific Modeling Languages (DSML) are dedicated to areas of expertise, such as medical or avionics systems. Their specificity can also be based on the kind of information that will be modeled, for example, to represent data flows, security, requirements or architectural components.

As with any language, DSMLs have two main components: syntax and semantics. The syntax of a DSML can be divided into the abstract and the concrete syntax. The abstract syntax of a DSML defines its concepts and the relationships between them, while the concrete syntax maps these concepts to visual elements used in models. The semantic of a DSML is the actual meaning of the syntax representations. From the two types of syntax and the semantics, the most important component of a DSML is its abstract syntax; it is common to find DSMLs without formal semantics definition or without a concrete representation, but the abstract syntax is imperative.

A metamodel is the model that defines the abstract syntax of a DSML and, therefore, it is the central artifact of a DSML definition. As the metamodel is also a model, it is also expressed in a third-level language like MOF, E-MOF, Ecore, etc; these third-level languages are bootstrapped, defining themselves their abstract syntax. Figure 2.3 shows a metamodel of a finite-state machine DSML, expressed using Ecore. This metamodel has three concepts (named EClass in Ecore): FSM, representing the finite-state machine itself; *State*; and $Transition$. It also has the relationships between the concepts, defining compositions or associations.

The essential and most important function of the metamodel of Figure 2.3 is that it can describe what is and what is not a finite-state machine. The concepts and relationships serve to define well-formedness rules. For example, the arrow from FSM to State, labeled *initialState*, with the number 1 in its end, determines that a finite-state machine must have exactly one initial state; in the same way, a Transition must have exactly one source State and one target State.

Therefore, if a model M does not violate any of the well-formedness rules of its corresponding metamodel, we say that M **conforms to** its metamodel. In Figure 2.4, we illustrate the design space of finite-state machines, i.e., the possible ways of designing an FSM model, being conforming or not to its metamodel. The models that do not violate any well-formedness rule, expressed in the FSM metamodel, are conforming models, belonging to a correctness envelop (the subset of all valid instaces of a metamodel and that also conform to all the well-formedness rules).

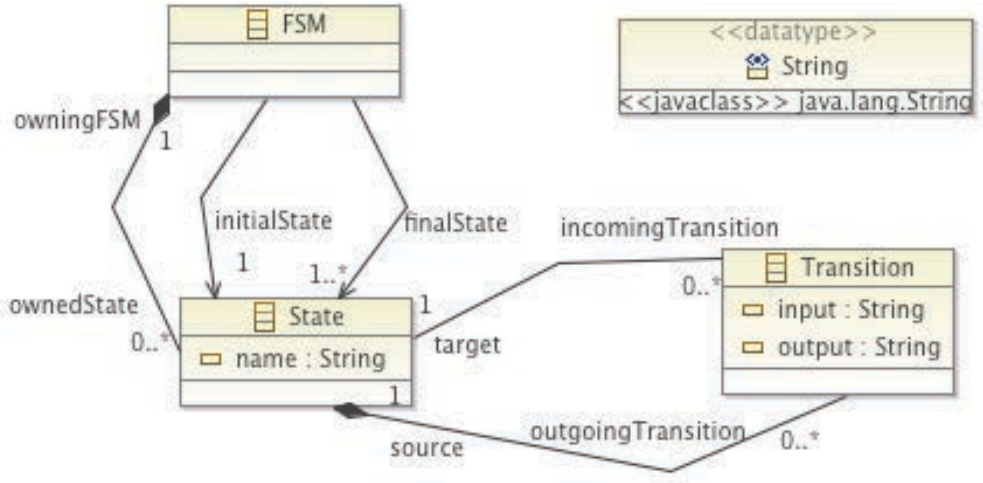


Figure 2.3: FSM metamodel.

2.3 Model-based Software Product Lines

Initially, it is hard to imagine completely automated product lines of computer programs; the fine-grained complexity of their language elements, their instructions and their control flow make the design of a safe SPL close to impossible – but at least we have models to abstract the program complexity. Therefore, one possibility to ease this complexity is to ally SPL with Model-driven Engineering (MDE), and then having *Model-based SPLs* (MSPLs). MSPLs have the same characteristics and objectives of an SPL, except that they extensively rely on models. Models, as high-level specifications of systems, are traditionally employed to automate the generation of products as well as their verifications [Sch06]. They are simpler artifacts than standard programs; their concepts and relationships are less diverse and easier to handle than the ones in a general programming language, for example.

A variety of models may be used for different development activities and artifacts 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 [Voi08a].

Numerous MSPL techniques have been proposed (e.g., see [PBvdL05, PKGJ08, HSS+10, CA05a, CHS+10b, CP06, ZJ06, VG07]). 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) [Gro07]), 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 key point of an MSPL is the ability to check whether the derived product is

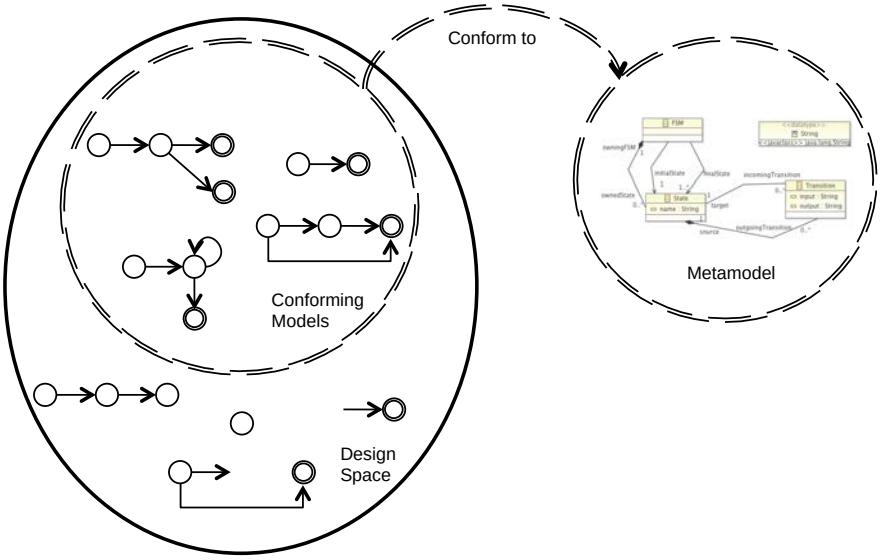


Figure 2.4: Design space and conforming models.

correct or incorrect. The ideal would be that an MSPL does not generate wrong model, however it is very hard to assure that an MSPL will only generate correct products (we discuss this issue in Section 2.6).

2.4 Reviewing the literature of variability modeling languages

Variability modeling is a trending topic on Software Engineering, numerous languages and approaches have been proposed during the last decades. At the beginning of the PhD, we conducted a systematic review of the literature in a quest for a suitable language/approach to address the variability modeling challenges of Thales, exposed in the Context chapter. The main goal of the review is to support the decision of whether constructing a new variability language/tool or simply using an existing one and suiting it to the Thales' needs. Although Thales was endorsing the Common Variability Language consortium as a submitter, they were constantly investigating alternative approaches for variability modeling.

2.4.1 Literature Review Process

The first definition that should be taken into account when carrying out a literature review is its goal [KC07]. Basically, we wanted to be able to answer five questions. The first question is the most general:

- Q1. What are the existing Variability Modeling approaches?

After answering this most general question, we will have a set of variability modeling approaches and then we can address four more specific questions:

- Q2. What kind of derivation mechanisms are supported by these approaches?
- Q3. How these approaches address the multi-level issue?
- Q4. Which and how approaches are related to the MDE paradigm?
- Q5. Which approaches provide tool support?

We motivate Q2 and Q4 with the fact that Thales already works with code and model generation, so they are always seeking to automate their development process; therefore, using a language that could support at maximum the product derivation and the MDE paradigm is essential. The motivation of Q3 is based on the complexity of the Thales development process: many stakeholders, at different levels and using different languages, need to work independently developing the product line – the approach must be able to be used in many levels with many DSLs and stakeholders. Q5 is important to identify the approaches that have already implementations, academic or commercial.

Once we set up the goals, we divide the Literature Review Process into three main steps. The first step is the definition of a search strategy. This definition is important to systematically acquire the papers related to the research question Q1.

The second step is the clustering and the selection of studies. This selection is made in order to discard irrelevant papers according to all the research questions.

The last step consists on performing a quality assessment over the selected studies, retrieving useful data and presenting it in a synthesised way. With this quality assessment we are able to answer the specific questions Q2, Q3, Q4 and Q5. These steps are presented in the next three subsections.

2.4.2 Search Strategy

We considered three different strategies to collect the papers in order to answer the research question Q1.

2.4.2.1 Existing Literature Reviews

The first strategy is to identify the existing literature reviews that focused on gathering variability modeling approaches; they gather important results from the SPL community and, as peer-reviewed papers, they are reliable sources of relevant articles.

Chen *et al.* [CAA09] present a systematic review of variability management approaches in the software product line context. The work is a collection of general approaches selected based on two main criteria: if it introduces an approach to dealing with some aspect of Variability Modeling (we will abbreviate as VM) in SPLE or if it reports an evaluation of a VM approach. The paper also analyses the kind of issue addressed by each approach. This review was performed in the year of 2007.

After the first review, in 2011, Chen and Ali Babar [CA11a] conducted another systematic review, but now addressing how the variability management approaches in

SPLE have been evaluated, and what is the quality of the reported evaluations. Still in 2011, Chen and Ali Babar [CA11b] also investigated the contemporary industrial challenges using the approaches already captured in their previous work.

On the other hand, Benavides *et al.* [BSRc10] conducted a literature review in order to identify papers regarding automated analysis of feature models 20 years after their creation. They selected and studied 53 papers in order to identify any automated operation done over feature diagrams. As a result of the study, they come up with a framework to understand the different proposals as well as categorise future contributions in the automated analysis field.

Rabiser *et al.*[RGD10] also presented a systematic review in the SPLE area, however they focused on the product derivation aspect. The review was motivated by the lack of requirements definition for product derivation support. Therefore, they researched which approaches exist in SPLE that support product derivation, identifying the features of each approach that enables it to support product derivation.

Hubaux *et al.*[HC10] have initiated a systematic review in order to identify the usage of feature diagrams in practice. Their study is still a pilot for a possible full systematic review and is still biased for few conferences. Otherwise, they found 29 papers that fit into their search string and they could extract preliminary evidences that few reports about feature usage have been performed

Djebbi *et al.* [DSF07] present an industry survey of product line management tool. The tools identification in their work was not done systematically. However, they found 12 tools for product line management, but only evaluated 4 tools. They used 13 requirements which they categorized in three different kinds: Product Line Engineering criteria, Management criteria and Technical criteria. They analyse if the requirements were matched, assigning a mark to each tool with respect to the given requirement.

Eichelberger *et al.* [ES13] performed a systematic analysis of textual variability modeling languages. They chose ten different languages and evaluate them according to their configurable elements, constraint support, configuration support, scalability support and other additional characteristics.

Berger *et al.* [BRN+13] investigated the actual use of variability modeling techniques in the industrial practice. They concluded that industry uses different notations and tools, which emphasizes the heterogeneity of the SPL industrial practice. Their survey is important in order to identify the approaches that are sufficiently mature to be prototyped or even consolidated in real scenarios.

All the papers before mentioned contributed to our knowledge of variability modeling and analysis approaches. They cover a great amount of the state of the art and can serve as basis for more sophisticated reviews, as the one we are performing in this chapter by adding new research questions adjusted to our needs.

2.4.2.2 Selection Criteria and Methods

Articles of interest for this literature review have been published in software engineering conferences. Available and properly referenced technical reports about variability modelling techniques were also valid candidates. We extracted the papers from electronic

data sources including the ACM Digital Library, the IEEE Xplore Digital Library, the RefDoc Service, the HAL open access archive, the IEEE Computer Society Digital Library, the CiteSeerX Digital Library, the IBM Technical Journals, and the book series of Lectures Notes in Computer Science available on SpringerLink.

These data sources were crawled using web-based services to retrieve the references of the articles to include in the selection. We use mainly the Google Scholar service but also to some extent the Researcher, the SciVerse ScienceDirect, the Oxford Journals, the ArnetMiner and CiteULike websites, and several authors or research team projects homepages. Because we reused results from the existing literature reviews exposed in section 2.4.2.1, we avoided making the same search and rather focus on papers published after 2008 (Chen *et al.* [CA11a] provide a large set of approaches in their systematic review, but all of them were published before 2008). We perform our selection of articles using the following string:

(variability (model OR management) OR feature model* OR software product line OR product derivation) AND (year >2008)*

The articles that are selected for this review are only those which discuss variability modeling or management. Relevant articles are identified first by their title and second by reading the abstract. If the articles do not discuss variability modelling techniques, the article is not included in the selection. Once a paper has been selected for review, we proceed the read to capture valuable information. The adequacy of articles, regarding the selection criteria, is then checked.

2.4.3 Study Selection

In this subsection, we present the application of the process to select the studies for being part of the Data Extraction step.

Among the articles that match the research criteria, some of them do not propose any variability modelling technique or approach, but deal instead with industrial requirements for variability modelling. The articles that were not proposing any variability modelling technique or approach explicitly were removed from the set of selected articles for this study. We also excluded papers that do not propose new approaches for modeling variability and that focus only on the analysis of SPLE – for this specific purpose, Section 2.7 presents the different techniques and approaches to analyse SPLs.

Considering the captured papers, we list bellow the tools studied during this work and their respective URLs:

- CaptainFeature, `http://sourceforge.net/projects/captainfeature/`
- Pure::Variants, `http://www.pure-systems.com/pure_variants`
- FeatureIDE, `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`
- RequiLine, `http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline/index.php`
- XFeature, `http://www.pnp-software.com/XFeature/`

- Feature Modeling Tool, http://giro.infor.uva.es/index.html
- Feature Model DSL,http://featuremodeldsl.codeplex.com/releases/view/20407
- CVM,http://www.cvm-framework.org/index.html
- BigLever Gears SPLE Tool, http://www.biglever.com/solution/product.html
- Feature Modeling Plug-in (FMP), http://gsd.uwaterloo.ca/projects/fmp-plugin/
- FORM CASE Tool, http://selab.postech.ac.kr/form/
- ToolDAy - Tool for Domain Analysis, http://www.rise.com.br/english/products_toolday.php
- CmapTools Knowledge Modeling Kit, http://cmap.ihmc.us/
- FAMILIAR, https://nyx.unice.fr/projects/familiar/
- Comper, https://github.com/multi-perspectives/cluster/
- Invar, invar.lero.ie/invarsite/

2.4.4 Data Extraction

We have conducted a study over the selected papers in order to extract data to answer the questions Q2, Q3,Q4 and Q5. Initially, we classified if the given approach answers or not the corresponding research question, considering also when the approach just partially handles the given issue. Table 2.4.4 shows the result for each question.

For Q2, we can refer to [DSF07, DD11] for a more detailed investigation of tools for SPLs. Similar to [DD11], we can point as possible answers if the given approach has tooling support used in academia (A), industry (I) or in both (B).

For Q3, we are interested in how the MDE assets are related to the VM approach. Each model asset conforms or not to a metamodel with its own semantics and its own set of constraints. The support of a consistent derivation semantics depends on the model-driven assets semantics. This is an important challenge in order to adequate the SPL in a model-based development process. We mark Y if the approach has means to relate to modeling assets and N if it does not.

For Q4, possible answers can be regarding to what are the product derivation activities supported by the VM approach. Product derivation activities are defined in [ROR11], and can be divided into a preparation activity (A), a configuration activity (B) and an additional development and testing activity (C). Besides following these derivation activities, managing variability modelling in a model-based context also leads to integrate an extension mechanism to refine the derivation semantics, depending on the asset's metamodel.

For Q5, we consider as premise to deal with multi-dimensions the fact that the approach is able (M) or not (S) to manage multiple variability models, or even, how the modularization of these models is tackled. This question shows that it is important to

Table 2.1: Comparison of variability modeling approaches

Approach/ Tool	Tool Support	MDE Assets	Derivation	Dimensions
Pure	I	Y	A,B and C	M
Gears	I	Y	A,B and C	S
FeatureIDE	A	N	A and B	S
FMP	A	N	A and B	S
FMT	A	Y	A and B	S
VELVET	-	N	-	M
RequiLine	B	N	A and B	S
XFeature	B	Y	A	S
CVMTool	B	N	A and B	M
CVL	B	Y	A,B and C	M
FAMILIAR	A	N	A	M
FORM	A	N	A and B	S
COVAMOF	B	N	A and B	M
Comper	A	Y	A	M
Invar	A	N	A	M
Captain	A	N	A	S
ToolDAy	B	N	A, B and C	S

study how we ensure the consistency of the variability model when it crosscuts several phases of the system development process. It is necessary to notice that this capability does not ensure that the approach consistently manages variability across all dimensions, but it gives initial modularization.

Remarks on the Data Extraction

We can conclude after planning and applying the review process that it is not a trivial task. Most of its complexity is due to the fact that variability modeling is still a recent research area. Therefore, by the time the review was applied, standardization was still missing to define variability techniques contributions and what they are able to express or compute. It is a hard task to identify if an approach can manage, for example, the product derivation, since it is a complex process and encompasses several steps. Another issue is how to categorize if an approach follows or not the model-driven paradigm. The multi-leveling characteristic is also not trivial to be recognized, mainly due to the fact that one could claim that several dimensions could be expressed in only one feature diagram.

Thus, from this preliminary discussion towards the suitability of existing approaches, the main challenge is probably not to define a new VM language, but to extend or facilitate the adoption of one. Existing VM approaches target each challenge: (i) provide

a language to capture commonalities and variation points (Q1), (ii) provide tool support for these languages and include a support to automatically derive a concrete product (Q2,Q4), (iii) use these languages at different phases of the software development process (Q5), and support model-driven assets (Q3). We highlight that CVL matches all the criteria of our research questions, which strongly encouraged its adoption. We present CVL and its main concepts in Section 2.5.

2.5 The Common Variability Language

In this section, we present the main concepts of CVL and introduce some formal definitions that are useful for the remainder of this thesis. CVL is a domain-independent language for specifying and resolving variability over any instance of any MOF[1]-compliant metamodel. The current revised submission document can be accessed at CVL's webpage: `http://www.omgwiki.org/variability`

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) resolutions of the chosen features triggers modifications in the base models to derive the final product model.

Figure 2.5 presents the overall modeling structure of an MSPL defined using CVL: a variability abstraction model (VAM), expressing the variability units (VSpecs) and their relationships; a variability realization model (VRM), containing the mapping relations between the VAM and the artifacts; the resolutions (i.e, configurations) for the VAM; and the base models conforming to a DSL.

- **Variability Abstraction Model (VAM)** expresses the variability in terms of a tree-based structure. Inspired by feature and decision modeling approaches [CGR+12], 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 thesis, we will focus on 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 [BSRc10] – as current implementations of attributed feature models are still being investigated to assist on the design of CVL models. Another reason is because our work focus on the derivation process of the SPL, rather than its configuration process. These *Choices* can be decided to yes or no (through *ChoiceResolution*) during 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

[1]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.

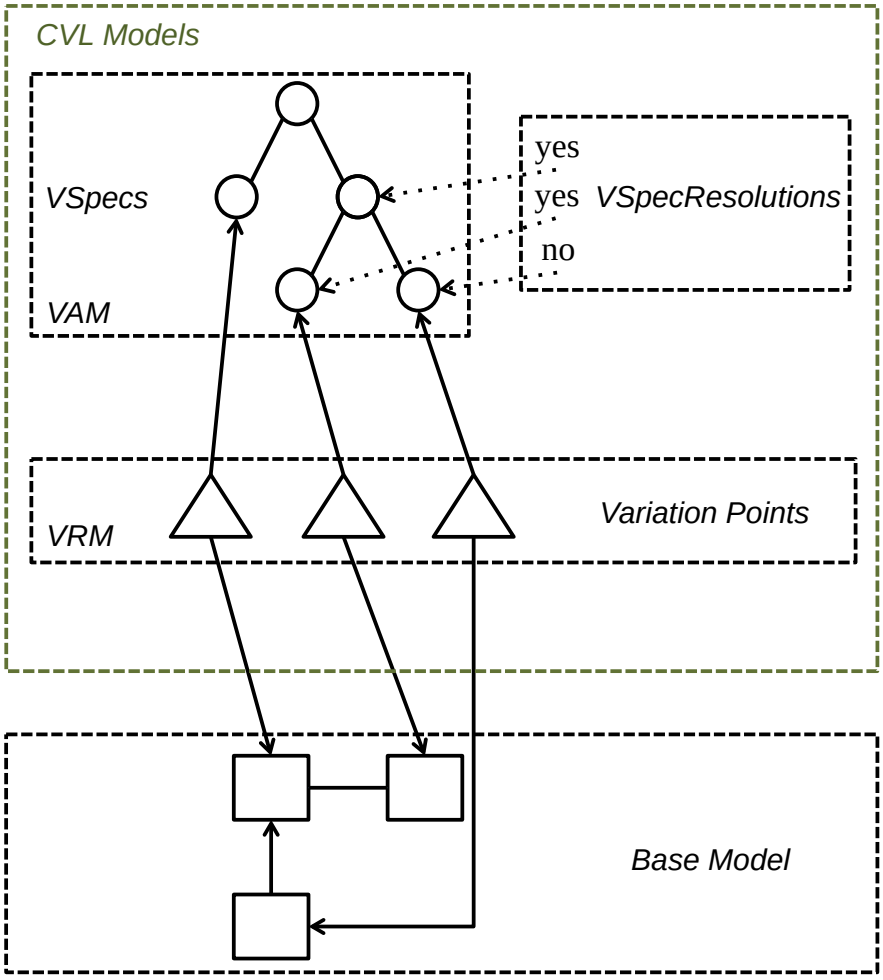


Figure 2.5: Overview of CVL models and a base model.

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 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. But in the last iteration we could identify discrepancies. With respect to the variability model, we have found evidences that it is a tough task to design it without leading to any wrong product models. It is also unfeasible to predict every possible configuration,

once this number can reach exponential.

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 = $\langle CVL, BMS, \delta \rangle$ is defined as follows:*

- *A $CVL = \langle VAM, VRM \rangle$ model is a couple 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 BMS[2];*
- *$BMS = \{BM_1, BM_2, \ldots, BM_n\}$ is a set of models, each conforming to a modeling language;*
- *$\delta : CVL \times c \times BMS \rightarrow DM$ is a function that produces a derived model DM from a CVL model, a set of base models and a configuration [3] $c \in \mathcal{C}_{VAM}$. This function represents the derivation engine.*

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

2.6 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 2.3, the FSM metamodel has three classes: *State*, *Transition*, and *FSM*. The metamodel defines some rules and constraints: a finite state machine has necessarily one initial state and a final state; a transition is necessarily associated to a state, etc. Some other rules may be expressed with OCL constraints (they are not in Figure 2.3 for conciseness), for example, to specify that there are no *State*s with the same name.

Using CVL and the metamodel of Figure 2.3, we can define a family of finite state machines. As shown in Figure 2.6, 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 2.3. The schematic representation of Figure 2.6 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.

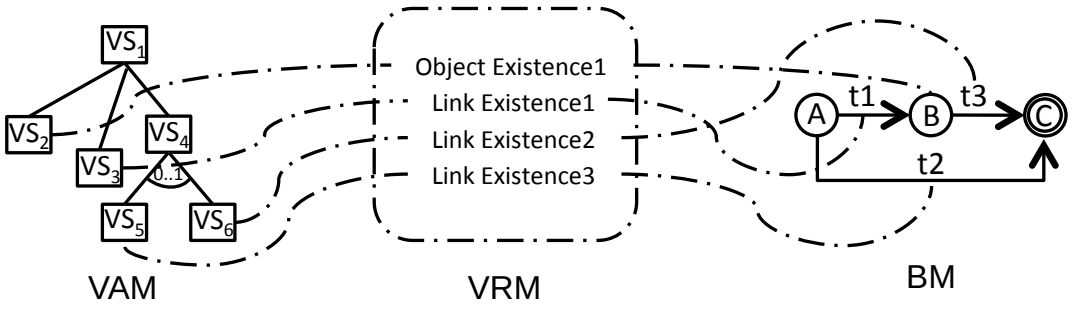


Figure 2.6: CVL model over an FSM base model.

Considering the MSPL of Figure 2.6, 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 2.7. Configuration 1 generates a correct FSM model, 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 non-conforming 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.

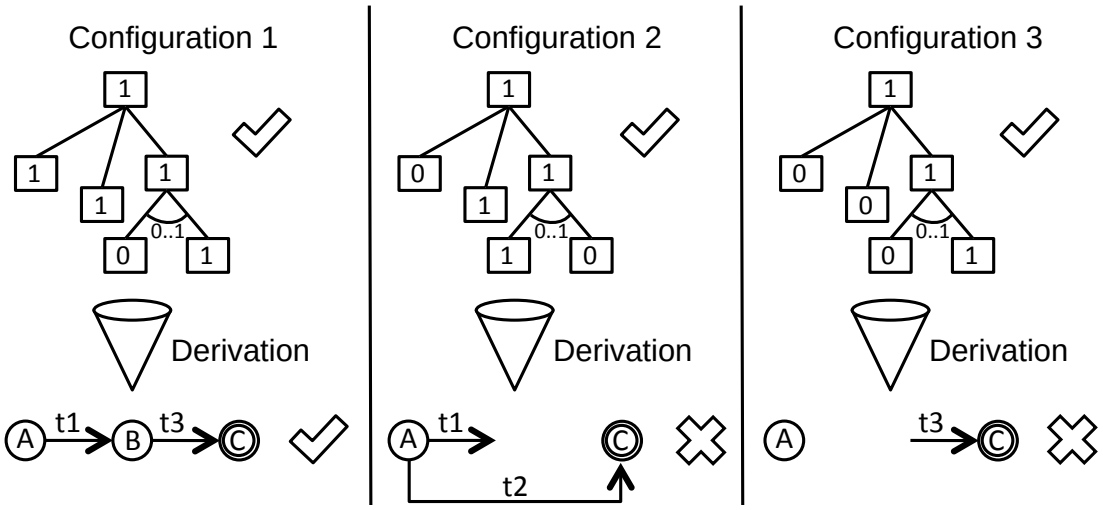


Figure 2.7: Configuration and derivation of FSMs.

The problem of safely configuring a feature or a decision model is now well understood [BSRc10]. Moreover, several techniques exist for checking the conformance of a model for a given modeling language. However, the connection of both parts (the VAM and the set of base models) and the management of the realization layer are still crucial issues [TBKC07, AtBGF11, CHS+10b, SHMP11, TAK+14a]; we present some analysis techniques and their characteristics in Section 2.7.

2.7 Analysis of Software Product Lines

Automated product derivation is the Holy Grail of SPL, having the means to automatically generate a product after a selection of features can save time and minimize costs for the company. However, achieving such a level of automation and most importantly, in a reliable way, is still a big challenge. We discuss in this section the different approaches to analyse SPLs, specifically those related to the analysis of issues in the variability realization (analysis only at the level of variability models are not considered). We rely on recent extensive classification and survey efforts about product line analysis strategies [TAK+14a, TAK+12, MTS+14].

2.7.1 Classification of SPL analysis approaches

SPLE is evolving and, as the number of analysis approaches increases, meta studies come up to strengthen the field. In [TAK+14a], Thüm and others propose a classification to these analysis; categorizing them allows to better know their capabilities and weaknesses, thus facilitating the retrieval and use of approaches for a specific need (e.g., choosing a model checking approach to analyse a small SPL but that needs a high degree of confidence). The approaches are driven by existing techniques to analyse software,

which we enumerate in the following; they are incorporated for the reality of multiple products and variability management.

1. **Type Checking**. This method of analysis consists on verifying whether a program is well- or ill-typed with respect to a type system [Pie02], i.e., a formal specification defining rules that contained systems must follow. Type checking is the foundation of the concept of conformance in model-driven engineering; a model is well-formed if its syntactic structure does not violate any rule expressed in its metamodel [AK03] (i.e., its type system).

   Advantage: Can be fully automated and scalable.

   Limitation: Rules in type systems are constrained by their decidability/checkability.

2. **Static Analysis**. This type of analyses works at compile-time by extracting semantic information and approximating the behaviours of a program [Lan92]. An example of application of static analysis is to check *live-variables* (i.e., whether a variable is accessed during the execution of its statement or not).

   Advantage: Can be fully automated and often does not need user input.

   Limitation: Some analysis techniques can be undecidable or uncomputable and have to work with approximation.

3. **Model Checking.** It is an analysis technique that verifies if a formal model, which represents a system, satisfies its specification [CGP99]. Model checkers rely on specific languages to make their computation; these languages are usually based on finite state machines and offer a precise abstraction of the systems behaviour.

   Advantage: Can be exhaustive and find errors that other analysis techniques cannot, because of its precise representation of the system.

   Limitation: Due to the state space explosion, model checking does not easily scale for large systems with many interacting parts. Another limitation is that the system must be encoded in a formal language, as well as its specification.

4. **Theorem Proving.** An automated theorem prover is a program that, given a logic formula, evaluates whether it is universally valid or not, according to an automatic deduction with the application of inference rules [Sch01].

   Advantage: Has a high precision and often generalizes over technologies.

   Limitation: It also requires the system to be encoded in a formal specification and does not scale for large programs.

In SPLE, besides the before mentioned techniques, an important aspect of the analysis approaches is in which part of the product line they focus. The most common types of analysis are at the following levels:

1. **Product-based analysis.** In this kind of analysis, the products of the SPL are generated and then analysed one by one.

   Advantage: An advantage of this technique is that it is easier to apply any software analysis technique to the products, as they are individual programs.

   Limitation: The main issue with this kind of analysis is the fact that the number of possible products in an SPL is exponential. Thus, product-based analysis can either rely on optimizations to reach a reasonable coverage of the problem space or just be exhaustive.

2. **Family-based analysis.** It operates only over domain artifacts and it considers the knowledge about valid combinations of features, which can be expressed in the variability model.

   Advantage: It is not necessary to generate and analyse all the products of the SPL; its complexity is generally reduced to a satisfiability problem.

   Limitation: The standard techniques for software analyses must be adapted to take into account the knowledge about variability.

3. **Feature-based analysis.** This type of analysis takes into account the features in an isolated way, without considering their valid combinations, as done in the family-based; it also operates only on domain artifacts.

   Advantage: As in the family-based, it is not necessary to generate and analyse all the products of the SPL

   Limitation: The main limitation of feature-based analysis is their primary assumption that features can be analysed in a modular way; it is still hard to imagine a legacy and complex system being divided in composable and completely independent parts.

These approaches can be combined among them; Thüm and others [TAK+14a] explain in details the possible combinations and the advantages and disadvantages of each combined approach. The combinations are: Feature-Product-based Type Checking, Feature-Product-based Model Checking, Feature-Product-based Theorem Proving; Feature-Family-based Type Checking, Feature-Family-based Theorem Proving; Family-Product-based and Feature-Family-Product-based.

We can conclude that there are many ways of increasing or just assess the safety of an SPL design; all of them have advantages and disadvantages, their suitability depends on the domain of the SPL and the desired properties for the checking mechanisms.

2.8 Synthesis

In Figure 2.8, we present the issues raised in the context chapter (see Section 1.6) and we relate them to techniques/features of existing variability modeling and analysis approaches that try to address the issue. The top boxes represent characteristics from

current variability modeling approaches, while the bottom ones represent the types of product-line analysis techniques. The arrow from one box to another means that the feature/characteristic is supposed to support the pointed issue. The color of the issue is how we see the level of existing support: very low (dark red), low (soft red), advanced (soft blue), very advanced (dark blue). Following, we explain how each challenge has been addressed in the state of the art (in case it has) and which features of existing approaches are related to them and how.

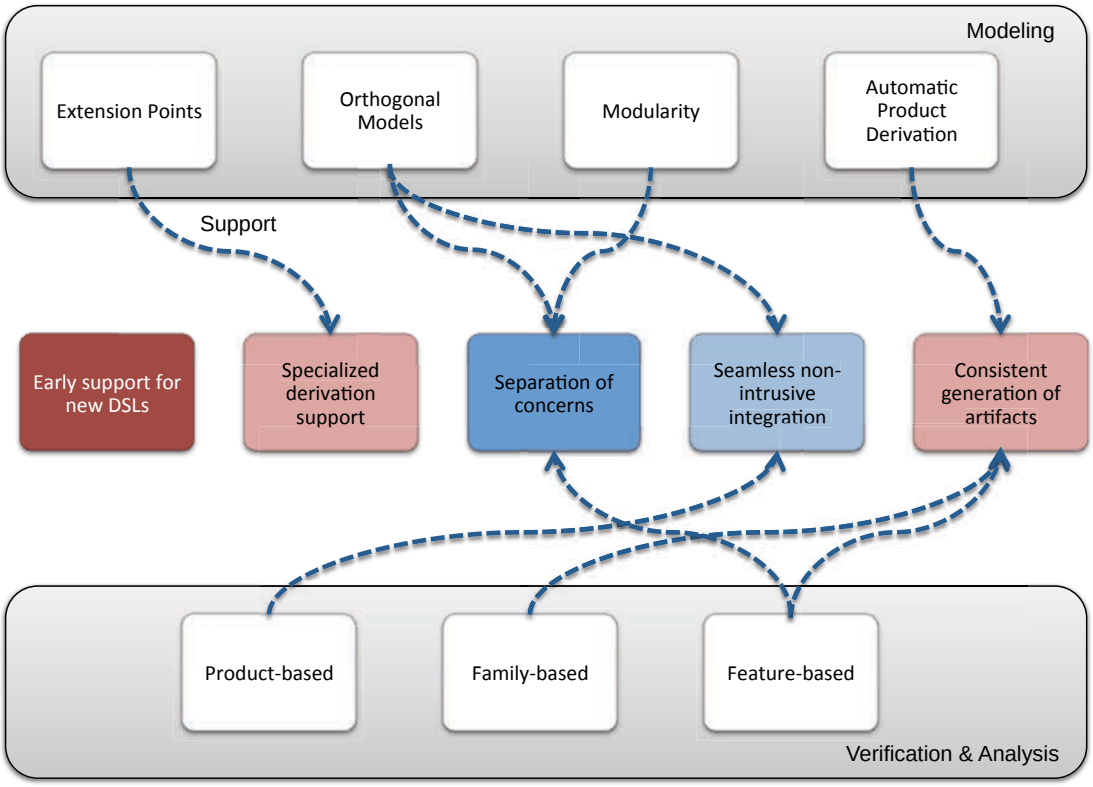


Figure 2.8: Synthesis of issues and efforts in the state of the art.

Separation of concerns

At the modeling and analysis levels, modularization mechanisms are the main features to leverage separation of concerns in variability management approaches. Some mechanisms rely on aspect-orientation [NK08, MFB+08], others in feature-orientation [KTS+09, MO04]; and CVL has configurable units and variability interfaces, which facilitate the specification of configurable components. Besides modularization, the orthogonal property of approaches like CVL and OVM plays an important role to separate variability concerns from the working domain. Although we believe that support for addressing this challenge is very advanced, crucial issues, like granularity and feature interactions [KAO11], still need special attention.

Seamless non-intrusive integration

As in separation of concerns, orthogonality of variability models is an efficient non-intrusive technique, opposed to amalgamated approaches [RFS08]. From the analysis perspective, product-based approaches can be easily integrated to existing product lines, as they can work directly in the notation/language of the product itself. Methodological and organizational challenges are important and current subjects of investigation in order to facilitate variability management in a seamless and non-intrusive way. Unfortunately, reengineering is still the most common practice to move from single systems to SPL [LC13].

Consistent generation of artifacts

This is one of the most challenging and also studied issue in the SPLE community; many approaches explore automated product derivation and program analysis techniques to ensure consistent generation of models or programs. In the case of model-based approaches, type checking techniques are used to constrain the set of possible valid products and try to ensure well-formedness of derived models [BS12, ZMP12]. Some techniques specifically address the problem of verifying SPL or MSPL[ARW+13]. 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 [TBKC07], 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 [CP06]. In [CHS+10b, CHSL11], 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 [AtBGF11].

In [ALHM+11], 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.* [ZMP12] 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 [SHMP11].

Early support for new DSLs

We consider early support for new DSLs as the capacity of providing ways to analyse the domain even before the domain engineering phase – before domain engineering, there is no variability model, configurations or even base models. The motivation is when one wants to build an SPL using only a language syntax definition as input. To the best of our knowledge, all approaches cited in 2.8 and in [TAK+14b] consider existing variability models, configurations or base models, therefore we could not find approaches providing early support for new DSLs.

Specialized derivation support

In a context with several languages and stakeholders, we cannot assume that derivation operators will have the same operational semantics in every situation [FBLNJ12]. Some variability modeling approaches propose extension points to add customized semantics or extend the current one of the product derivation. For example, CVL has opaque variation points, which work as black boxes components that can be plugged to the language. In delta-oriented approaches, one can create custom derivation deltas to specific languages [SBDT10, CHS10a]. Yet, the majority of product line approaches assume a unique/unchanged semantics in their derivation engines.

2.9 Conclusions

We presented the foundations of this thesis and the state of the art of variability modeling and analysis. From the numerous approach, we chose to present CVL as our working variability language, justified by a comparison to the other approaches of the literature review and its standardization effort. We have shown the critical issue of realizing variability in a model-based context and the numerous approaches that try to analyse SPL and address this challenge.

Observing the state of the art, we could assess if and how our context challenges were addressed. Our conclusions are:

1. There are no approaches providing support for MSPL engineering that do not consider domain engineering artifacts or existing products as input. Therefore, early support for leveraging SPL in new DSLs is absent.

2. Few approaches provide means to customize their derivation support mechanisms, the existing ones use extension points.

3. Many approaches address separation of concerns with modularization and orthogonality mechanisms.

4. Product-based analysis and orthogonal modeling are non-intrusive techniques to integrate variability modeling in existing systems development life cycle. Several approaches use both techniques and we can consider that one can analyse products without interfering in their development process; however they lack efficiency.

5. Many approaches progressed on the consistent generation of artifacts by considering valid combination of features, composition algebras or conformance to well-formedness rules. These advances are important, but assuring that all generated products will be valid is still a theoretical and practical problem.

Part II

Contributions

Overview of the Contributions

In this part of the thesis, we present our contributions to the state of the art of engineering model-based software product lines for the systems engineering domain. Our contributions are driven by the challenges raised at the end of the context chapter (see Section 1.6) and by the gaps left by the current approaches in the state of the art in relation to these challenges (see Section 2.8).

To illustrate this, Figure 2.9 shows the five challenges in the middle surrounded by our contributions, using the arrows to point to the challenge that is being tackled. We placed CVL at the top because it plays a major role in our work. CVL is not our contribution, however, as INRIA is a submitter of CVL to OMG's RFP, we participated on the consortium meetings, helping on the definition of some concepts of the language. Besides, we implemented our version of CVL tooling in INRIA, the kCVL [4].

In Chapter 3, we present – the core contribution of this thesis – our generative and automated approach to produce counterexamples of MSPLs (see ① in Figure 2.9). The work in this chapter has as main motivation to provide a novel approach to identify erroneous designs of MSPLs, that, despite of having correct variability models, can still derive wrong products with valid configurations. This approach needs only a metamodel as mandatory input, contributing to the early support for new DSLs. Because it uses the existing V&V mechanisms of systems engineering (e.g., conformance checking of models), the approach is less intrusive than the family-based or feature-based ones.

In Chapter 4, we show the different mechanisms for customizing the semantics of CVL's realization model and derivation engine, promoting specialized derivation support (see ② in Figure 2.9). Adjusting CVL's semantics is already a practice in Thales; engineers want to get rid of tedious operations, like always having to delete dangling references to a no longer existing model element; and they also want to enhance the reliability of their derivation engines – by incorporating secondary operations that will be executed automatically, complementary to the original semantics of a CVL variation point. We present this practice in a more structured way, showing three possibilities for performing the customizations; they can be used according to some non-functional requirement, like reusability or checkability.

In Chapter 5, we present an empirical study on the application of the derivation operators of CVL in real Java programs, assessing their capacity of generating correct programs (see ③ in Figure 2.9). This chapter helps to know the adequacy of the CVL's

[4] https://github.com/diverse-project/kcvl

variation points when applied to a complex programming language such as Java. It allows us to evaluate not only CVL and our counterexample-based approach, but also Java, as we are able to assess how hard is to randomly vary Java programs in a way that they continue to be compilable or testable, and therefore, how fragile they are.

Our contribution to the separation of concerns challenge relies on the use of CVL, considering that it can guarantee the degrees of modularity and orthogonality we need when designing MSPLs for systems engineering. Our contributions are intrinsically linked to real industrial challenges extracted from Thales Group; therefore we also provide in Chapter 6 a methodology to apply them in the industry, showing the engineers roles and activities in a defined process.

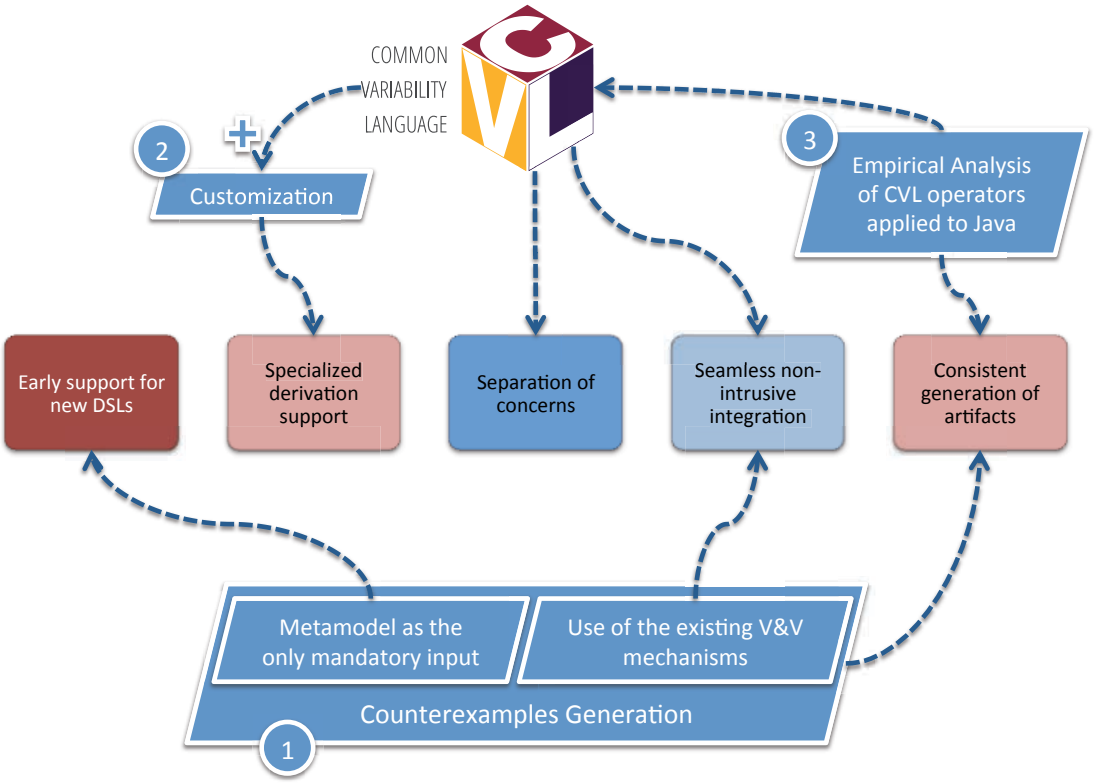


Figure 2.9: Overview of the contributions.

Chapter 3

Generating Counterxamples of MSPL

A one-size-fits-all support for designing MSPLs is unlikely, since models have to conform to their own well-formedness rules and business rules. Each time a new modeling language is used for developing an MSPL, the realization layer should be revised accordingly. We observed this kind of situation in the context of prototyping the use of CVL with Thales.

Without adequate support, a developer of an MSPL is likely to introduce errors. The tooling support can provide different facilities: antipatterns (counterexamples) to document what should be avoided during the design of an MSPL; domain-specific rules to avoid earlier the specification of incorrect mappings; examples to show possible correct MSPL, etc. Moreover, the support offered to domain experts should be ideally specific to a domain metamodel. Methodological support and guidelines are also needed to identify what constructs of a metamodel are likely to vary; to define an accurate realization model; or to develop specific derivation engines for a given modeling language.

In this Chapter, we provide a way to generate *counterexamples of MSPLs*, which are examples of MSPLs that authorize the derivation of syntactically or semantically invalid product models despite of 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 (see Section 3.1); this process is implemented in a tool named LineGen (see Section 3.2). In Section 3.3, 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. In Section 3.5, we extend the evaluation to our industrial case in Thales.

We discuss how counterexamples can guide practitioners when customizing derivation engines, when implementing checking rules that prevent early incorrect CVL models, or simply when specifying an MSPL (see Section 3.4). Overall, we conclude in Section 3.6 that the generative techniques and exploratory study help to construct solutions aware of the semantics of the targeted modeling languages when developing MSPLs.

The contributions of this chapter are published in [FBA+13] and in [FBA+14].

3.1 Our Approach

Our approach seeks to reveal MSPL designs that can generate invalid products even after a satisfiable set of Boolean choices (i.e., counterexamples, see Definition 2), as explained in Section 2.6. Our approach can help 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).

3.1.1 Counterexamples to the Rescue

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. We precisely want to support the two kinds of users before mentioned in their activities, by exploring the design space of their DSLs.

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 3.1 gives an “antipattern").

- developers of derivation engines can exploit counterexamples as testing oracles, anticipating the kinds of inputs that should be properly handled by their implementation. Furthermore, they can enrich the derivation engine with domain specific validation rules (customizing their operational semantics with one of the mechanisms described in Chapter 4). In addition, specific error reports can be generated when an MSPL is incorrect, inspired by the catalogue of counterexamples.

In our approach, we will randomly explore MSPL designs that are possible to be a counterexample. For doing this, we first define the concept; definition 2 formalizes this kind of MSPL as *counterexamples*, while in Figure 3.1, we show an example of a counterexample resulting in a wrong finite state machine when derivation is executed. Having this definition set, we show in the next sections how we proceeded to explore the design space of an MSPL of a given DSL.

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: $\mathcal{C}_{VAM} \neq \emptyset$;*
- *The set of BM conforms to its modeling language.*
- *$\exists\ c \in \mathcal{C}_{VAM}, \delta\ (CVL,\ c\ , BM) =\ DM'$ such that DM' does not conform to its modeling language.*

3.1.2 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 3.2 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, which are explained in details in the following subsections; the second and the third phases are part of the greater activity of generating a CVL model.

1. The first phase is the set up of the input that will be taken into account; different activities can be performed, depending on the input.
2. The second phase is the generation of a random variability model and of a valid random configuration.
3. The third phase is the generation of the relationships between the VAM and the base model elements, i.e., the variability model (VRM).
4. The fourth and last phase is to identify whether the generated model is a counterexample or not. In case it is not, we go back to the second step.

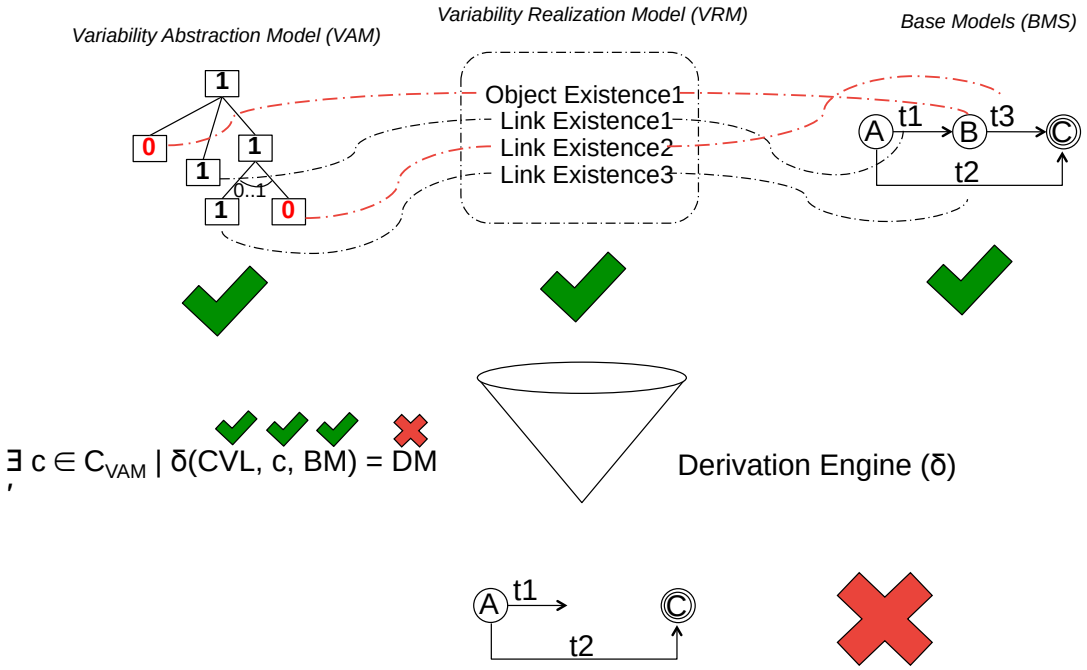


Figure 3.1: An example of counterexample.

3.1.3 Set up input

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, we can use mutation operators to increase the number of samples, or just not modify the base models. Mutations operators are basic CRUD (Create, Read, Update, Delete) operations on the base model that are applied randomly.

3.1.4 Generate VAM and Resolution

For generating the VAM and the VRM, the following parameters are required:

- The maximum depth of the VAM *(MAX_DEPTH)* and the maximum number

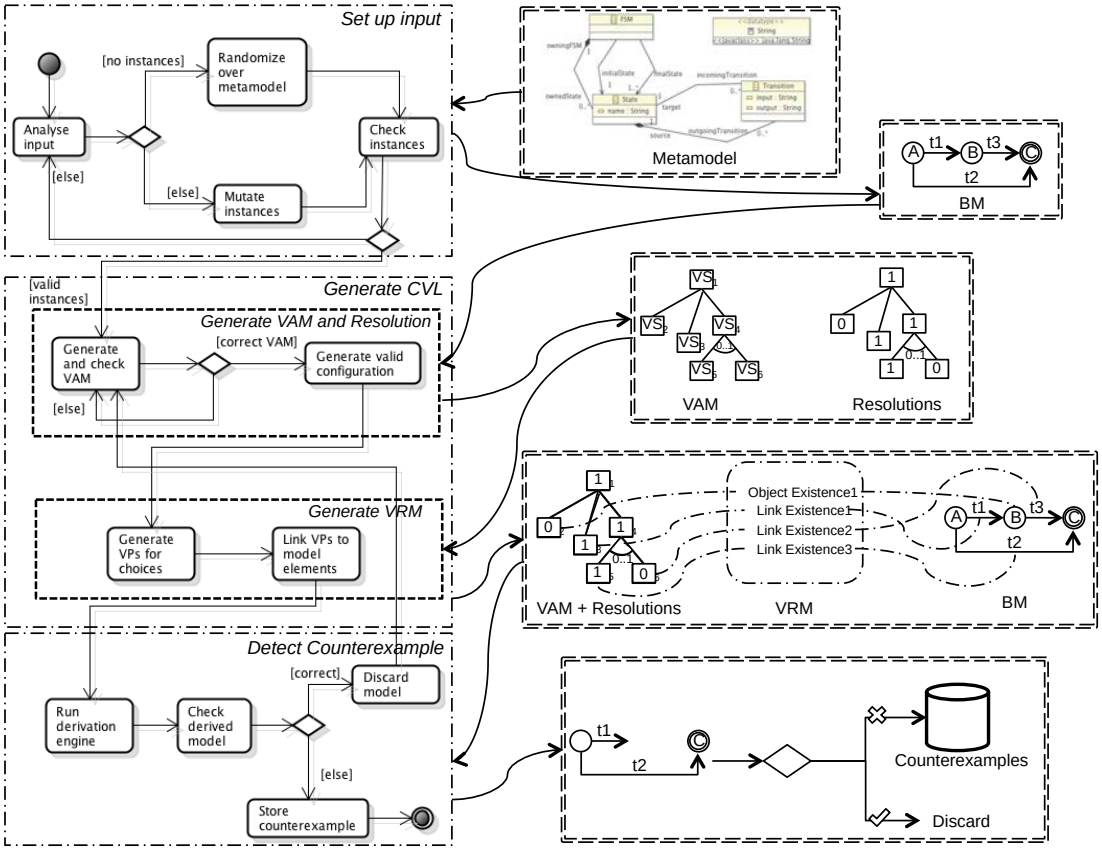


Figure 3.2: Overview.

of children for each $VSpec$ *(MAX_CHILDREN).*

- The percentage of $VSpecs$ that will be linked to variation points *(LINK_PERCENT).* For example, in Figure 3.2, the VAM was generated with a percentage of 66%, as four out of six $VSpecs$ are linked to VPs.

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, if the VAM is not provided by the user, we generate it, 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. The only imposed generation is of the root node of the tree, after, it is a random decision between creating (or not) each child.

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 [ACLF13] (we could

also rely on existing frameworks like FaMa [BSRc10]). The kinds of VAM we consider in this thesis 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.

3.1.5 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. The VRM generation can also be independent, from existing VAMs and BMs, one could then explore the possibilities of relationships between them.

3.1.6 Detect Counterexample

Although Figure 3.2 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.

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 meta-model. 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 we come back to the generate VAM phase, synthesising a new entire CVL model.

The derivation engine is an algorithm that visits each of the variation points in the CVL model, executing them according to the resolution of the variability model. Our implementation of the CVL derivation engine follows the operational semantics of each variation point defined in the CVL specification (for further details, see the Annex A of the CVL revised submission provided in `http://www.omgwiki.org/variability`).

To check if the derived model is correct, we relied on the EMF Diagnostician, using it as a black box to validate the conformance of the generated instance of the given metamodel.

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.

3.2 Tool Support

To support the process of generating counterexamples of MSPLs (exposed in the previous section), we developed a dedicated tool, called *LineGen*. Figure 3.3 gives an overview of the main features of LineGen. Depending on the inputs, the tool addresses different scenarios of counterexamples' generation – from the whole exploration of a modeling space (in the case only a metamodel is given) to the design of a specific MSPL (the variability model and the base model can be given by the user).

Specifically, the only mandatory input for LineGen is the metamodel of the base language. Additionally, the user can choose to provide existing base model and variability model; if this is the case, LineGen will not modify these models, setting them as immutable during the generation. To generate an MSPL example or counterexample, LineGen synthesizes a variability model, a configuration, a base model, and a set of realization relationships. LineGen calls the EMF's Diagnostician and checks the conformance of the base model with its input metamodel. After, LineGen checks the correctness of the variability model and the satisfiability of the configuration; to do so, it uses the reasoning engine within the FAMILIAR language. If they pass, LineGen carries on generating the realization relationships, finishing the CVL model.

After everything is generated, LineGen calls the CVL derivation engine, giving as input the generated CVL model (the triplet: variability model, realization model and base model) and a configuration. The goal of the call to the derivation engine is to determine whether the derived model is conforming to its modeling language. If it is, the CVL model given as input to the derivation engine is considered as an example of MSPL; otherwise it is considered as a counterexample.

We used different technologies as part of the LineGen implementation. As the user interface is an Eclipse 4 RCP application, it is written in Java. The core algorithms of the model generation parts are written in Scala. We used the EMF API to manipulate and check the Ecore metamodels and model instances. To benefit from automatic analysis of the variability model, we translated the VAM to the FAMILIAR language [FAM].

Figure 3.3 shows the graphical user interface of LineGen. The user must load the Ecore metamodel of the modeling language to be able to perform the generation steps (see ①). Once the metamodel has been successfully loaded—the Console (see ⑥) shows whether LineGen successfully completed an operation or not—it is possible to generate a base model by pressing the Generate BM button (see ②); a file named *BaseModel* is created with the chosen extension. The *Max Many* field should be set to limit the number of instances of a given model element.

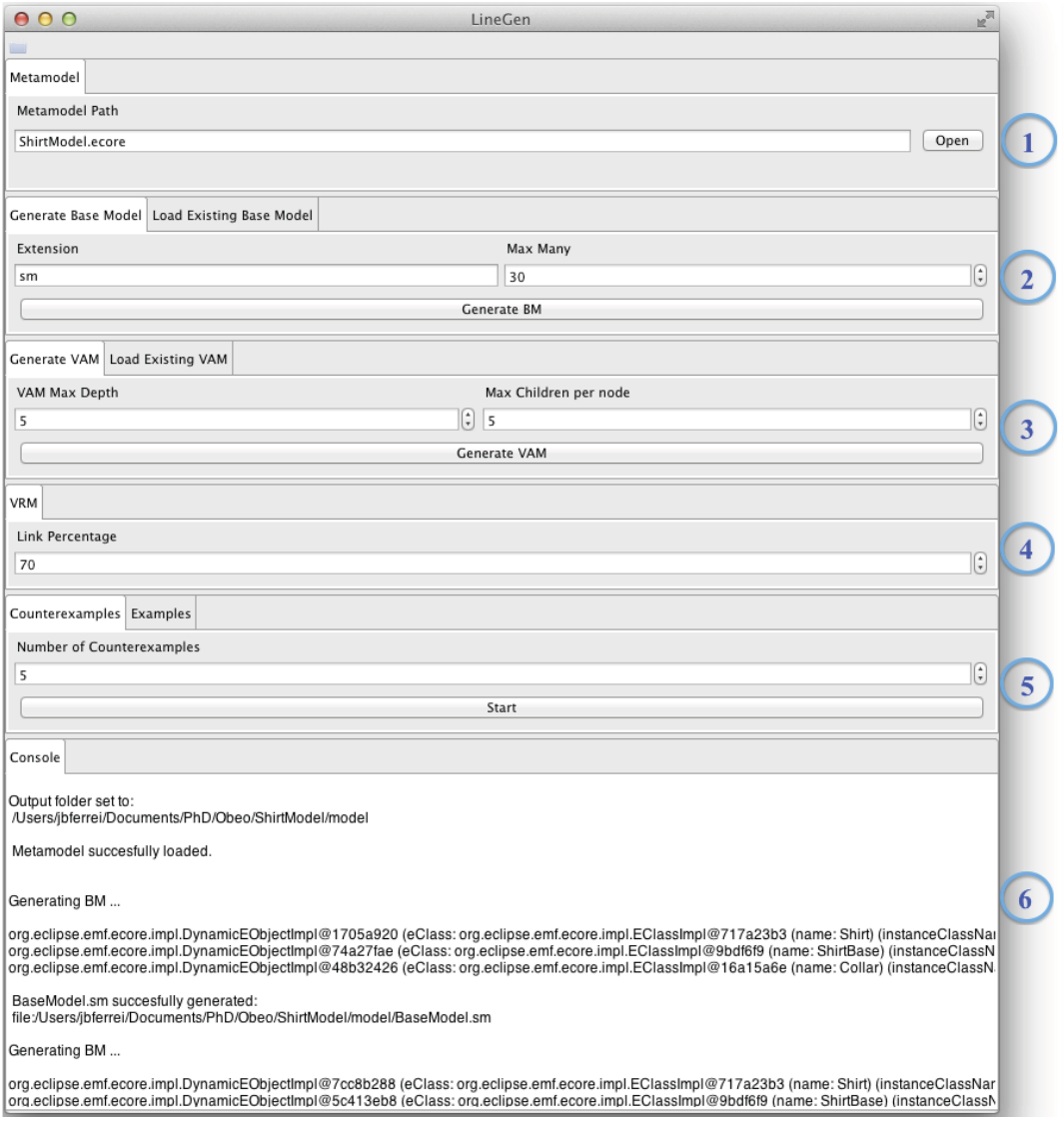


Figure 3.3: LineGen user interface

The same process applies to the VAM generation (see ③ in Figure 3.3). The user specifies the maximum depth of the VAM, as well as the maximum number of children per feature. After pressing the Generate VAM button, LineGen creates a CVL model with just the VAM part defined. In the VRM tab, the user can define the percentage of features linked to a variation point in the VRM (see ④).

In the Counterexamples and Examples tabs (see ⑤), the user can start the generation process that will randomly search for examples or counterexamples of MSPLs. If the user chooses to use the Load Existing Base Model or Load Existing VAM tabs, LineGen uses the loaded models without modifying them and just generates VRM models. The field Number of Counterexamples determines when LineGen has to stop the

search. The Console tab also provides exception messages, in case an unexpected error occurs. More details about LineGen can be found online: `https://code.google.com/p/linegen/wiki/LineGen`.

3.3 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?

3.3.1 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 validation 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. 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 3.4, 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. The linear trendlines are a good fit to the obtained time values, with R^2 values close to 1. Each time value is an average of 10 executions, this was done to minimize the random effect.

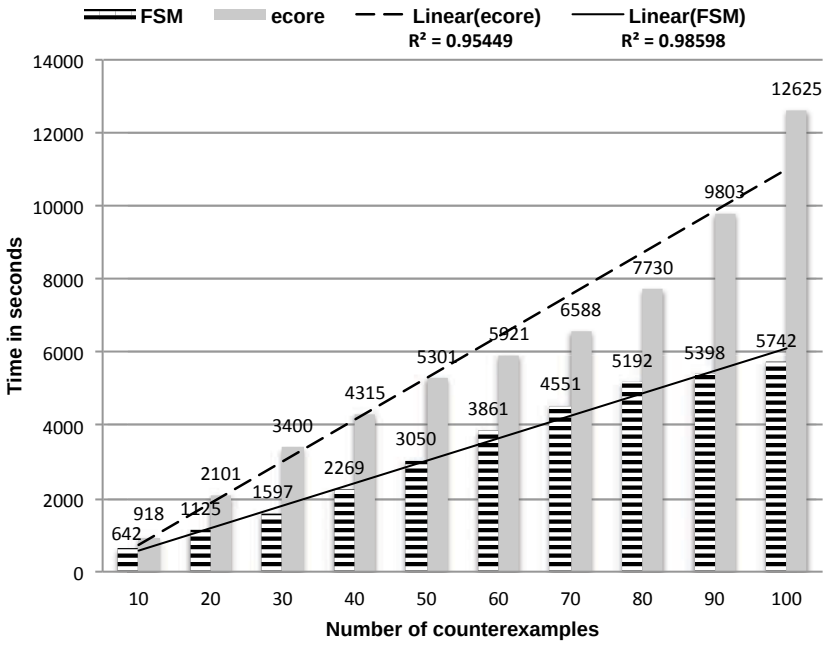


Figure 3.4: Counterexamples for FSM and Ecore.

3.3.2 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[1].

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. 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.

3.3.3 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 come from structural inconsistencies. Finally UML contains so many validation rules that it is unfeasible to create a valid

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

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.

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 [CBS12] showing there exists different "styles" of expressing business or domain-specific rules within a metamodel.

3.3.4 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 valid CVL model and a correct 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. Rule 1 prevents a substitution between a final state and an initial state, and vice versa. Rule 2 constrains the fact of having an object existence that targets the initial state of an FSM. These rules have been implemented in Scala and can be written in few lines using an OCL writing style, as shown in Listing 3.1.

Listing 3.1: Antipattern rules for FSM

```
1  def checkVRM(f:FSM,vrm: VPackage):Boolean = {
2      vrm.asInstanceOf[VPackage].getPackageElement().foreach(e=> {
3  /*Rule 1: Replacing a final state by an initial one, and vice versa, is
4  forbidden.*/
5           if (e.isInstanceOf[ObjectSubstitution]){
6          var p = e.asInstanceOf[ObjectSubstitution].getPlacementObject().getReference()
7          var p1 = e.asInstanceOf[ObjectSubstitution].getReplacementObject().getReference()
8          if ((f.getFinalState().contains(p) && f.getInitialState().equals(p1)) ||  (f.getFinalState().
               contains(p1) && f.getInitialState().equals(p))) return false;
9           }
10 /*Rule 2: Pointing an ObjectExistence to an initial state is forbidden.*/
11          else if (e.isInstanceOf[ObjectExistence]){
12            e.asInstanceOf[ObjectExistence].getOptionalObject().foreach(p=> {if (f.getInitialState().
                  equals(p.getReference())) return false;})}})
13     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.

3.4 Discussion

Besides the checking operations, the time results presented in Figure 3.4 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 an 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 [ZMP12]) 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.

3.5 Approach in an Industrial Case

In the last session, we evaluated our approach against well-known modelling languages; we could verify that it produces counterexamples in a reasonable time and we could also assess properties of the counterexamples. In this section, we present how the approach performs facing an industrial case. First, we describe the company's scenario; second, we report on how we could successfully apply the approach on it; and finally, we reproduce the applicability and effectiveness experiments done in the RQ1 of the evaluation.

3.5.1 Thales Scenario

Thales is a large company involved with different industry sectors (aerospace, space, defence and transportation areas, etc.); they produce software intensive systems, using model-based technologies, and they seek to evolve towards a product line approach. Thales already has a well-established and functional model-based method for developing their systems and software, the ARCADIA, however they seek to leverage this development from single software to families of software, maintaining their safety and quality standard [FBBLN12].

The ARCADIA method is a viewpoint-based architectural description, defining 5 different abstraction levels of a system, following the ISO/IEC 42010, Systems and Software Engineering - Architecture Description [ISO10]. Thales' engineers use numerous domain specific modeling languages to develop integrated sets of systems according to ARCADIA. These languages are built within a set of dedicated representations to analyze specific problems. The language workbench provides a set of customizable and highly dynamic representations working seamlessly together on top of models. These representations can be combined and customized according to the concept of Viewpoints.

Views, dedicated to a specific Viewpoint, can adapt both their display and behavior depending on the model state and on the current concern. The same information can also be simultaneously represented through diagram, table or tree editors.

These languages are defined as a set of 20 metamodels with about 400 metaclasses and about 200 validation rules; they model the ARCADIA method in an eclipse-based environment. Besides, this workbench is extensible and new languages can be defined to design specific viewpoints of a system. Therefore, leveraging product line engineering for each of these languages and domains is very expensive and error-prone; it has to be supported by automated tools.

Several stakeholders have to work during the design process on the tool chain:

- Product-line engineers who have to identify the commonalities and the variants and in charge of designing the VAM and the VRM.
- Product engineers who have to create specific products, focusing on creating valid products regarding a set of requirements.
- DSL designers who are in charge of creating or extending existing DSLs (base metamodel). They define where and how we can put variability within (at the M2 level) the architecture and the derivation semantics [FBLNJ12]

The use of the proposed counter example framework aims at easing the correct cooperation between these stakeholders. It is used to provide a pragmatic approach to guide these stakeholders to design CVL model that provides only valid products.

3.5.2 Approach Application and Results

We applied the approach to the Thales' representative sample model of weather balloons; this base model has 2079 model elements and 563Kb and, despite of being one single subdomain, it can serve as a pilot application for other similar areas of the organization. The set of metamodels and validation rules of ARCADIA are considered as input to the approach. In contrast of what we did to evaluate the approach in a generic way (generating everything else besides the metamodel), we could simplify the generation because Thales provided a variability model and the aforementioned preliminary base model, narrowing down the problem space. Therefore, we fixed the VAM and the BM, randomizing only over the configurations of their variability model and generating the set of variation points to compose the realization model. However, it was necessary to adapt the implementation to meet some technical requirements from Thales for loading and saving the models.

Reproducing the same experimental setup of RQ1, we performed 10 rounds and measured the average time for generating 100 counterexamples. The results in seconds are shown in Figure 3.5. We could verify that in a situation where the VAM and the BM are provided, it is around 27 times faster to generate the same amount of counterexamples, and the curve still behaves linearly.

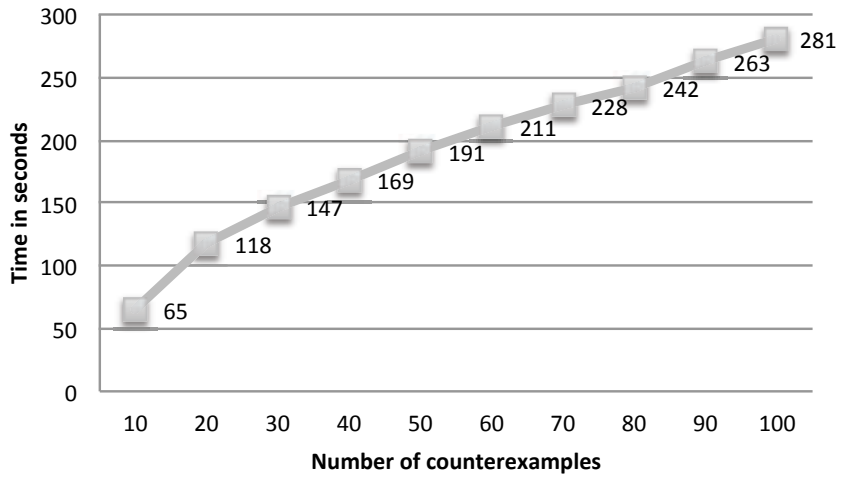


Figure 3.5: Counterexamples for ARCADIA sample model.

About 30% of the models generated for this domain were wrong, meaning that, in average, if we randomly define realization relationships among the features and the model elements of this domain, almost one third can result on counterexamples. Another interesting result is the fact that only one OCL rule added to the VRM can remove 50% of the counterexamples (Forbidden an object existence on a specific kind of model element "*EventSendCallAction*") and 80% of the counterexamples can be removed in writing 8 basic OCL rules. With this example, we can show that the generation of counterexamples from a reference model can help to detect some anti-patterns that can be easily constrained and detected for a particular domain. The result is the improvement of the use of CVL in this industrial context an early detection of CVL model that capture invalid products.

3.6 Conclusions

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 chapter, 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 ex-

plained in details each step of our generative approach and illustrated it with a running example. The tool LineGen, built on top of CVL and modeling technologies, supports the generative process. It enables practitioners to explore the whole design space of a given modeling language but also to focus on a specific MSPL with a pre-defined variability and base models. We performed experiments to assess the applicability and effectiveness of the tool-supported approach. The conducted experiments allowed us to evaluate the approach when applied to different modeling languages, at different scales of complexity. We could successfully generate counterexamples for each modeling language in a reasonable amount of time, which could be drastically reduced when the approach received additional input. In addition, we explored the natures of errors found in the counterexamples and our ability to detect antipatterns. We also reported on our experience when instantiating the approach and LineGen in an industrial context.

Chapter 4

Customization of Derivation Semantics

Recalling the motivations of Section 1.3.1 for varying the operational semantics of CVL, we have that: the semantics can vary within different metamodels (e.g., it is different to exclude a UML Class and a BPMN activity, however both are model elements that can be pointed by an Object Existence); within the same metamodel and with the same model elements (e.g., excluding a Singleton Class is different of excluding a Parent Class, in terms of secondary operations it may lead).

In this chapter, we first introduce the two possibilities to exploit the generated counterexamples (see Section 4.1). We then expose the mechanisms to implement and extend the semantics of CVL's variation points (VP) (see Section 4.2), showing how this semantics can be customized in practice according to a domain or to different model elements. In Section 4.3, we summarize the mechanisms presented and give a brief comparison. Some of the contributions presented in this chapter are published in [FBLNJ12].

4.1 What to do after generating counterexamples?

After generating counterexamples for a given DSL, the MSPL infrastructure engineer (see Roles in Section 6.1) can decide to use the knowledge acquired to: design checking rules to detect errors and develop repair actions in an MSPL design; or to change the derivation semantics and avoid these errors when executing the derivation.

In Figure 4.1, we illustrate the following situation. After having successfully generated counterexamples for finite state machines, we observe a type of counterexample that is frequently produced: the case of excluding a state and letting dangling references that used to point to it. This counterexample is due to an object existence, referring to a state with incoming and outgoing transitions, that had its operational semantics executed during the derivation algorithm.

As shown in Figure 4.1, there are two options to handle this situation. The first one would be at design time, including a checking rule that would detect if an object

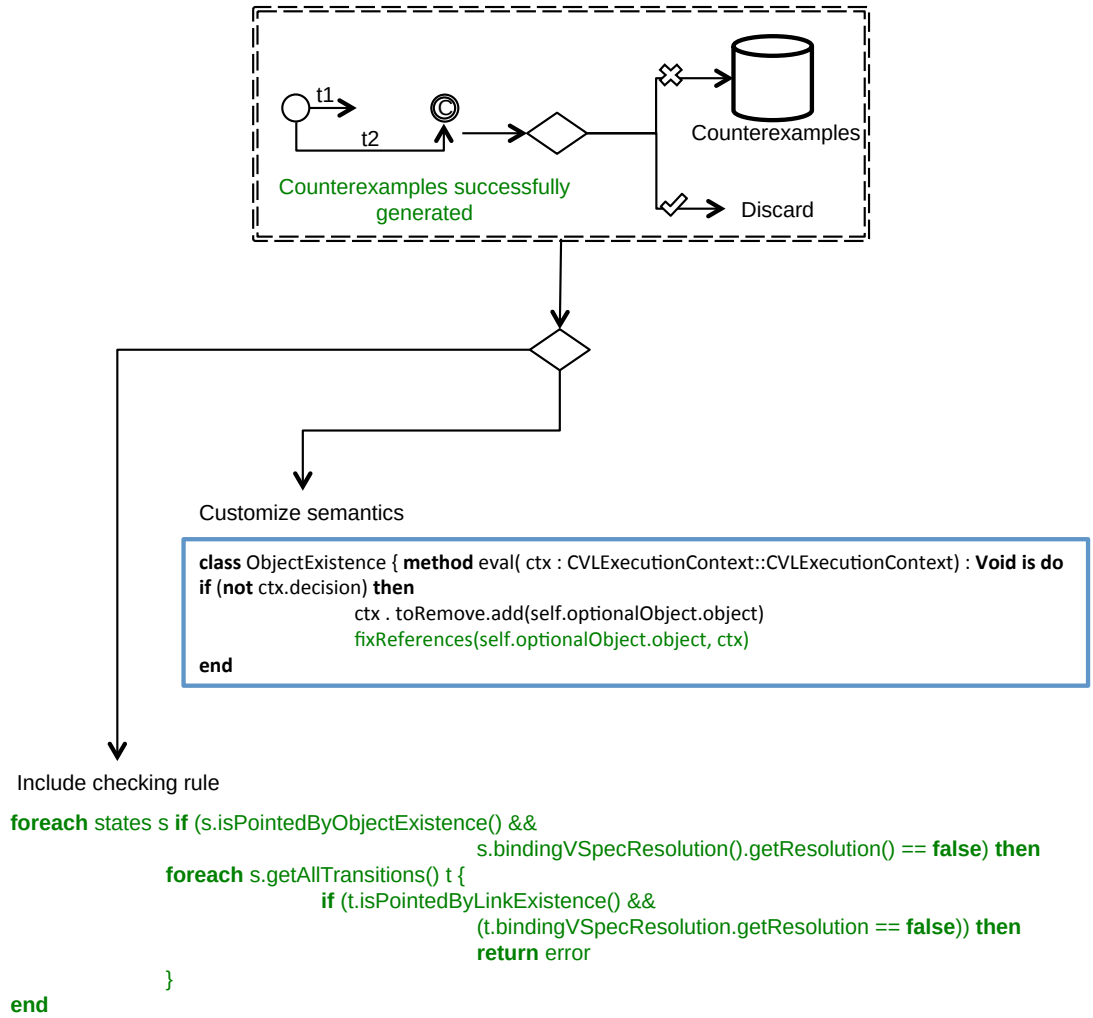


Figure 4.1: Customizing the derivation semantics or including checking rules

existence that points to a state will be executed, in case it will (i.e., its binding VSpec is negatively decided), there must be link existences pointed to all its incoming and outgoing transitions, and they shall also point to negatively decided VSpecs (i.e., features configured to false). One advantage of this option is that the designer of the MSPL can know that his design is faulty and then correct it. A disadvantage is if the error is very frequent, he/she will have to spend a lot of time correcting the design, including additional variation points.

The second option is to customize the semantics of the object existence variation point. The new semantics would automatically make the secondary operations after removing a state, cleaning all the dangling references. The disadvantage is that this can only be done if the error is a pattern in the domain – in this case it is true, a transition of a finite state machine must always have at least one source and one target state –

otherwise, the semantics would be adequate to one case and broken for others.

The main advantage of customizing the derivation semantics is that engineers could get rid of tedious operations, such as with dangling transitions; it also increases the safety of the MSPL, in this case, assuring that it would never generate finite state machines with dangling transitions. In this chapter, we choose to further explore this idea of customizing the semantics of the CVL's variation point

4.2 Approaches to customize CVL's derivation semantics

An engineer that wants to specialize the CVL's derivation semantics can do it in different ways; we present three in this section, as illustrated in Figure 4.2. The first approach is the **static introduction of semantics**. It consists on directly redefining new semantics into the derivation engine. The meaning of the variation point is changed to cope with other specific need. It is the simplest way to customize the semantics, as the engineer re-implement the code of the CVL derivation engine, developing a new behaviour.

The second approach is the **opaque customization**. CVL proposes a set of VPs with a well-defined semantics and keeps one type as an extension point to implement its own semantics: the *Opaque Variation Point* (OVP). The OVP is a black box that can define an arbitrary behaviour to execute during derivation; they can be unknown algorithms from third parties and just be invoked by the derivation mechanism. The use of OVPs can be seen as a mechanism to propose a particular semantics for the derivation engine.

The third approach is the **extensible customization**. As the name says, this approach consists on extending the original semantics of the derivation engine, but without changing the original one; the new semantics is added as an explicit and focused extension. An additional logic is included in the derivation engine, in order to either switch between the extensions or just choosing which should be executed according to the nature of the elements to which the VP points.

4.2.1 Semantics in CVL

Before presenting how the customization approaches can be implemented in practice, we show the basic concepts of how the original semantics of CVL was defined and developed. We consider CVL to be a modeling language as any other. Therefore, next subsection shows what is the practice of implementing the semantics of a modeling language.

4.2.1.1 Weaving semantics into a modeling language

Metamodeling is the current practice when defining a modeling language; it consists on defining the structure of a language, using concepts and relationships. This activity is supported by metalanguages like MOF, EMOF [MOF02] and Ecore [SBMP08]. However, in many cases, it is not sufficient to define only the language structures; the

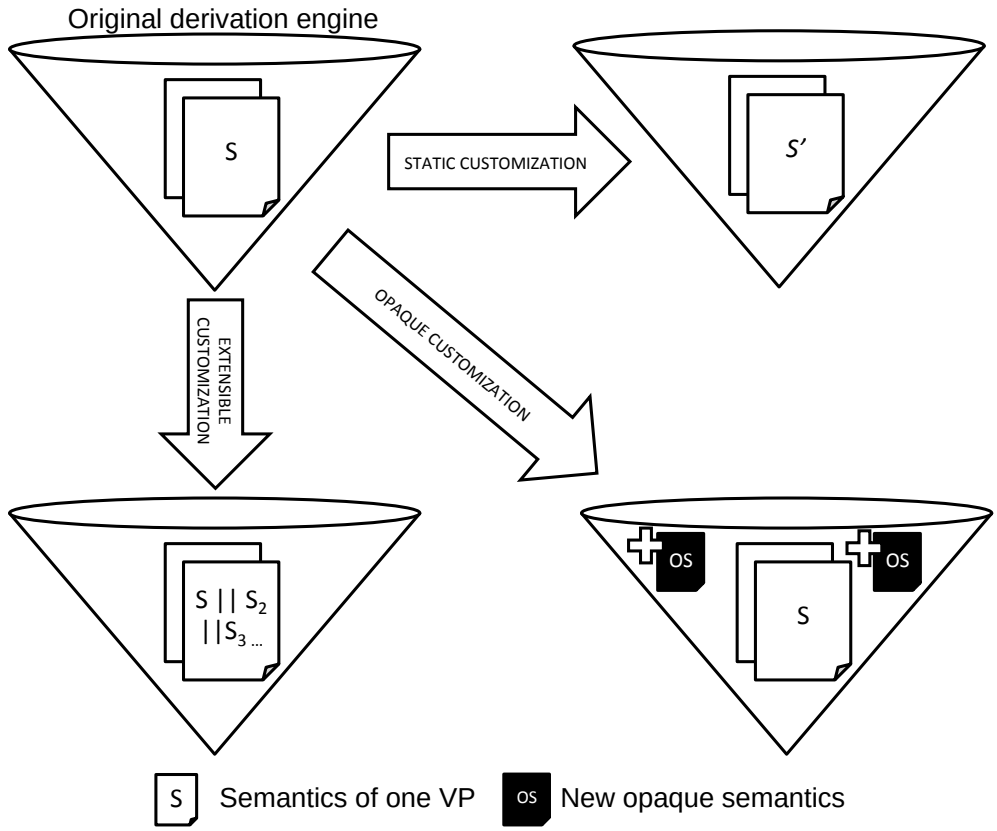


Figure 4.2: Three approaches to customize the semantics of the derivation engine.

behaviour is also a major concern in many languages. It can be seen as the actual meaning and the actions of the language; it is what makes a language executable.

One could implement this behaviour in a general purpose imperative language, like Java, or in a declarative one like OCL. However, they are not the best fit for this practice, for example, Java does not contemplate some of the concepts existing in MOF, such as associations, enumerations, opposite properties, multiplicities, derived properties, etc.). In our team, we use *Kermeta* [MFJ05] to handle this challenge of implementing a behaviour for a modeling language. Kermeta is a language workbench made for specifying and designing domain specific languages (DSL). For this, it involves different languages, depending on the concern: abstract syntax (i.e., metamodel[1]), static semantics and behavioural/operational semantics. The workbench integrates the OMG *de facto* standards EMOF and OCL, respectively for specifying the abstract syntax and the static semantics; it also provides the *Kermeta Language* to address the specification of the operational semantics and to integrate an action language in EMOF. The workbench composes these different concerns into a standalone execution engine (interpreter

[1]Using metamodel as synonym of abstract syntax is one definition in the community. For some researchers, "metamodel" is sometimes referred to abstract syntax plus static semantics.

or compiler) of the DSL.

In Kermeta, all pieces of static and behavioral semantics are encapsulated in metamodel classes. The `aspect` keyword enables DSL engineers to relate the language concerns (abstract syntax, static semantics, behavioral semantics) together. It allows DSL engineers to reopen a previously created class to add some new pieces of information such as new methods, new properties or new constraints. It is inspired from open-classes [CL00] – the keyword `require` enables the composition. A DSL implementation *requires* an abstract syntax, a static semantics and a behavioral semantics. The require mechanism also provides some flexibility with respect to static and behavioral semantics. For example, several behavioral semantics could be defined in different modules (all on top of the same metamodel) and then chosen depending on particular needs (e.g., simulation, compilation).

4.2.1.2 Weaving Semantics into CVL

We translated CVL's metamodel to the .ecore notation; it is the base metamodel in which we will introduce operational semantics. Consequently, the CVL metamodel is required as an input and can be easily invoked in Kermeta using the `require` keyword, (*e.g.*, `require CVLMetamodel.ecore`).

Once the CVL metamodel is loaded, it is possible to weave the operational semantics into any model element in the metamodel. The Listing 4.1 shows how we can simply attach operational semantics into the CVL abstract syntax (metamodel). The code that implements the operational semantics of a model element is placed in an `aspect class` block, named according to the model element name (line 1). In the case of the variation points, we define an abstract operation to evaluate the operational semantics of the variation point (line 2). This operation is abstract because the actual implementation is in the concrete class that inherits from the *Variation Point* class (e.g., Object Existence).

Listing 4.1: Eval method header in all the VPs

```
1  operation eval(ctx:CVLExecutionContext::CVLExecutionContext):Void is abstract
2  }
```

Each concrete variation point has an *eval* method, which overrides the abstract operation and must contain the operational semantics to be executed, following an implementation of the Interpreter Design pattern [GHJV94]. An execution context (*CVLExecutionContext*) is provided as a parameter of the *eval* method. This context stores relevant information to the execution of the materialization engine, such as the set of selected/unselected *VSpecs* and the set of model elements in the resolved model. The operational semantics in Kermeta of all concrete variation point is inside the CVL submission document[2], in the Annex A: More on semantics. To be concise, we will adopt the *Object Existence* variation point as working example.

The Listing 4.2 presents the operational semantics of the *ObjectExistence* variation

[2] www.omgwiki.org/variability/

point. First, in line 3, it is verified whether the binding *VSpec* (i.e., similar to feature, see Section 2.5) of the current variation point is selected or not. If the *VSpec* is not selected for the current materialization, we need to remove the corresponding element in the base model. In line 4, we navigate to the binding object of the current variation point (`self`) to provide the optional element in the base model that is inside the collection *ctx.domainResource* to the method remove.

Listing 4.2: Excerpt of the ObjectExistence semantics

```
1 aspect class ObjectExistence {
2 method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
3  if (not ctx.decision) then
4   ctx.toRemove.add(self.optionalObject.object)
5  end
6 end
7 }
```

This operational semantics is executed whenever the derivation engine runs to produce a product after a configuration. In a negative derivation algorithm, the Object Existence is linked to existing model elements, and if the binding choice is set to false, the operational semantics is executed, in this case, the object is removed from the base model.

4.2.2 Static customization

By using the built-in *require* composition mechanism of Kermeta, it is possible to statically customize the semantics of a CVL variation point. Indeed, *require* provides a mechanism to weave aspect in an existing metamodel. Therefore, the DSL engineer can reopen a previously created metaclass to add new pieces of information such as new methods, new properties or new constraints. It also allows engineers to easily replace the behaviour of an existing method. The method (*eval*, see Listing 4.1), which is introduced in all the variation points (*ObjectSubstitution*, *ObjectExistence*, ...) can be changed by requiring a new Kermeta file. This modification is static, modifying the types and requiring to recompile all the CVL's Kermeta implementation.

This extension mechanism has two main drawbacks. First, Kermeta does not allow the new implementation to call the previous aspect implementation, contrarily to the situation in which we can call the code of an operation contained in the super-class with the keyword *super*. Secondly, using this mechanism, the DSL engineer can change (and potentially break) completely the CVL implementation. Kermeta does not provide any checker to ensure that a new implementation is a *refinement* of the previous implementation. The main advantages is the fact that the extension is modular and can be statically plugged or unplugged. As an example, we present in Listing 4.3, an excerpt of a customization of the CVL Object Existence, which, besides removing the model element (line 4), also fixes dangling references (line 5).

Listing 4.3: Excerpt of the ObjectExistence semantics with fix references procedure

```
3 class ObjectExistence { method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
4  if (not ctx.decision) then
```

```
5     ctx.toRemove.add(self.optionalObject.object)
6     fixReferences(self.optionalObject.object, ctx)
7    end
8   end
9   }
```

4.2.3 Extensible customization

The basic semantics used in the default CVL implementation is the following. Each variation point can modify the model to change relationships between model elements and can introduce new model elements. To remove a model element, each variation point acts on a context that contains a list *toRemove* of the model elements that must be removed. Removing elements of a base model is performed at the end of the derivation to avoid side-effects among variation points.

With this behaviour, CVL combines positive variability and negative variability. The default semantics for the *remove* implementation is the following. Each element contained (containment relationship) by an element that must be removed is also removed. All the references of an element that must be removed are set to *null*. All the elements, that reference an object that must be removed through a reference with a violation of the lower cardinality are also removed, *e.g.* if $a : A$ references $b : B$ and A is associated exactly with one B, if b is removed, a is also removed. This can already be seen as a semantic customization as it has an additional load of operations that must be performed for a given variation point.

We can introduce in the default semantics a strategy pattern [GHJV94] to provide the ability of dynamically specializing the default semantics. The idea is that a domain expert can define a new CVL semantic extension and can register it. During the derivation, when a model element has to be removed, all the registered extensions are called to determine the list of model elements to be removed (as depicted in Figure 4.3). To implement a new metamodel extension, the DSL expert has to create an object that respects the following interface (see Listing 4.4).

Listing 4.4: Interface for remove strategies

```
1  ToRemoveStrategy { method remove ( objToRemove: Object, ctx: CVLExecutionContext) : Void is
       abstract
2  }
```

Listing 4.5: Excerpt of the ObjectExistence semantics with strategies

```
10  ObjectExistence { method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
11   if (not ctx.decision) then
12    ctx.remove(self.optionalObject.object)
13   end
14  end
15  class CVLExecutionContext {
16  method remove( obj:Object) : Void is do
17   self.toremove.add(obj)
18   //Call the strategies
19   self.toremoveStrategies.each{strat | strat.remove(obj, self)}
```

```
20 end
21 ...
22 }
```

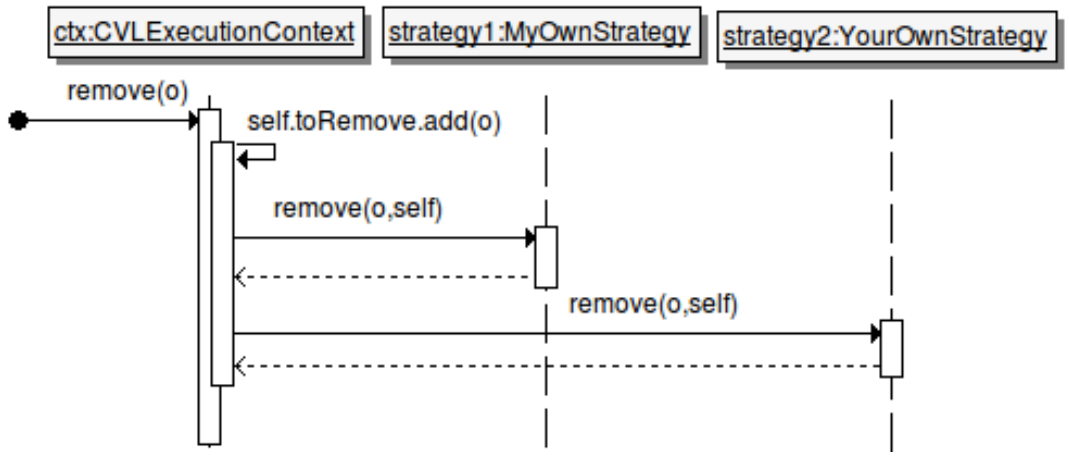


Figure 4.3: Strategies Sequence Diagram

This extension mechanism provides several benefits. First, it ensures that the default semantics of the CVL variation point is respected. Indeed, domain engineers can only refine the semantics in removing elements, and not directly in the variation point. It can be compared to the idea of post directives in Kompose [FBFG08]. Second, new strategies can be registered or unregistered dynamically. Finally, each specialization can be modularized in a distinct building block.

4.2.4 Opaque customization

The last way to customize the CVL derivation semantics is the use of *Opaque Variation Points* (OVP). An OVP is the Variation Point in which the behavior is defined by using an expression defined in an action language. We currently propose an implementation that supports OVP definition in Groovy [3], in Javascript or in Kermeta. With these action languages, designers can modify the base model directly. Each of this Variation point can access to a context that contains the list of Objects to remove (*toRemove*), the list of *objectHandles* associated to this Variation Point (*ctx*), the list of variable and their associated value defined in the the resolution model ($args$), and a map of key value that can be used to propagate value between the execution of variation points *map* (see Figure 4.4). An example of OVP is defined in Listing 4.6 as an expression attribute of the OVP, it adds all the UML properties that references a base model element that must be removed.

[3] http://groovy.codehaus.org/

Listing 4.6: OVP example in Groovy

```
ctx.each {e  ->
 org.eclipse.uml2.uml.PackageableElement elem = e;
 org.eclipse.uml2.uml.Package p1 =elem.getPckage();
 p1.getMembers().findAll{m ->
  m instanceof org.eclipse.uml2.uml.Association}
 .each{m->
  m.getProperties().entrySet().findAll{ p2 ->
   p2.getType().equals(e)}.each{ m2 ->
    notSelected.add(m)
   }
 }
}
```

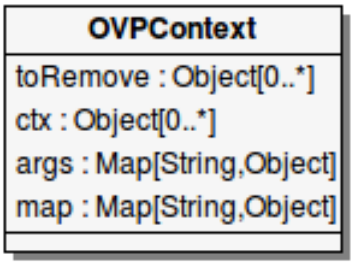


Figure 4.4: OVP context execution

The main drawback of this extension mechanism is the opacity of the OVP itself. No CVL checker can ensure the correctness of the variability model and it becomes complex to understand the expected behaviour of a variability realization model.

4.3 Synthesis

	Benefits	**Drawbacks**
Static	Modular	No guarantee, highly invasive
Extensible	Modular, Dynamic	Less flexible
Opaque	Flexible	Black Box, uncheckable, no reuse

Table 4.1: Synthesis of the three extension mechanisms

Table 4.1 shows a comparison of the three extension mechanisms provided in our CVL implementation to support the customization of the CVL semantics for a domain model. We could observe that the second mechanism is generally the best to specialize the CVL semantics for a specific metamodel. Indeed, it does not change the CVL semantics but it only refines the semantics of removing, adding or substituting an element. Opaque Variation Point is often useful even if we loose the ability to understand the materialization and therefore of analyzing the CVL realization model. Besides, it is currently missing in CVL the notion of Opaque Variation Type to ease the reuse of

an existing OVP. The first mechanism is built-in within Kermeta but seems to be dangerous for the case of CVL because experts should be perfectly aware of the previous implementation to change it without introducing side-effects.

4.4 Conclusion

As CVL is a generic language for handling variability in any domain, there is often a need for specializing its semantics. We have shown in this chapter three different ways of making this specialization, comparing them and showing examples. We believe that these customizations are the key for building derivation engines better fitted to the specific domains. The customizations can be assisted by the counterexamples approach presented in Chapter 3, using its random explorations of a particular domain to find antipatterns of MSPLs. The customizations can help engineers to ease the task of defining a realization model, by encapsulating tedious operations or reassuring the correction of the transformations.

Chapter 5

Experimenting CVL variation points with Java program constructs

Each time a domain specific modeling language or a programming language is used for developing an SPL, practitioners (e.g., domain experts or software developers) need to understand the language constructs subject to variation and the means to realize a variant. In the case of Java, numerous constructs are subject to transformation: we can add a parameter into a constructor, remove a statement, substitute a field, etc. Some transformations lead to errors; some others not.

In particular, not all constructs of a language are subject to variation because of the numerous well-formed and domain-specific rules. For instance, removing a `return` statement in a non-void Java method is not possible; adding a `try` block without a `catch` block either; replacing the type of a parameter by another unrelated type is unlikely to produce a correct variant, etc.

Our industrial experience with Thales confirmed the relevance of the problem and its practical difficulty, as explained in previous chapters. Each time a new modeling language is used for developing a variability-intensive system, domain experts need to understand the language constructs subject to variation and the means to realize a variant.

Though generic foundations and tools (independently of any particular technology) for describing variations are emerging [EW11, AKL13], the *specificity* of the language's syntactic structure and semantics has to be considered at some points. In practice, each time software artefacts are concerned with variation, the *what* and *how* (i.e., the removal, adding, or replacement of a program or model element) of the conforming languages should be considered carefully. The problem impacts both users of languages (e.g., Java) and developers of tools (e.g., integrated development environment). A traditional approach is to hand-craft a solution by relying on domain knowledge and empirical observations.

In this chapter, we empirically answer the question: which transformations (i.e.,

derivation/realization operators, or variation points) can synthesize variants of Java programs that are incorrect, correct and perhaps even conforming to test suites? We adopt an approach with no assumptions about the targeted language that relies on full extensive automations for exploring a variation space. We implement source code transformations, based on the Common Variability Language, that add, remove, substitute any kind of element of a Java program. We automatically synthesize 376,185 program variants based on source code elements in a set of 8 real large Java projects (up to 85000 lines of code). We obtain a comprehensive panorama of the sanity of the transformations based on statistical data collected and qualitative reviews of synthesized Java variants.

This chapter is organized as follows. Section 5.1, introduces our model-based approach for combining variability modelling and automatic program transformation to understanding what can be vary in a Java program with CVL derivation operators. Section 5.2 presents the experiment, its methodology and the hypotheses to be tested. Section 5.3 and Section 5.4 analyse the results, discuss them and present the threats to validity of this experiment. Section 5.5 concludes the paper and presents future work.

5.1 Automatic synthesis of Java Programs with CVL

This section presents an overview of the approach to automatically synthesize variants of Java programs using CVL. The goal of this approach is to empirically analyse the suitability of CVL derivation operators when they transform real Java programs.

5.1.1 Definition

In CVL, the operators are always linked to a target element. Consequently, we will further refer to the definition of a transformation – T as a pair $\langle O, E \rangle$, in which O is a kind of operator (from the aforementioned list) and E is a type of targeted program element (e.g., code statement, class, package).

Given a program transformation T, a program P that successfully compiles and a test suite TS that passes on P, the possible results for a transformed program $P' = T(P)$ are:

1. P' is syntactically incorrect and contains compilation errors–P' is a *counterexample*;
2. P' is syntactically correct and successfully compiles but at least one test case in TS fails–P' is a *variant*;
3. P' compiles and all test cases in TS passes–P' is a *sosie*[1].

5.1.2 Process overview

Figure 5.1 shows an overview of the process of transforming an input program P. First, we use Spoon[PNP06] to extract P's Abstract Syntax Tree (AST), which provides the

[1]Sosie is a French noun that means "look alike" and it has been previously defined in [BAM14].

set of program elements and their relationships. This step makes possible to handle P as a model, therefore we can use the concept of model-based SPL with CVL; another reason is to reuse the program manipulation facilities provided by Spoon. Second, we use the AST of P and the list of CVL realization operators as input to the transformation. In the transformation step, we pick a random program element and a random operator, composing a transformation and then applying it to the AST; the result of this is a transformed AST' of P. As a third step, we print back as source code AST', having as result a transformed program P'. Finally, we try to compile P' and also to test it against the test suite.

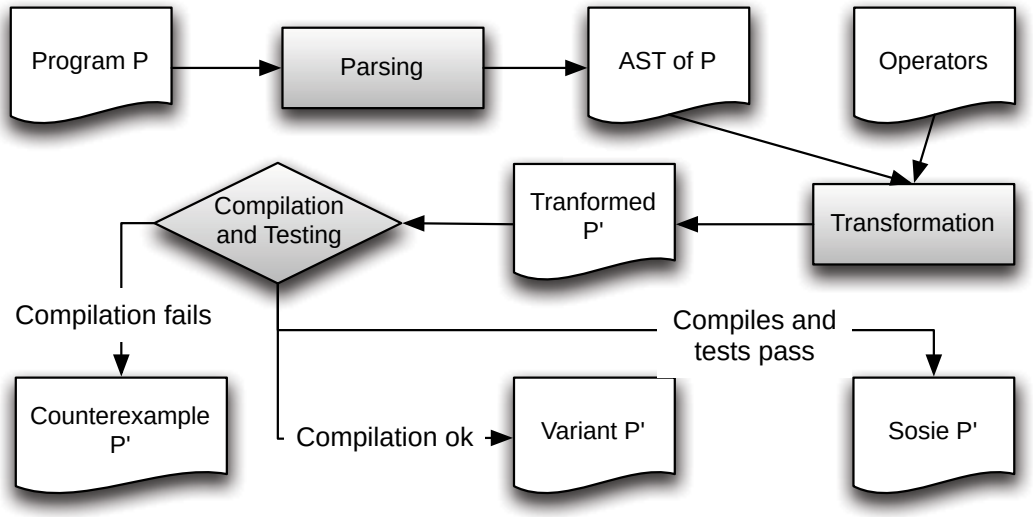


Figure 5.1: Process to transform Java programs.

Listing 5.1 shows an excerpt of a transformed program that does not compile; precisely, it is the result of an *Object Substitution* in the statement of the line 38 inside the constructor of the class *UniformReservoir* of the *metrics*[2] project. The replaced statement is commented (instead of actually deleted, in order to facilitate visualization and retrieval) and a new statement is placed right after. In this case, one of the reasons it does not compile is because the variable *registry* was not declared before.

Listing 5.1: Object Substitution generating a counterexample.

```
1 //class com.codahale.metrics.UniformReservoir, line 38
2 public UniformReservoir(int size) {
3        this.values = new AtomicLongArray(size);
4        for (int i = 0; i < values.length(); i++) {
5            values.set(i, 0);
6        }
```

[2]http://metrics.codahale.com/

```
7         //subtitution
8         //count.set(0);
9         registry.register(name(prefix, "mean-get-time"));
10    }
```

Differently, we have cases in which a substitution of a program element does not imply any error. Listing 5.2 shows one of these cases and, like in the previous transformation, a statement is replaced by another. In this case, the replaced statement is an independent method call, as well as the inserted one, which is a static method call. However, the transformed program does not have the same behaviour of the original one, therefore it does not pass on the test suite of the original one. The reason is because the replaced statement plays a role on the functionality of the *time* method.

Listing 5.2: Object Substitution generating a variant.

```
1  //class com.codahale.metrics.Timer, line 101
2  public <T> T time(Callable<T> event) throws Exception {
3         final long startTime = clock.getTick();
4         try {
5             return event.call();
6         } finally {
7             //substitution
8             //update(clock.getTick() - startTime);
9             com.codahale.metrics.ThreadLocalRandom.
10            current().nextLong();
11        }
12    }
```

In some situations, a transformation can generate compilable variants and, at the same time, preserve the behaviour of the original program–*sosie*. Following, Listing 5.3 presents a sosie generated from a replacement of a literal value by another of the same type (the *string"csv-reporter"* is replaced by *"m5_rate"*), therefore not leading to compilation errors; besides, this literal did not play an important role on the program execution and its behaviour remained unchanged.

Listing 5.3: Object Substitution generating a sosie.

```
1  //class com.codahale.metrics.CsvReporter, line 135
2  private CsvReporter(MetricRegistry registry,
3                      File directory,
4                      Locale locale,
5                      TimeUnit rateUnit,
6                      TimeUnit durationUnit,
7                      Clock clock,
8                      //substitution
9                      MetricFilter filter) {
10        super(registry, /** nodeType: class
              spoon.support.reflect.code.CtLiteralImpl
11 "csv-reporter" **/
```

```
12 "m5_rate", filter, rateUnit, durationUnit);
13         this.directory = directory;
14         this.locale = locale;
15         this.clock = clock;
16     }
```

The success of a transformation depends on the kind of the targeted element. It is expected that it is not possible to modify or remove some program elements without leading to compilation errors (e.g., remove a return keyword from a method with non-void value type). On the other hand, we can easily expect that some statements that do not have any impact on the program execution, like log statements, can be removed without any further problem.

In the reminder of this chapter, we empirically study the results of applying product line variation points into real Java programs, exploring the possibilities of transforming a Java program in an SPL fashion.

5.2 Experiment

In this Section, we present in details the empirical study we conducted in order to assess the product line derivation operators in the context of Java programs.

5.2.1 Goal

The main objective of the experiment is to answer the following questions. How suited are existing product line derivation operators when used as source code transformations? Can we assess the safety of transformations with respect to the program elements?

5.2.2 Measurement Methodology

Our empirical evaluation is focused on analysing the applicability of the aforementioned operators in the elements of a Java program. We measure the percentage of non-compilable, compilable program variants and of sosies generated by a given operator applied to different program elements. We want to observe these percentages both with respect to the operators, the program elements and the pairs operator & program element (transformation). For each analysed program, our experimentation algorithm performs one transformation per time and tries to compile the transformed program, if it compiles, we proceed to run the test suite, checking whether it passes or not.

5.2.3 Experiment Variables

We define our variables according to the theory of scales of measurements; additionally, they are also classified as independent, dependent or controlled variables. Independent and controlled variables influence dependent variables, but the controlled ones remain unchanged during the entire experimentation. Table 5.2.3 presents the experiment variables with their classification and the range of values they can assume during the

Table 5.1: Experiment variables.

Name	Abbreviation	Type	Scale Type	Unit	Range
CVL Realization Operator	operator	Independent	Nominal	Text	{ObjectExistence, LinkExistence, Object-Substitution, LinkEnd-Substitution}
Program Element	element	Independent	Nominal	Text	{constructor, class, parameter, statement, etc}
Non-compilable programs	counterexamples	Dependent	Ratio	%	[0,100]
Compilable programs	compile%	Dependent	Ratio	%	[0,100]
Variants with preserved behaviour	sosie%	Dependent	Ratio	%	[0,100]
Original input program	input	Controlled	Nominal	Text	see Table 5.2

experiment. The number of non-compilable, compilable and programs with preserved behaviour is dependent on the operator and the program element[3] used in the program transformation. We perform the experiment for a controlled set of 8 input programs.

5.2.4 Hypotheses

Following, we enumerate the hypotheses we want to test against the results of our experiment, also explaining their respective motivations. They were categorized within three types, regarding: the overall safety of CVL applied to Java code (H_1), the safety of specific transformations (H_2), and the safety of transformations with respect to the types of operators and program elements (H_3).

- H_1 : It is easier to randomly produce incorrect programs than correct ones.

  Testing H_1 : is the first lead to understand that the derivation operators are prone to generate wrong products; a user without the necessary knowledge of the domain language (imitated by the random nature of the transformations) is more likely to design wrong product lines than correct ones.

- H_2a : There are transformations that will always lead to counterexamples (There exists $T = \langle O, E \rangle$ that has a counterexample percentage equal to 100%).

  If validated, H_2a can be the basis for identifying transformations to be always avoided; we can imagine for instance using these transformations as antipatterns to detect errors at design time.

- H_2b : There are transformations that will always lead to variants or sosies (There is $T = \langle O, E \rangle$ that has a variant percentage or sosies percentage of 100%).

  In the same way of H_2a, H_2b can help to collect "good" transformations and perhaps be the basis of recommendations to the designer.

- H_3a : Object Substitution is more prone to generate wrong programs than Object Existence (the counterexample percentage for transformations with $O =$

[3]The complete list of program elements can be found in the Spoon API: http://spoon.gforge.inria.fr/mvnsites/spoon-core/apidocs/index.html

Table 5.2: Descriptive statistics about our experimental data set

	#LoC	#classes	#test cases	#assert	coverage	#stmt	#transf. stmt	compile time (s)	test time (s)
JUnit	8056	170	721	1535	82%	2914	1654	4.5	14.4
EasyMock	4544	81	617	924	91%	2042	1441	4	7.8
JBehave-core	13173	188	485	1451	89%	4984	3405	5.5	22.9
Metrics	4066	56	214	312	79%	1471	319	4.7	7.7
commons-collections	23559	285	1121	5397	84%	9893	5027	7.9	22.9
commons-lang	22521	112	2359	13681	94%	11715	9748	6.3	24.6
commons-math	84282	803	3544	9559	92%	47065	12966	9.2	144.2
clojure	36615	150	NA	NA	71%	18533	12259	105.1	185

ObjectSubstitution is greater than with $O = Object\ Existence$).

An Object Substitution can be seen as a combination of two Object Existences (i.e., A ceases to exist; while B starts to exist in A's place). Therefore, we expect that randomly succeeding to create and apply a transformation depends on its complexity (in terms of number of instructions).

- H_3b : Removing program elements that are blocks of code, instead of single instructions, is likely to generate correct programs (we will consider a compile percentage greater than 70%).

  Testing H_3b : can give us initial insights on how the granularity affects the chances of succeeding on varying a Java program.

5.2.5 Subject Programs

The dataset of our experiment is composed by 8 widely-used open source projects. One important selection criterion is that they need to have a good test suite (a statement coverage greater than 70%); they are all expressed in JUnit. Table 5.2 shows the included projects and some relevant properties for the experiment.

The size of each program ranges from 1 to 80 KLOC and the number of classes from 23 to 803; they are in the category of APIs and frameworks–programs that are used by other programs. All of them have a good test coverage percentage, ranging from 79% to 94%; we consider that test suites with many assertions and high coverage indicate an important effort and care put into their design.

Table 5.2 also provides the number of statements for each program. Since the transformations manipulate statement for object substitutions, this number is an indicator of the size of the search space for it. None of the programs have a compilation time greater than 10 seconds, which helps on the total time for running the experiments. However, their testing time ranges from 7 to 144 seconds[4].

5.2.6 Protocol

The experiment is designed to randomly explore the possible transformations that can be done in a given program, having its AST nodes and the four operators as the universe

[4]CPU: Intel Xeon Processor W3540 (4 core, 2.93 GHz), RAM: 6GB

to be sampled. Algorithm 1 defines the protocol to run the experiments. It takes as input the program to be transformed and returns the data we use further to analyse the transformations and the AST elements (we get either a counterexample, a variant or a sosie for each random transformation applied). Our stopping criteria is not strict and it is defined by the amount of computational resources available in the Grid5000[5]–we seek to achieve a reasonable statistical relevance.

Data: P, a program to transform
Result: values for the dependent variables of Table 5.2.3

```
1  VP={the four kinds of VP}
2  E={elements in the AST of P}
3  while resources_available do
4  |  randomly select vp ∈ VP
5  |  randomly select compatible e ∈ E
6  |  P' ← apply T ← ⟨vp, e⟩ to P
7  |  if compile(P') = true then
8  |  |  if test(P') = true then
9  |  |  |  store P' as a sosie
10 |  |  else
11 |  |  |  store P' as a variant
12 |  |  end
13 |  else
14 |  |  store P' as a counterexample
15 |  end
16 end
```

Algorithm 1: The experimental protocol for creating and applying the transformations.

5.3 Analysis

5.3.1 Results

Table 5.3 presents the results after running the experiments for the 8 subject programs, having 196816 lines of code in total. We calculate the number of possibilities of applying a specific operator in the universe of the 8 programs (the *candidate* column). The candidates for Object Existence is simply the number of nodes in the AST; for Object Substitution is each node type of the AST squared, than the sum of them; for Link Existence is the number of fields plus the number of inheritance links; and for Link End Substitution is the number of fields squared plus the number o inheritance links squared.

The *Trial* column describes how many times we applied a transformation containing the given operator. Given the number of candidates, the number of trials, and a confidence of 99%, we calculate the margin of error for each operator. This margin holds

[5]www.grid5000.fr

meaning if one wants to consider the results as probabilities inside our universe. An example of interpretation is: the probability of having a program that compiles after removing a random program element is between 10.97% and 11.97% (compile% = 11.47 and margin of error = 0.50).

There are 86 possible combinations between operators and program elements. In Table 5.4, we show the 15 first and the 15 last transformations ordered by their compilation percentage. The first column refers to the type of operator: OE (Object Existence), OS (Object Substitution), LE (Link Existence) and LS (Link End Substitution). The second column is the affected program element. We also show the number of possibilities for each transformation and how many times we actually apply them in our experiments.

Table 5.3: Results for the operators.

	candidate	trial	%trial	margin of error	compile	compile%	sosie	sosie%
Link Existence	11248	7247	64.43	0.70	856	11.81	539	7.44
Link Substitution	14869609	85459	0.57	0.40	3851	4.51	3572	4.18
Object Existence	626258	79913	12.76	0.30	17559	21.97	6994	8.75
Object Substitution	14706362886	203566	<0.01	0.20	18776	9.22	12127	5.96
Total	14721870001	376185	<0.01	0.20	41042	10.91	23232	6.18

Figure 5.2 shows the results for 7 groups of program elements, independent on the kind of operator. Each of the vertical bars represent the compilable and sosie percentages for a given project (they are arranged in the same order of Table 5.2). We measured the dependent variables for all 42 program elements. However, to fit the results to be seen here, we have selected 20 elements and grouped them according to their common function. The first group is the *if*, containing the *if* and the *conditional* (i.e., A?B:C) program elements. The *loops* group contains the *do*, *while*, *foreach* and *for* elements. The *invocations* group is composed by the *invocation* (i.e, a method call such as .a() , where a is a method), *unary operators* and *binary operators*; we see the two last as invocations of methods (e.g., a + b is equivalent to add(a,b)). The *read* group contains the program elements in charge of access: *variable*, *field* and *array access*. The write group is composed by the *assignment* and *operator assignment* program elements. The *new* group contains the elements responsible to create objects or primitive types (the case for literal): *new array*, *new class* and *literal*. In the *exception* group, we gathered the *catch*, *try* and *throw* elements. In Table 5.3.1, we show the values for the variance, standard deviation, mean and margin of error for each of the aforementioned group of AST elements.

We excluded from Figure 5.2 program elements that have never compiled after being affected, which is, for example, the case for methods, classes, interfaces and packages.

5.3.2 Visualizing the Results

Due to the large amount of data produced as result of the experiment, we had to provide means to ease the visualization of the transformations. Figure 5.3 shows the web-based

Table 5.4: Global results for the 15 first and the 15 last transformations ordered by compilation %.

operator	AST element	candidate	trial	%trial	compile	**compile%**	sosie	sosie%
OE	AnnotationType	46	23	50.00	23	100.00	19	82.61
OE	Continue	124	31	25.00	31	100.00	9	29.03
OE	ForEach	888	330	37.16	325	98.48	47	14.24
OS	SuperAccess	60348	189	0.31	186	98.41	183	96.83
OS	ThisAccess	5790636	2282	0.04	2203	96.54	2006	87.91
OE	SuperAccess	456	86	18.86	83	96.51	24	27.91
OE	While	609	95	15.60	88	92.63	18	18.95
OE	For	3461	251	7.25	230	91.63	60	23.90
OE	Break	1008	121	12.00	110	90.91	74	61.16
OE	OperatorAssignment	1825	153	8.38	137	89.54	53	34.64
OE	If	12859	2175	16.91	1851	85.10	587	26.99
OE	Annotation	3802	903	23.75	699	77.41	655	72.54
OE	Throw	3092	523	16.91	370	70.75	150	28.68
OS	Annotation	3150678	1591	0.05	1019	64.05	980	61.60
OE	Synchronized	95	27	28.42	16	59.26	1	3.70
...	...	...	...	...	...	...	...	...
OS	Parameter	172555379	13594	0.01	12	0.09	12	0.09
OE	Method	18906	3998	21.15	3	0.08	3	0.08
OE	Parameter	28701	4494	15.66	2	0.04	1	0.02
OE	Catch	602	218	36.21	0	0.00	0	0.00
OE	Class	2477	309	12.47	0	0.00	0	0.00
OE	Enum	61	4	6.56	0	0.00	0	0.00
OE	Interface	671	52	7.75	0	0.00	0	0.00
OS	Break	409674	192	0.05	0	0.00	0	0.00
OS	Case	2035772	530	0.03	0	0.00	0	0.00
OS	Catch	51158	650	1.27	0	0.00	0	0.00
OS	Continue	4554	27	0.59	0	0.00	0	0.00
OS	Do	916	8	0.87	0	0.00	0	0.00
OS	Field	13019255	3639	0.03	0	0.00	0	0.00
OS	LocalVariable	162483228	6604	0.00	0	0.00	0	0.00
OS	Throw	2361268	1270	0.05	0	0.00	0	0.00

visualization tool we built to achieve this task. First, we provide a global view of the input program by packages (see ①), within each package we have the classes, which are represented as long rectangles with colored lines inside. It is possible to click on those rectangles to zoom in the classes (see ②). Once zoomed, it is possible to see and access each of the colored lines that make part of a class; they represent code locations (a line number) that received transformations. The red portions of the lines represent the amount of transformations in that place of the code that did not succeed to compile; while blue portions represent the ones that compiled and the green portion the ones that resulted on sosies.

Furthermore, we made possible to click on each line to visualize the list of the transformations done in a given place (see ③). This third view provides details on the actual number of transformations performed in that code location, the name of the

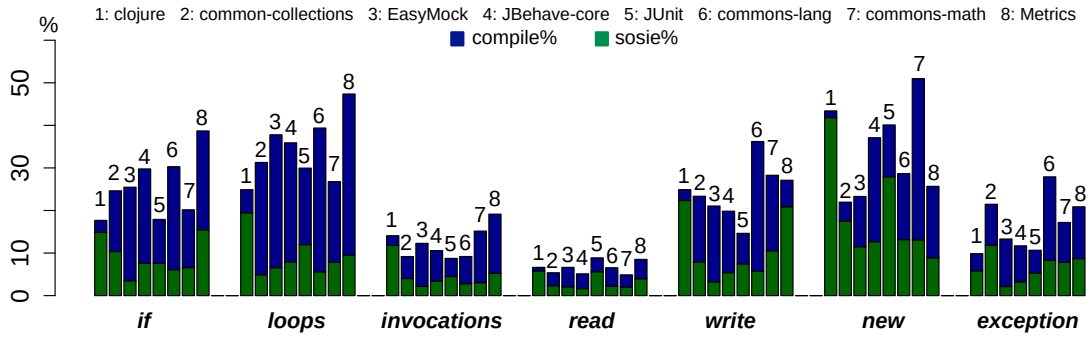


Figure 5.2: Results by categories of program elements for the 8 projects.

Table 5.5: Variance(σ^2), standard deviation(σ), mean(μ) and margin of error (ME) for the compilation% of the 7 groups of Figure 5.2.

	σ^2	σ	μ	ME
if	47.44	6.89	26.32	0.01
loop	54.90	7.41	34.16	0.02
invocation	13.31	3.65	12.29	0.00
read	2.22	1.49	6.57	0.00
write	41.50	6.44	24.42	0.01
new	111.49	10.56	33.90	0.01
exception	40.81	6.39	16.62	0.02

applied transformation and their status (0 means it compiled and passed the tests, -1 it compiled and -2 it did not compile). In the transformed code, we comment everything that was supposed to be removed/substituted in a transformation in order to let the user compare the before and after the transformation. In the specific case of Figure 5.3, the first transformation erased the first parameter of a method call and the second one replaced the *"null"* keyword by the *"unchecked"* string.

5.3.3 Hypotheses Testing and Discussion

5.3.3.1 Overall Safety of CVL to Java

With respect to H_1, we can use the data from table 5.3 to see that the chances of generating incorrect programs is much higher than of generating valid ones (see Total line). According to the experiments, around 10% of the transformed programs compiled and 6% were sosies, therefore 90% were counterexamples, which validates H_1.

Discussion. H_1 confirms that the CVL operators are not safe and do not take into account the syntax or semantics of the target language. As CVL is designed to be generic to any target language, this is acceptable and even expected; it is unfeasible to anticipate every possible domain-specific syntax or semantics rules. On the other hand, observing Figure 5.2 and Table 5.3.1, we can see that there are groups of elements that

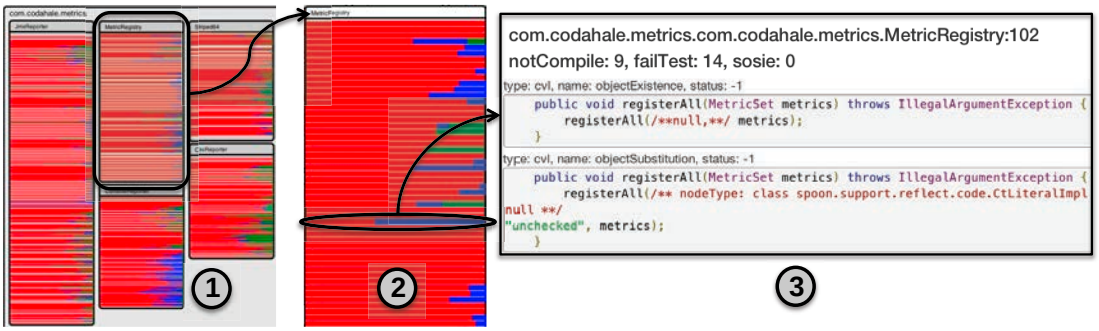


Figure 5.3: Visualizing the transformations.

are reasonably above the average compile%. For example, changing *if* blocks or *loops* has in average 26% and 34% chances to generate safe programs, respectively.

5.3.3.2 Safe and Unsafe Transformations

To test H_2a, we refer to table 5.4, specifically to its 12 last lines, in which we can find 12 transformations that have never worked, therefore validating H_2a. In the same way, we validate H_2b by observing the first 2 transformations in table 5.4 that have always generated variants or sosies.

Discussion. Some elements of a Java program have an optional nature with respect to correctness. For example, if we remove a *"continue"* from a loop it will continue syntactically correct, and maybe even semantically, since a *"continue"* can be used for optimization purposes, not implicating changes in the loop semantics. On the other hand, there are some Java constructs that can be considered as mandatory with respect to others and therefore must be handled carefully. For example, we know that a *"try"* block is often followed by a *"catch"* one, therefore removing a catch have great chances of giving compilation errors (still, there are cases in which it can work, such as when a *try* has more than one *catch*, therefore removing a catch does not lead to compilation error; however this was not the case in any of the subject programs). In the same way, replacing a *"field"* by another, will raise up errors in the rest of the code that remains using the old name and type of the *"field"*.

Despite of the validation of H_2a, we observe that the transformations that never generate correct programs are minority: 14% (12 out of 86). This indicates that there are several possibilities for varying a Java program without crashing it. Besides, we can think of combining two or more transformations so they can work together, like in the aforementioned example (remove a *catch* together with its *try*).

5.3.3.3 Safety of Transformations vs. Types of Operators and Program Elements

We refer to table 5.3 to validate H_3a. We can see that only about 9% of the transformations based on Object Substitution have succeed to generate correct programs, while

the ones based on Object Existence had about 21% of success.

In order to test H_3b, we pick the transformations that have Object Existence as their operation and blocks of code as elements: *Do, For, ForEach, While, If, Throw*. We can observe in Table 5.4 that their variant percentage range from 70%, in the case of the *Throw*, to 98%, in the case of the *ForEach*. We can also observe from Table 5.3.1 and Figure 5.2 that the mean for *loops*, which are blocks of code, is the greatest comparing to the others program elements. We only partially confirm H_3b and we discuss the reason following.

Discussion. Although the promising results for the aforementioned program elements, we can just partially confirm the idea that blocks of code are easier to vary. The reason is because not every block of code is self-contained, and therefore they refer to other blocks or are referenced in other blocks. For example, the probability to have compilable programs after removing a Class or a Method is less than 0.1%.

5.4 Discussion

5.4.1 Modeling Languages: Comparison with Chapter 3

Our previous effort provides evidence that the usage and tooling support of CVL should be specialized for a given domain-specific modeling language [FBLNJ12, FBA+13] if one wants to achieve safe product derivation.

The experiment presented in this chapter provides further evidence that the *direct* use of CVL leads to improper support – this time we consider a programming language (not a modeling language). Our empirical study confirms that the syntactic structure and semantics of the targeted language (here Java) should be taken into account. Otherwise, developers will spend their time specifying variability over language elements that lead to unsafe variants. In Chapter 3, we have identified a tendency to generate incorrect product variants for three modeling languages. The evidences pointed that the more complex is a language (quantity of syntactic and semantic rules), the easier is to synthesise wrong product variants. The results in Chapter 3 shows that: in a small modeling language such as the Finite State Machine, the percentage of incorrect variants is around 16%; in the Ecore language it jumps to 25%; at last, in a very big modeling language like UML, the incorrect variants represent 66% of the total. We confirmed this tenet for Java, which raised the percentage of incorrect variants to 90%.

For applying the approach and fully exploring the variation space of a language (Java), we had to develop novel automated techniques. The techniques exposed in Chapter 3 do not seek to categorize which model elements are likely to produce unsafe variants. We also develop a large-scale infrastructure to launch 86 kinds of transformations while checking some properties of the variants. Such an infrastructure was required due to the computational resources involved in the experiment.

5.4.2 Diversity: Comparison with [BAM14]

Comparing to the experiment in [BAM14], our approach to synthesize sosies is clearly different. In [BAM14], program transformations are built based on domain knowledge and empirical observations to decide which transformation can be used. In this chapter, we use an automatic, agnostic method to explore the variability space. We use four CVL realization operators on any kind of Java AST node types. The approach has the merit to discover new kinds of transformation (86 against 9 in [BAM14]) that have not been used in [BAM14]. We empirically show that a substantial number can also produce sosies. In particular, the *Link Existence* realization operator has not been used in [BAM14]. Another example is the *Object Existence* realization operator used on specific Java AST node type, – *e.g.* the call to the super operation – that allows to discover numerous sosies. As a result, we discover that (i) some CVL transformations have a good percentage of sosies and can compete with program transformations introduced in [BAM14] (ii) as recognized by the authors of [BAM14], CVL transformations are sometimes unintuitive for a human and surprisingly work. The removal of a parameter in methods is an example. This result demonstrates the benefits of the use of full automated techniques that do not make any assumption and that visit the whole problem space; (iii) finally, some CVL transformations could be adapted (leading to the creation of new ones) in the context of *diversification*.

5.4.3 Towards a Methodology and Systematic Approach

We chose Java as it is a widely used programming language, however our experiments can be reproduced to analyse other languages. Following we enumerate the essential steps to reproduce the approach, learnt from the application of the experiment in Java.

1. Perform random executions of the transformations over the constructs of the language in a set of examples;
2. Collect and categorize the generated variants in succeeded and not succeeded, together with the used transformations;
3. Quantitatively analyse the best and the worst transformations with respect to a criteria;
4. Qualitatively analyse subsets of transformed programs by domain expert. Looking into transformed programs/models can reveal patterns of errors (a visualization tool for the examples in the target language is helpful).

After acquiring all the data and performing analysis, an immediate usage is to serve as basis to enhance the transformations and create more robust ones. In Chapter **??**, this enhancement was done having as basis only the domain expert knowledge. We believe that our methodology could better guide customizations by showing real examples of Java program variants. Second, we can envision practical applications, such as heuristics to enhance design of the mapping relationships between features and program elements. –ranging from syntax highlighters to recommender systems.

5.4.4 Threats to Validity

Our experiment has the necessary conditions for *causality*. No other changes are done in the programs by each iteration, we only perform one transformation at a time (see 1), therefore the changes in our dependent variables are only related to the execution of the given transformation.

Internal Validity: One potential threat for *internal validity* is the number of trials with respect to the possible number of transformations in our universe. We have addressed this threat by controlling the margin of error for the transformations, having it always less than 1% and using a confidence level of 99%. However, some program elements were not numerous enough for being representative, such as *annotation types*; they are rarely used and therefore our experiments may not be conclusive for these program elements.

External Validity: Regarding the threats to external validity, we relied on the fact that our subject programs are widely used by programmers and that they make part of lots of projects that use them as APIs and frameworks. Besides, they have a considerable amount of lines of code (200K in total). Regarding the representativeness of the transformations, in total, we generated and executed 376,185 transformations. We tried to compute the maximum number of transformations as possible, given the available resources; if we multiply the total number of transformations we have performed by, the compilation time added to the testing time (when compilation is ok) of the 8 projects, we have a total of around 97 days of computation in a personal computer.

By calculating the variance and standard deviation of transforming specific program elements over our 8 different subject programs, we could notice some discrepancy among them. This fact can be an evidence that factors such as choices of design and programming style, can have an influence on the compilation percentages; it needs to be further explored.

5.5 Conclusions

We executed 376,185 variability transformations over 8 real large Java programs with up to 85,000 lines of code. The transformations consist in adding, removing, or replacing all kinds of Java elements (classes, loops, fields, assignments, etc.) – they respresent the main variation points in CVL realization layer. We obtained 376,185 variants of Java programs, out of which 41,042 compile and 23,232 pass the test suite. We could obtain a panorama of the suitability of CVL derivation operators applied to such a complex language like Java. The results were as expected: CVL operators are dangerous to be applied in fine-grained programing language constructs; calling for specialized semantics or more sophisticated (de)composition languages. However, we could also verify that there are categories of constructs that can better behave with variational transformations, like blocks of codes.

Chapter 6

Towards a Methodology

In this chapter, we provide a methodology to integrate our work into any organization that seeks to implement model-based software product lines. We focus on the response to the event of having to engineer an MSPL for a given DSL, showing the different activities, how they follow each other and the roles involved in each of them. We model the set of processes using the Business Process Modeling Notation (BPMN) [OMG06], a widely used and standardized language for this purpose. Instead of explaining the technical details of the approach like in previous chapters, we rather provide high-level tasks for the purpose of situating our approach into the development process.

6.1 Roles

Although the construction of an MSPL can involve several stakeholders, there are two roles particularly interested in our methodology, and we will further concentrate on the activities related to them. The two fundamental roles in our methodology are: the MSPL Designer and the MSPL Infrastructure Engineer. Both are directly related to our counterexamples generation approach described in Chapter 3 and are essential to leverage variability management in a model-based context.

The MSPL Designer is the role responsible for constructing the models of an MSPL. This role requires a solid knowledge about the domain, as designing the MSPL models is essentially deciding which and how products can vary. Therefore, the MSPL Designer can be thought as a domain engineer with considerable knowledge of the product engineering phase (required to foresee the outcome of possible products when designing the realization mappings).

The MSPL Infrastructure Engineer is the role responsible for building, extending and maintaining the mechanisms and tools that support the activities of the MSPL designer role. This includes the construction of model editors, interpreters, derivation engines and verification & validation techniques to increase the safety of the designed MSPLs (we concentrate on these last in this chapter). Besides the knowledge of the domain, infrastructure engineers are also required to master model-driven technologies and their workbenches.

6.2 Activities

In this section, we present the main activities of MSPL engineering that relate to our generative approach of counterexamples. The goal of these activities is to leverage product line engineering for a given DSL, and they concern both roles before mentioned. As illustrated in Figure 6.1, it starts with a request inside the organization to build an MSPL for a given DSL. After, the engineers can carry on with three main parallel tasks: the *MSPL modeling*, the *Engineering of verification mechanisms* and the *generation and organization of counterexamples*; they are explained in the next subsections together with the shared activity *Consult counterexamples*.

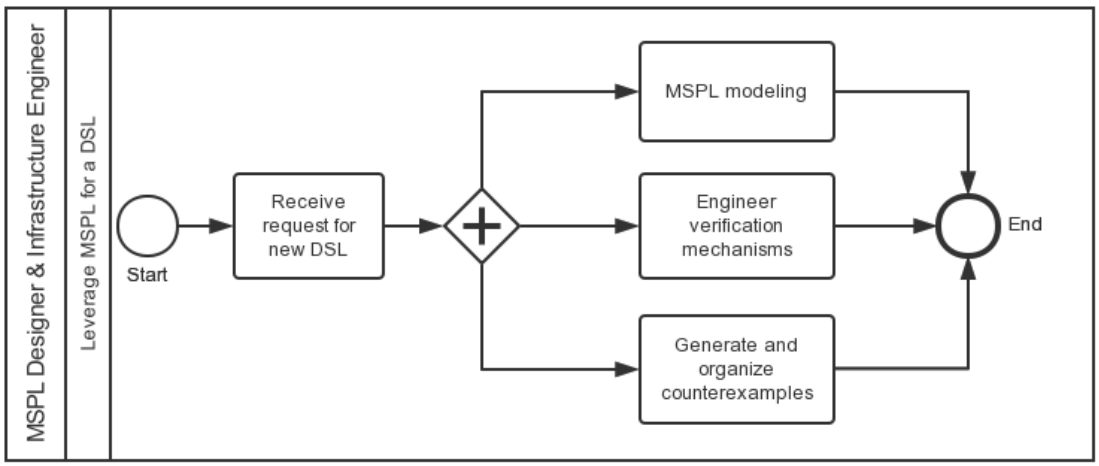


Figure 6.1: Leverage MSPL for a DSL

6.2.1 Generate and organize counterexamples

This activity is the basis of our methodology, it provides as outcome the material to help designers and infrastructure engineers of MSPL with their activities of building safer models and tools. The counterexamples generation part is explained in details in Chapter 3; it feeds the counterexamples database, as shown in Figure 6.2.

After generating the counterexamples, we can organize them to be better exploited. They can be clustered and categorized to represent groups of errors and then sorted according to their safety or other criteria. For example, in Table 5.4 of Chapter 5, we ordered the types of counterexamples (in that case, the pairs of variation point + program element) by their percentage of compilation; while in Figure 5.2 of the same chapter, we categorize the counterexamples by type of program elements.

6.2.2 Consult counterexamples

Before explaining the two other main activities, we present the *Consult counterexamples*, which is an important task shared by both. Evidently, it needs to be performed after

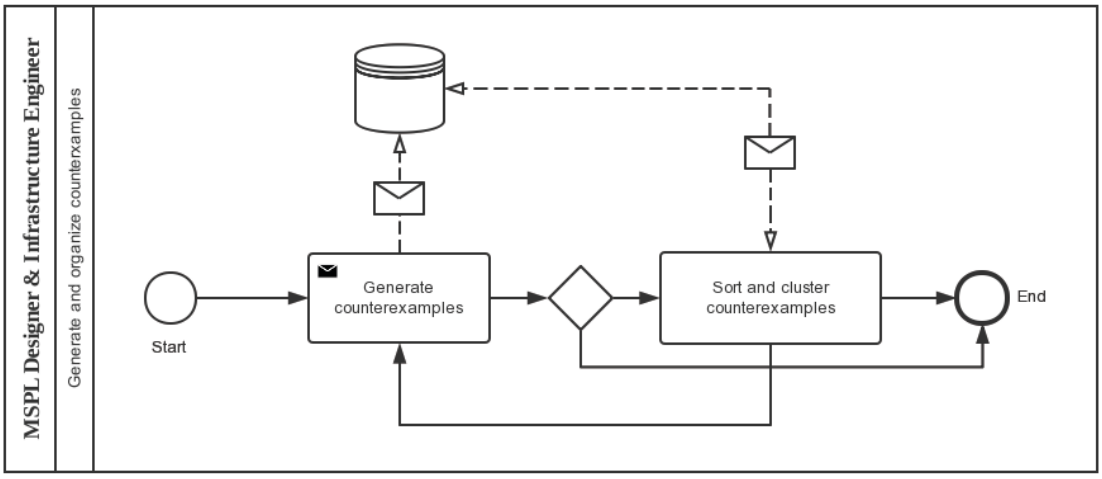


Figure 6.2: Generate and organize counterexamples

the previous activity of generating counterexamples. There are several possibilities to exploit the generated counterexamples; we concentrate on four basic sub activities that can serve both to the MSPL Designer and the Infrastructure Engineer, as shown in Figure 6.3. The *check unsafe assets* activity consists on evaluating whether a base model element is intrinsically dangerous to be modified or not; in Chapter 5, we have identified Java constructs that were very likely to lead to errors when modified (e.g., read statements). Incrementally, one can also access the safety of the assets associated with a variation point (*check unsafe VP + assets* activity) and together with a configuration of the variability model (*check unsafe configuration + VP + assets*).

The before mentioned activities rely mainly on comparing the counterexamples with a candidate MSPL design, in an approximated way, like comparing types/categories of errors. Whereas in the *Match existing counterexamples* activity, the goal is to actually use counterexamples as testing models, this is particularly useful in smaller problem spaces. These activities can serve as basis for more sophisticated ones, like code recommendation and static analysis of artifacts.

6.2.3 MSPL modeling

Modeling an MSPL using CVL is essentially constructing its three main models: Base, Variability and Realization models. In Figure 6.4, we illustrate these three main activities, preceded by the user task *acquire available models* and, depending on the existing models, the other construction activities are performed or just passed. The MSPL Designer is the role responsible for handling these activities. The three model construction activities of Figure 6.4 can be expanded in sub tasks; in the next section, we present the *Construct realization model* expanded, as it is the main activity that can benefit from our approach of counterexamples generation.

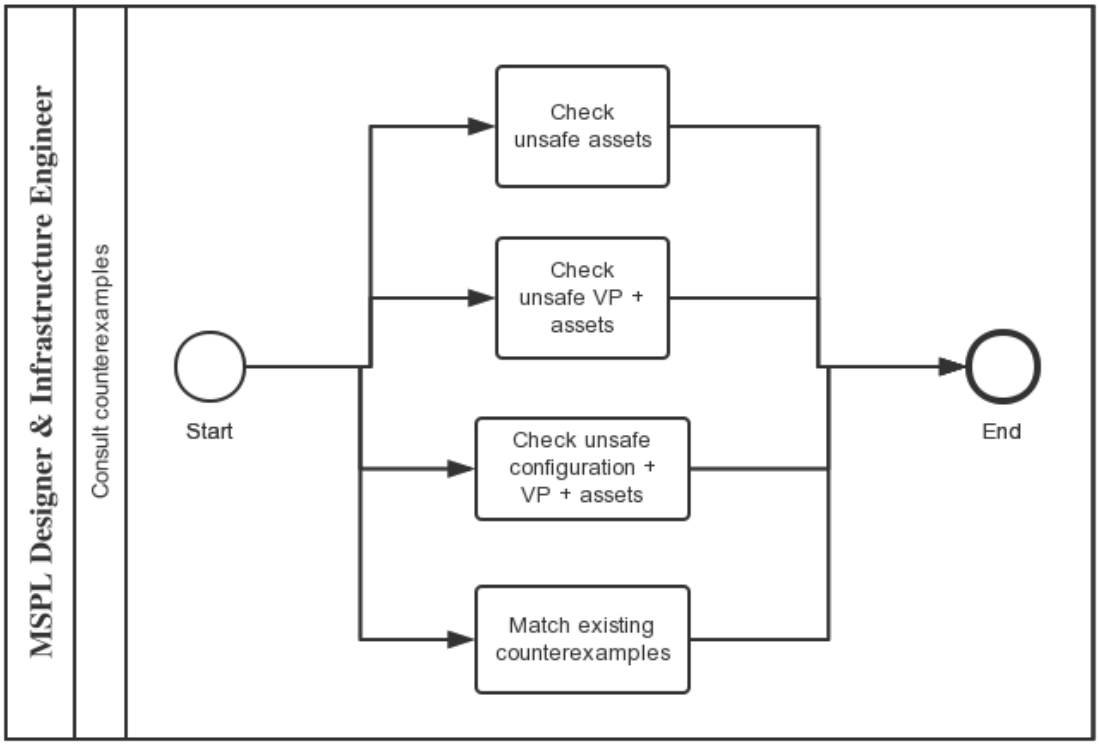


Figure 6.3: Consulting counterexamples

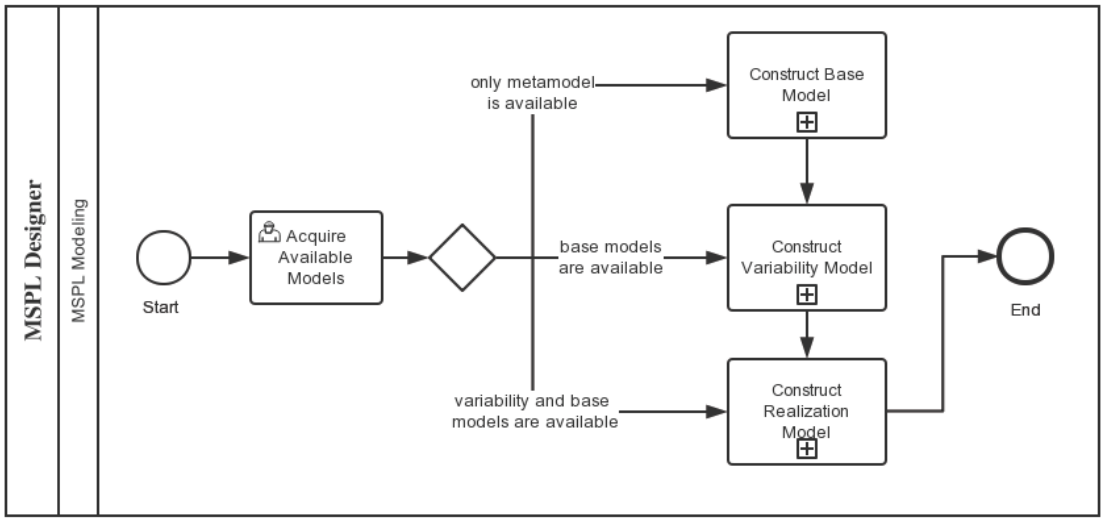


Figure 6.4: Activities to design an MSPL

Construct realization model

The realization model is the set of mappings between the features in the variability model and the elements of the base models; therefore, it assumes that these models exist

and can be referenced. Figure 6.5 illustrates the modeling process of a realization model. Before choosing the mapping relationship itself, the MSPL Designer has to choose the concerned feature (or set of features, or a feature expression). After, the designer has to identify the assets (base model elements) that correspond to the chosen feature and then to pick the desired variation point to link them. In parallel, or right after these three steps: *Choose feature(s), Choose corresponding asset(s) and Choose variation point(s)*, the MSPL Designer can consult counterexamples to assess the safety of the candidate MSPL design. If he/she considers the design to be safe – automatically or manually checking if the candidate design is similar or equal to those of some counterexamples – the process is repeated for other mappings or finished; else, the model must be corrected.

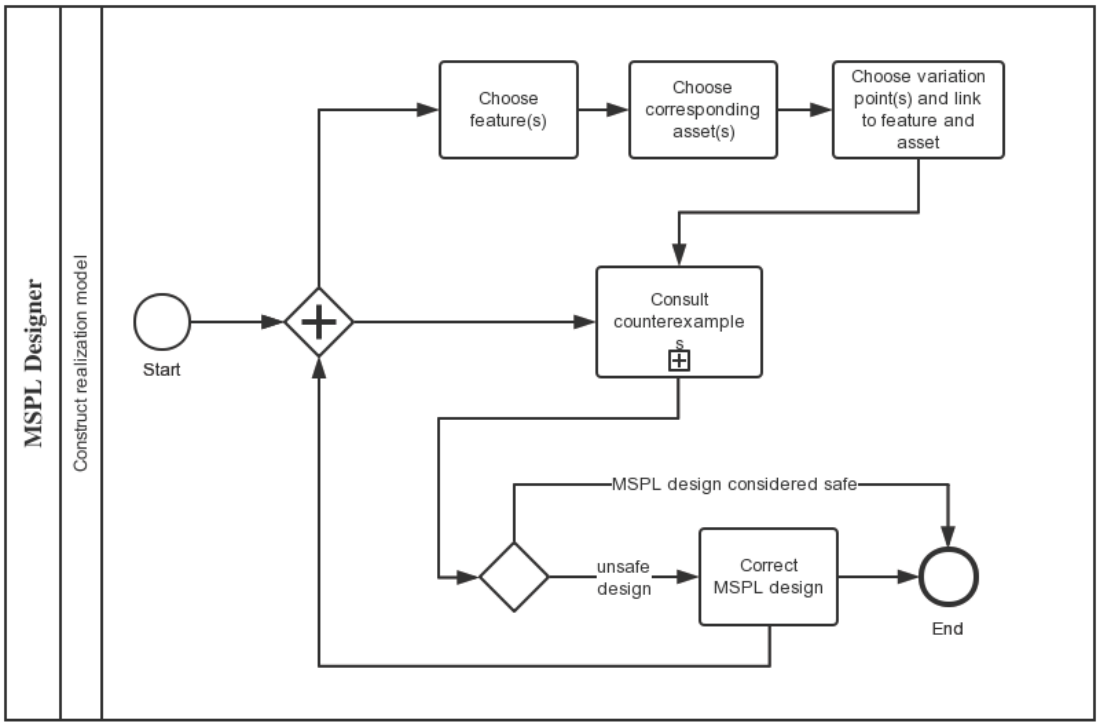


Figure 6.5: Activities to construct the realization model

6.2.4 Engineer verification mechanisms

At a different level of the MSPL Designer, the MSPL Infrastructure Engineer has to deal with tasks that will support the modeling activities previously presented. We choose to present the activity of engineering verification mechanisms (see Figure 6.6), in which we consider that our counterexamples approach can be useful. The infrastructure engineer can decide either to *construct a domain specific checker* or to *customize the derivation engine*, or both, as they are not mutually exclusive; we present next these two possible activities expanded.

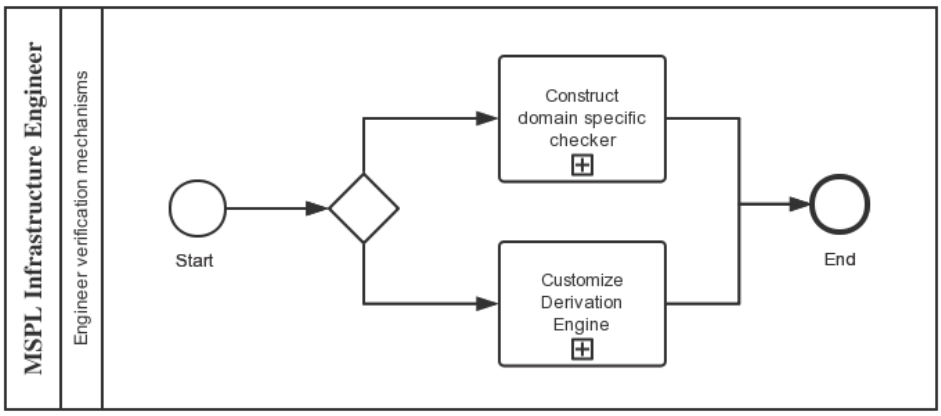


Figure 6.6: Infrastructure mechanisms for MSPL verification

6.2.4.1 Construct domain specific checker

Often, product line engineers seek to have checking mechanisms to verify their design choices at design time, avoiding in first-hand bad designs. We can imagine mechanisms similar to syntax highlighting in programing languages environments to reveal these errors, before derivation (compilation is the analogy). However, they are often limited to syntax checking, as it is easier to be statically checked using type checking mechanisms.

This activity is illustrated in Figure 6.7 and starts with consulting the counterexamples in order to analyse the unsafe designs that can serve as antipattern. After, the engineer can encode verification rules in a chosen formalism to detect automatically the unsafe designs. For example, in Section 3.3.4 of Chapter 3, we encoded two rules for the Finite-State Machine domain. Finally, the engineer can choose to incorporate these rules in a checking mechanism, deciding whether to use them at design time or after, incrementally or at once.

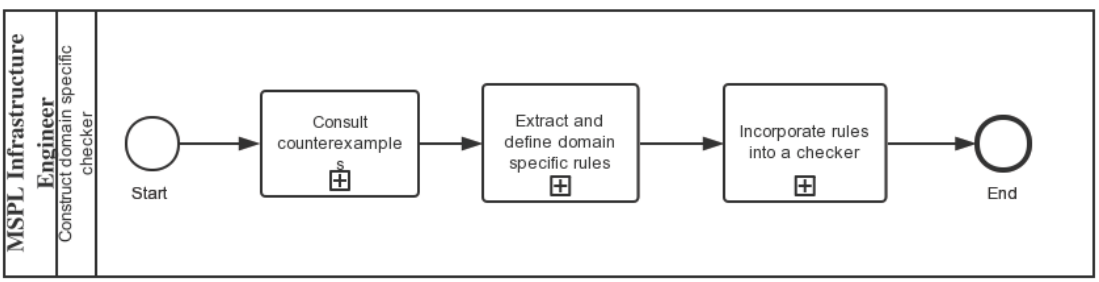


Figure 6.7: Activities to construct a checking mechanism for a MSPL

6.2.4.2 Customize derivation engine

The infrastructure engineer can decide, based on the counterexamples, that the derivation engine must be specialized for a given domain. This case is illustrated in Figure 6.8 and it is detailed in Chapter 4. Like in the construction of a checker, the first task is to consult the counterexamples and then extract the domain specific rules, however, the engineer does not necessarily need to encode the rules in a formalism. The rules will serve as basis to specialize the existing semantics of the variation points. In the situations that variation points resulted in counterexamples, the engineer has to think how the semantics could be extended to avoid the errors. For example, we showed in Section 4.2.1.2 of Chapter 4 customizations for the Object Existence variation point that could reduce errors due to dangling references.

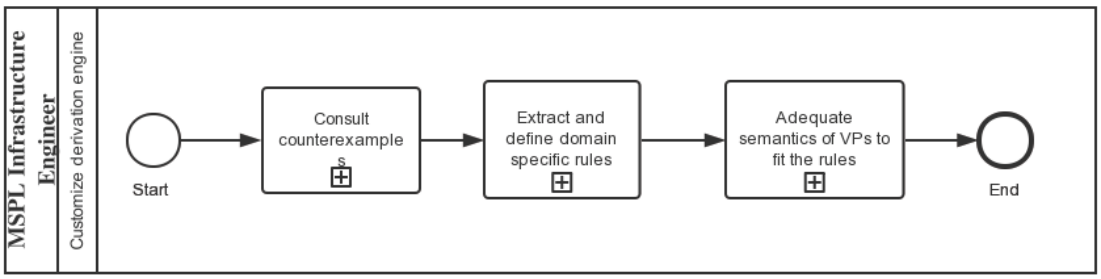


Figure 6.8: Activities to customize the derivation engine

6.3 Conclusion

In this chapter, we have presented a methodology to leverage MSPL engineering for a given DSL, defining the roles and activities involved using BPMN. The methodology serves to guide the organizations that want to use our approach to support their activities. Our counterexamples generation approach can be integrated as a parallel activity to both the MSPL modeling and its infrastructure engineering, emphasizing its non-intrusive nature.

Part III

Conclusion and Perspectives

Chapter 7

Conclusion and Perspectives

In this chapter, we first synthesize and conclude all the contributions of this thesis, re-enumerating the challenges and how we addressed each of them. Next and finally, we discuss some perspectives for future research.

7.1 Conclusion

This thesis is an effort to support the tasks of managing variability in systems engineering. This management needs special assistance due to the complexity of the development process and of all the machinery involved. Indeed, we showed that, developing a system in Thales (our case organization that uses systems engineering) is a complex task due to two main characteristics: their diversity of domain specific languages and their existing model-based software development life cycle.

We concluded that leveraging variability management in this context raises five big challenges that need to be addressed:

1. provide early support for constructing product lines for new domain specific languages;
2. provide specialized support for product derivation based on each particular domain;
3. provide separation of concerns at both modeling and V&V levels;
4. integrate the modeling and verification approaches in a seamless and non-intrusive way into the existing systems development process;
5. facilitate the consistent generation of artifacts.

From the five challenges identified, we considered those which the scientific community had less advanced in the state of the art and focused on them. The ones more studied could be contemplated in our choice to use the Common Variability Language. Nevertheless, we found limitations on CVL and we concentrated on overcoming them,

by addressing the issues on realizing variability and the need for customized assistance in the realization layer.

We introduced and explored the concept of counterexamples of model-based software product lines, having it as basis for engineering better derivation engines and verification mechanisms. We automated the generation of counterexamples in a systematic and domain-independent approach, so that we could synthesize MSPLs starting only from a metamodel of the domain – the fundamental characteristic of our approach in order to provide early support.

We then validated our approach with four different modeling languages, being one of them acquired from a real industry scenario. The approach could generate counterexamples in a reasonable time, both when only the metamodel of the language was available, as well as when we could use existing variability or base models. After, we gave first insights on how to customize the derivation engine based on the counterexamples knowledge and how one could weave new semantics into the derivation process.

Using the same idea of the counterexamples generation, we extrapolated our experiments to the vast scenario of Java programs. The experiments aimed to analyse how adequate was the use of CVL in a complex programming language like Java. We concentrated on the adequacy of the derivation operators of CVL in fine-grained program elements. Following the empirical method, we could systematically asses the safety of product-line-based code transformations. This experiment also served to demonstrate the first steps after having generated counterexamples: analysing, ordering and categorizing them.

Finally, we synthesized our contributions in the form of a methodology, defining the roles and the high-level activities for an organization that wants to benefit from our approach.

We conclude that our approach contemplates the five challenges raised in the beginning:

1. we provided early support by enabling the approach to work in the case of non-existing models (e.g., initial MSPLs can be generated even before domain engineering phases, before the variability model definition);

2. we provided mechanisms to customize the CVL derivation engines, adapting the operational semantics of its variation points;

3. we used CVL and implemented tooling support following its concepts, providing an orthogonal and modular way to express variability on top of multiple modeling languages;

4. our methodology showed that the approach can be used in parallel to the other activities of the development process – it does not require the engineer to learn and use any new notation – and there is no need to change it, avoiding intrusive mechanisms (e.g., other approaches require augmenting the current notations with variability information);

5. we showed that counterexamples can reveal design errors that can be used to reduce the possibility of generating unsafe products; they can be the basis for building safer derivation engines and checking mechanisms for MSPLs.

7.2 Perspectives

In this section, we present some long- and short-term ideas for research around the contributions of this thesis.

Advanced use of counterexamples

This thesis strongly advanced on the counterexamples identification and generation; we also showed initial ways to exploit these valuable artifacts. However, we believe that the counterexamples have an enormous potential still to be explored. We envision that they can feed more sophisticated mechanisms, like Recommender Systems [RV97], Expert Systems [GR98] or Machine Learning approaches that identify, correct or forecast possible antipatterns.

Intelligent IDEs for MSPL Engineering

Concretely, these advanced techniques could be integrated into more Intelligent IDEs for constructing software and systems product lines. Nevertheless, there is still a lot to improve on product line adoption, and one of the reasons is because variability is not an evident concern in today's IDEs. Some works, based on feature-oriented software development, try to address this issue, by bringing the preoccupation of features into the code. However, we are still far from integrated development environments that can handle the full product line life cycle. The ideal IDE should facilitate the seamless integration among variability models and any other artifacts of the organization (textual requirements, code, models, etc.), in a way that **a valid configuration would always lead to a safe combination of any artifacts** – *The Safe Derivation of Anything*.

To achieve *The Safe Derivation of Anything*, variability information must be correctly linked to the artifacts, and the semantics of these links should be fitted to each kind of artifact (so far, what we have been advocating in this thesis). We do not believe that an universal product line language, with a common algebra for combining assets, would do the job; new languages and new kinds of artifacts are always being incorporated into complex systems development. We believe that our counterexamples generation approach is a first step to overcome this, opening avenues to even more automated approaches than the one presented in this thesis.

When designing an SPL in this IDE, the engineer would be helped on the definition of features and realization links. For example, if existing source code or models were available, recommenders would suggest features corresponding to parts of code or model elements responsible to a functionality of the system. The link would be made automatically, considering all the other artifacts that are related to that feature and the way they are related (semantics of the relationship). When configuring a product, the

concrete syntax of the variability model would already prohibit feature combinations that would lead to incorrect products, or maybe mutate the links so they could exclude mutually exclusive parts or include absent dependent parts (e.g., two classes that could not be at the same product, because of a mutually exclusive pair of methods, could select a method and erase or comment the other one).

The IDE would capture the *intention* of the engineer. If he/she defines a feature called "Log System", the realization layer would suggest automatically to link to, for example, the code statements using methods/classes from the *java.util.logging*, but still considering the impacts of removing, replacing or changing the different statements, based on the gained knowledge from the counterexamples. This could be true for any language incorporated in the IDE, as the counterexamples approach would work independently and agnostic to the domain.

Self-tuning of product line checkers

Many verification mechanisms of SPL's models are based on type checking. With a Type System, they have a set of rules that constrain the possible correct model designs, detecting defects that makes a model ill-typed. However, designing such a type system for the triplet (Variability Model, Realization Model, Base Models) is a very hard task. Therefore, a promising vision is to automate the construction of these rules. From the generated counterexamples, we can extract valuable knowledge about common errors and synthesize new rules to be incorporated into checkers – **a self-tuning of checking mechanisms**. An initial process to do such a self-tuning system is to take the triplet elements that always resulted on counterexamples and synthesize preventive rules (e.g., **if in** (Variability Model, Realization Model, Base Models) **contains** (feature A, object existence, mandatory model element E) **then return** error).

Narrowing down the search space

A short-term perspective is to narrow down the search space, decreasing the time to find counterexamples and, perhaps, finding more meaningful ones. Search-based algorithms can be of great value if one can define good fitness functions for finding and generating counterexamples; they already proved value on software test data generation [McM04]. Another possibility is to limit the search on the possible configurations of the feature model; t-wise algorithms have been developed and used to bypass the combinatorial explosion imposed by variability models [PSK+10, JHF12]. Therefore, our idea is to incorporate a t-wise coverage criteria into our algorithm.

Integration with model slicers

Another short-term perspective is to see and implement the realization model as a model slicer, which is a mechanism to extract a subset of a model [BCBB11]; this could be particularly helpful in a negative derivation scenario. The advantage would be to reuse the slicing techniques to better depict model elements from the base model; for

example, when selecting a class in a model, a slicer is able to return all the other classes associated to the selected one.

Bibliography

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert 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.

[AJTK09] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. Model superimposition in software product lines. In *ICMT'09*, pages 4–19, 2009.

[AK03] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July/August 2009.

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[ALHM+11] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. Supporting consistency checking between features and software product line use scenarios. In Klaus Schmid, editor, *ICSR*, volume 6727 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2011.

[ARW+13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grosslinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, Piscataway, NJ, USA, 2013. IEEE Press.

[AtBGF11] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. Formal description of variability in product families. In Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid, editors, *SPLC*, pages 130–139. IEEE, 2011.

[BAM14] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA'14)*. ACM, 2014.

[BCBB11] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *Model Driven Engineering Languages and Systems*, pages 62–76. Springer, 2011.

[BGdPL+03] Felix Bachmann, Michael Goedicke, Julio Cesar Sampaio do Prado Leite, Robert L. Nord, Klaus Pohl, Balasubramaniam Ramesh, and Alexander Vilbig. A meta-model for representing variability in product family development. In van der Linden [vdL04], pages 66–80.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013.

[BS12] Thomas Buchmann and Felix Schwägerl. Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, pages 37–44. ACM, 2012.

[BSRc10] David Benavides, Sergio Segura, and Antonio Ruiz-cort. Automated Analysis of Feature Models 20 Years Later : A Literature Review. *Information Systems*, 35(6), 2010.

[Bur93] Neil Burkhard. Reuse-Driven Software Processes Guidebook. Version 02.00.03. Technical Report SPC-92019, Software Productivity Consortium, November 1993.

[CA05a] Krzysztof Czarnecki and Micha Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, volume 3676 of *LNCS*, pages 422–437, 2005.

[CA05b] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glck and Michael Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer Berlin / Heidelberg, 2005. 10.1007/11561347:28.

[CA11a] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53(4):344–362, April 2011.

[CA11b] Lianping Chen and Muhammad Ali Babar. Variability management in software product lines: an investigation of contemporary industrial challenges. *SPL: Going Beyond*, 2011.

[CAA09] Lianping Chen, M. Ali Babar, and Nour Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90. Carnegie Mellon University, 2009.

[CAK+05] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof 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.

[CBA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines: a systematic review. In *SPLC'09*, pages 81–90, 2009.

[CBS12] Juan José Cadavid, Benoit Baudry, and Houari A. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *ICST*, pages 131–140, 2012.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 156–172, London, UK, 2002. Springer-Verlag.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CFW90] G. H. Campbell, S. R. Faulk, and D. M. Weiss. Introduction to synthesis. Technical report, June 1990.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In Robert L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

[CHS10a] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In *Proceedings of the 9th GPCE'10 conference*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[CHS+10b] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*, pages 335–344. ACM, 2010.

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[CL00] Curtis Clifton and Gary T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145, 2000.

[CN01] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[COM09] EMBEDDED COMPUTING. Cyber-physical systems. 2009.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06*, pages 211–220. ACM, 2006.

[DD11] Mohammed El Dammagh and Olga De Troyer. Feature Modeling Tools: Evaluation and Lessons Learned. *Advances in Conceptual Modeling.*, pages 120–129, 2011.

[DGRN10] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010.

[DRGN07] Deepak Dhungana, Rick Rabiser, Paul Grünbacher, and Thomas Neumayer. Integrated tool support for software product line engineering. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 533–534. ACM, 2007.

[DSF07] Olfa Djebbi, Camille Salinesi, and Gauthier Fanmuy. Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues. *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 301–306, October 2007.

[ES13] Holger Eichelberger and Klaus Schmid. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference*, pages 12–21. ACM, 2013.

[EW11] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, December 2011.

[FAM] FAMILIAR: FeAture Model scrIpt Language for manIpulation and Automatic Reasonning. `http://nyx.unice.fr/projects/familiar/`.

[FBA+13] Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Benoit Baudry, and Jérôme Le Noir. Generating counterexamples of model-based software product lines: an exploratory study. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 72–81, New York, NY, USA, 2013. ACM.

[FBA+14] Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Jérôme Le Noir, Axel Legay, and Benoit Baudry. Generating counterexamples of model-based software product lines. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2014.

[FBBLN12] Joao Bosco Ferreira Filho, O. Barais, B. Baudry, and J. Le Noir. Leveraging variability modeling for multi-dimensional model-driven software product lines. In *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pages 5–8, 2012.

[FBFG08] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69073-3_2.

[FBLNJ12] Joao Bosco Ferreira Filho, Olivier Barais, Jérôme Le Noir, and Jean-Marc 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.

[FC22] H. Ford and S. Crowther. *My Life and Work*. Library of American civilization. Doubleday, Page, 1922.

[FFB02] Dániel Fey, Róbert Fajta, and András Boros. Feature modeling: A metamodel to enhance usability and usefulness. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 198–216, London, UK, UK, 2002. Springer-Verlag.

[FHMP+11a] F. Fleurey, Ø. Haugen, B. Møller-Pedersen, A. Svendsen, and X. Zhang. Standardizing Variability - Challenges and Solutions. In *SDL Forum*, pages 233–246, 2011.

[FHMP+11b] Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, Andreas Svendsen, and Xiaorui Zhang. Standardizing variability - challenges and solutions. In Iulian Ober and Ileana Ober, editors, *SDL Forum*, volume 7083 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 2011.

[FMP08] Thomas Forster, Dirk Muthig, and Daniel Pech. Understanding Decision Models – Visualization and Complexity reduction of Software Variability. In *Proceedings of the 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, January 2008.

[GFA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA, 1998. IEEE Computer Society.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Gom06] Hassan Gomaa. Designing software product lines with uml 2.0: From use cases to pattern-based software architectures. In Maurizio Morisio, editor, *ICSR*, volume 4039 of *Lecture Notes in Computer Science*, page 440. Springer, 2006.

[GR98] Joseph C Giarratano and Gary Riley. *Expert systems*. PWS Publishing Co., 1998.

[Gro07] Object M. Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Technical report, November 2007.

[GS02] Hassan Gomaa and Michael Eonsuk Shin. Multiple-view meta-modeling of software product lines. In *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, ICECCS '02, pages 238–, Washington, DC, USA, 2002. IEEE Computer Society.

[HC10] Arnaud Hubaux and Andreas Classen. A preliminary review on the application of feature diagrams in practice. *Proceeding of VaMoS*, (1):53–59, 2010.

[HMlPO+08] Ø ystein Haugen, Birger Mø ller Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. *2008 12th International Software Product Line Conference*, pages 139–148, September 2008.

[HPS08] Günter Halmans, Klaus Pohl, and Ernst Sikora. Documenting application-specific adaptations in software product line engineering. In

Proceedings of the 20th international conference on Advanced Information Systems Engineering, CAiSE '08, pages 109–123, Berlin, Heidelberg, 2008. Springer-Verlag.

[HSS+10] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza amd Ana Moreira, and Awais 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.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[IKP11] Paul Istoan, Jacques Klein, and Gilles Perouin. A Metamodel-based Classification of Variability Modeling Approaches. *VARY 2011 workshop @ MODELS Conference*, 2011.

[ISO10] ISO. International organization for standardization: Iso/iec fcd 42010: Systems and software engineering - architecture description, June 2010.

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC'12*, pages 46–55, 2012.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 311–320, New York, NY, USA, 2008. ACM.

[KAO11] Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 5. ACM, 2011.

[KC07] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. *Engineering*, 2(EBSE 2007-001), 2007.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-

specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, January 1998.

[KLD02] K C Kang, J Lee, and P Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *ICSE*, pages 611–614. IEEE, 2009.

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[LC13] Miguel A Laguna and Yania Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, 78(8):1010–1034, 2013.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[MFB+08] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model driven engineering languages and systems*, pages 782–796. Springer, 2008.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[MOF02] OMG MOF. Omg meta object facility (mof) specification v1. 4, 2002.

[MTS+14] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. An overview on analysis tools for software product lines. *(to appear)*, 2014.

[NK08] Natsuko Noda and Tomoji Kishi. Aspect-oriented modeling for variability management. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 213–222. IEEE, 2008.

[OMG06] Business Process Modeling Notation OMG. Version 1.0. *OMG Final Adopted Specification, Object Management Group*, 2006.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[PKGJ08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC'08*, pages 339–348. IEEE, 2008.

[PM06] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 1049–1050, New York, NY, USA, 2006. ACM.

[PNP06] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA, 2006.

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 459–468. IEEE, 2010.

[RFS08] Fabricia Roos-Frantz and Sergio Segura. Automated analysis of orthogonal variability models. a first step. In Steffen Thiel and Klaus Pohl, editors, *SPLC (2)*, pages 243–248. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[RGD10] Rick Rabiser, Paul Grünbacher, and Deepak Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, 52(3):324–346, March 2010.

[ROR11] Rick Rabiser, Pádraig O'Leary, and Ita Richardson. Key activities for product derivation in software product lines. *Journal of Systems and Software*, 84(2):285–300, February 2011.

[RV97] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[S+00] Richard Soley et al. Model driven architecture. *OMG white paper*, 308:308, 2000.

[SBDT10] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico 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.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[Sch01] Johann M Schumann. *Automated theorem proving in software engineering*. Springer, 2001.

[Sch06] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.

[SHMP11] Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. Specifying a testing oracle for train stations - going beyond with product line technology. In Jörg Kienzle, editor, *MoDELS Workshops*, volume 7167 of *LNCS*, pages 187–201. Springer, 2011.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51:456–479, February 2007.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Sci. Comput. Program.*, 53:259–284, December 2004.

[SR09] Andrew P Sage and William B Rouse. *Handbook of systems engineering and management*. John Wiley & Sons, 2009.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[SZLT+10] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and Gøran K. Olsen. Developing a software product line for train control: A case study of cvl. In Jan Bosch and Jaejoon Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2010.

[TAK+12] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product

lines. *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*, 2012.

[TAK+14a] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):6, 2014.

[TAK+14b] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. In *Computing Surveys, 2014. To appear; accepted 2014-01-30*. ACM, 2014.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE '07*, pages 95–104, New York, NY, USA, 2007. ACM.

[vdL04] Frank van der Linden, editor. *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*. Springer, 2004.

[VG07] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC'07*, pages 233–242. IEEE, 2007.

[Voi08a] Jean-Luc 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.

[Voi08b] JL Voirin. Method and Tools for Constrained System Architecting,. *INCOSE*, 2008.

[Wal13] Eric Walkingshaw. *The choice calculus: a formal language of variation*. PhD thesis, 2013.

[WHR14] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, May 2014.

[ZHJ03] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a uml profile for software product lines. In van der Linden [vdL04], pages 129–139.

[ZJ06] Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the uml: Deriving products. In Timo Käkölä and Juan C. Dueñas, editors, *Software Product Lines*, pages 557–588. Springer, 2006.

[ZMP12] Xiaorui Zhang and Birger Møller-Pedersen. Towards correct product derivation in model-driven product lines. In Øystein Haugen, Rick Reed, and Reinhard Gotzhein, editors, *SAM*, volume 7744 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2012.

List of Figures

Abstract

Systems Engineering is a complex and expensive activity in several kinds of companies, it imposes stakeholders to deal with massive pieces of software and their integration with several hardware components. To ease the development of such systems, engineers adopt a *divide and conquer* approach: each concern of the system is engineered separately, with several domain specific languages (DSL) and stakeholders. These languages are built within a set of dedicated representations to analyze an area of expertise, and the current practice is to rely on the Model-driven Engineering (MDE) paradigm to construct them.

On the other hand, systems engineering companies also need to construct slightly different versions/variants of a same system; these variants share commonalities and variabilities that can be managed using a Software Product Line (SPL) approach. A promising approach is to ally MDE with SPL – *Model-based SPLs (MSPL)* – in a way that the products of the SPL are expressed as models conforming to a metamodel and well-formedness rules. The *Common Variability Language (CVL)* has recently emerged as an effort to standardize and promote MSPLs; it is our adopted language for constructing MSPL.

Engineering an MSPL is extremely complex to an engineer: the number of possible products is exponential; the derived product models have to conform to numerous well-formedness and business rules; and the realization model that connects a variability model and a set of design models can be very expressive specially in the case of CVL. Managing variability models and design models is a non-trivial activity. Connecting both parts and therefore managing all the models is a daunting and error-prone task. Added to these challenges, we have the multiple different modeling languages of systems engineering. Each time a new modeling language is used for developing an MSPL, the realization layer should be revised accordingly.

The main objective of this thesis is to assist the engineering of MSPLs in the systems engineering field, considering the need to support it as earlier as possible and without compromising the existing development process. To achieve this, we provide a systematic and automated process, based on CVL, to randomly search the space of MSPLs for a given language, generating counterexamples that can server as antipatterns. We then provide ways to specialize CVL's realization layer (and derivation engine) based on the knowledge acquired from the counterexamples.

We validate our approach with four modeling languages, being one acquired from industry; the approach generates counterexamples efficiently, and we could make initial progress to increase the safety of the MSPL mechanisms for those languages, by implementing antipattern detection rules. Besides, we also analyse big Java programs, assessing the adequacy of CVL to deal with complex languages; it is also a first step to assess qualitatively the counterexamples. Finally, we provide a methodology to define the processes and roles to leverage MSPL engineering for new DSLs in an organization.