Leveraging software product lines engineering in the construction of domain specific languages

David Fernando Méndez Acuña

ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

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

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

David Fernando MÉNDEZ ACUÑA

Préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et en Automatique
Rennes, Bretagne Atlantique

Leveraging Software Product Lines Engineering in the Construction of Domain Specific Languages

Thèse soutenue à Rennes
le 16 décembre 2016

devant le jury composé de :

Isabelle BORNE
Professeur à l'Université de Bretagne Sud / examinatrice

Jean-Michel BRUEL
Professeur à l'Université de Toulouse / rapporteur

Reda BENDRAOU
Professeur à l'Université de Paris Ouest / rapporteur

Benoît BAUDRY
Chargé de recherche. INRIA Rennes / directeur de thèse

Contents

Introduction en français

Contexte scientifique

La complexité croissante des systèmes logiciels modernes a motivé la nécessité d'élever le niveau d'abstraction dans leur conception et mise en œuvre [1]. L'usage des langages dédiés a émergé pour répondre à cette nécessité. Un langage dédié permet de spécifier un système logiciel à travers des concepts relatifs au domaine d'application [2]. Cette approche a plusieurs avantages tels que la séparation des préoccupations et la capitalisation de l'expertise acquise dans un domaine particulier [3]. De nombreux langages dédiés sont apparus pour divers buts, par exemple : la construction d'interfaces graphiques [4], la spécification de politiques de sécurité [5] ou le prototypage d'applications mobiles [6].

La figure 1.1 présente un aperçu du processus de développement logiciel basé sur des langages dédiés [7]. Dans ce cadre, il y a trois types d'acteurs : concepteurs des langages, ingénieurs des systèmes, et utilisateurs finaux [3]. Les concepteurs des langages sont des informaticiens qui possèdent des compétences techniques pour la mise en œuvre de langages dédiés [8]. À travers un processus d'analyse du domaine et conception de langages, ils transforment les connaissances du domaine en forme de concepts de langage. Les ingénieurs des systèmes sont des développeurs de logiciels et du matériel qu'utilisent des langages dédiés pour construire un système concret. Finalement, les utilisateurs finaux sont ceux que profitent des fonctionalitées funis par les systèmes.

La notion de "systèmes logiciels" n'est pas limitée aux applications purement logicielles. Dans le cadre de cette thèse, nous adoptons une vision plus large de l'ingénierie de systémes qui inclut aussi des systèmes cyber-physiques tels que les avions ou les trains.

Problématique

Malgré les avantages fournis par l'usage des langages dédiés, cette approche présente des inconvénients qui remettent en question sa pertinence dans des projets réels de développement

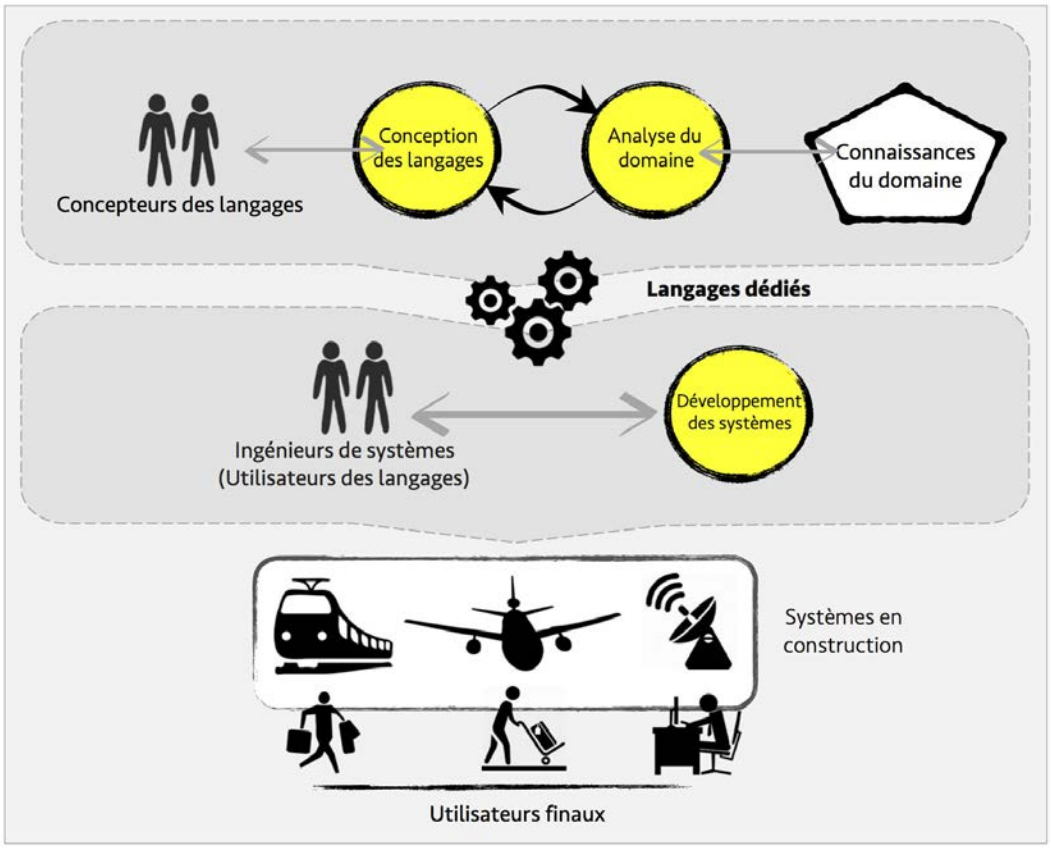


Figure 1: Aperçu du processus de développent logiciel basé sur des langages dédiés

logiciel [9]. L'un de ces inconvénients est le coût de la construction des langages dédiés. La définition et l'outillage de ces langages est une tâche complexe qui prend du temps et qui requiert des compétences techniques spécialisées [3]. Les concepteurs des langages doivent maîtriser plusieurs outils et passer beaucoup de temps à définir des artefacts comme la grammaire, interpreters ou les compilateurs. Cela vaut-il la peine de construire un langage dédié ? Il s'agit d'une question laquelle il n'est pas toujours facile de répondre [9].

Le processus de développement des langages dédiés devient encore plus complexe lorsque nous prenons en compte le fait que les langages dédiés peuvent avoir plusieurs *dialectes*. Un dialecte est une variante d'un langage qui introduit des différences au niveau de la syntaxe et/ou de la sémantique [10]. Ce type de variantes apparaît dans deux situations. La première situation est celle dans laquelle un même formalisme est utilisé pour la modélisation de plusieurs domaines. Par exemple, les machines à état ont été utilisées comme formalisme de modélisation dans une grande variété de langages dédiés conçus pour des objectifs différents tels que la définition de règles de navigation dans des interfaces graphiques [4] et la construction des jeux vidéo [11]. Ces langages utilisent les concepts classiques liés aux

machines à état comme État ou Transition. Cependant, chaque langage adapte ces concepts pour ses besoins particuliers.

La deuxième situation qui favorise l'apparition des variantes des langages dédiés est associée aux domaines complexes comprenant plusieurs aspects divergents. Dans ce cas, l'outillage qui supporte le domaine est un ensemble de langages dédiés différents (un langage pour chaque aspect) qui partagent quelques concepts. Par exemple, à cause de sa complexité, le domaine de gestion des lignes de trains requiert plusieurs langages dédiés comme : (1) celui présenté par James et al. [12] que permet la spécification et la vérification de plannings des systèmes de trains, et (2) celui présenté par Iliasov et al. [13] qui facilite la spécification et la validation de la capacité et la sécurité du système. Ces langages dédiés partagent quelques concepts comme la notion de train et du rail. Cependant, chaque langage fournit d'autres concepts liés à son objectif particulier ainsi comme une sémantique spécialisée.

Le phénomène des variantes dans les langages dédiés montre le pouvoir d'abstraction fourni par des formalismes tels que les machines à état ou des réseaux de Petri qui, avec quelques adaptations, peuvent être utiles dans plusieurs domaines. En outre, ce phénomène met en évidence la façon dont l'usage des langages dédiés peut proportionner la séparation des préoccupations dans un domaine, à travers de la définition des langages différents pour traiter chaque aspect du domaine. Néanmoins, la construction d'un ensemble de variantes de langages (a.k.a. une famille de langages) implique deux défis. Le premier défi est la réutilisation des parties communes entre les variantes : les concepteurs des langages doivent réutiliser autant de définitions que possible afin de réduir au maximum la mise en ouvre en partant de zéro dans la construction des variantes [14]. Le deuxième défi est la gestion des differences de chaque variante : les concepteurs des langages doivent garantir que chaque variante fournisse les particularités propres à son contexte d'application.

Dans cette thèse, nous proposons une réponse à la question de recherche suivante :

Question de recherche: Comment augmenter la *réutilisation* et gérer la *variabilité* durant la construction de familles de langages dédiés ?

Contributions

Afin de répondre à la question de recherche précédemment énoncée, la communauté de recherche autour de l'ingénierie des langages a proposé l'usage des lignes de produits. En conséquence, la notion de lignes de langages a récemment émergé. Une ligne de langages est une ligne de produits où les produits sont des langages [14, 15]. Le principal but dans les lignes de langages est la définition indépendente de morceaux de langage. Ces morceaux peuvent être combinées de manières différentes pour configurer des langages adaptés aux situations spécifiques.

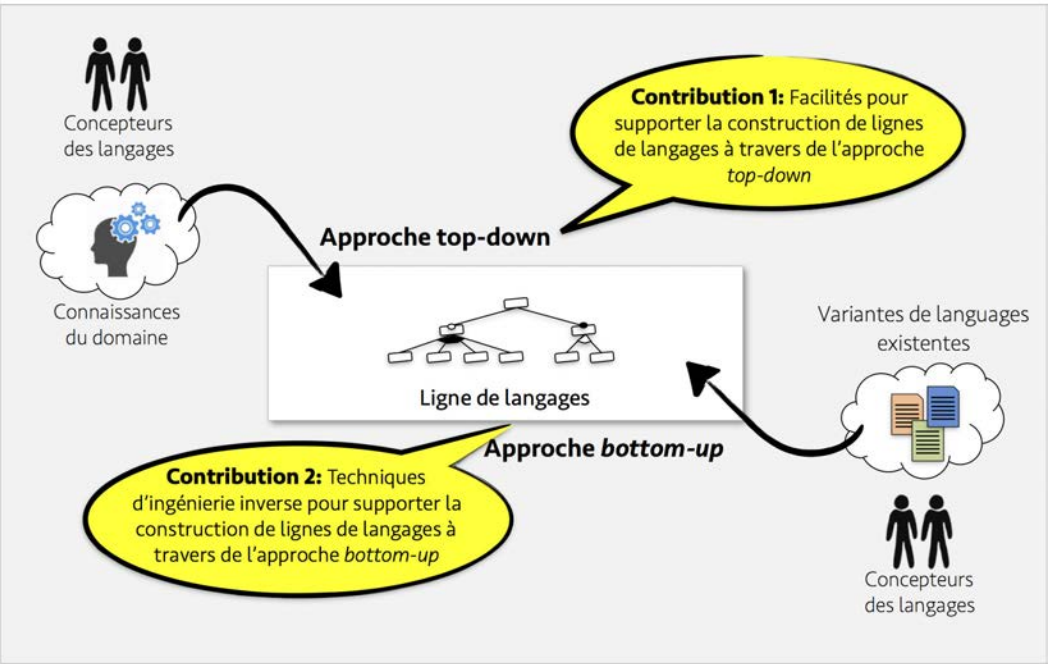


Figure 2: Deux approches différentes pour fare face à l'ingénierie des lignes des langages

D'une manière similaire aux lignes de produits, les lignes de langages peuvent être construites à partir de deux approches différentes: *top-down* et *bottom-up* comme illustré par la figure 2 . Dans l'approche top-down, les lignes de langages sont conçues et mis en œuvre au travers d'un processus d'analyse du domaine où les connaissances du domaine sont utilisées pour définir un ensemble de modules de langage qui réalisent les caractéristiques de la ligne de langages. En outre, les connaissances du domaine sont aussi utilisées pour représenter la variabilité de la ligne de langages à travers des modèles bien structurés qui, en plus, servent à configurer des langages particuliers. Dans l'approche *bottom-up*, les lignes des langages sont construites à partir d'un ensemble de variantes des langages existant au travers de techniques d'ingénierie inverse. Ces techniques doivent fournir des mécanismes pour : (1) récupérer un design et une implémentation modulaire qui comprenne tous les concepts de langage existants dans les variantes, et (2) synthétiser les modèles de variabilité qui capturent les différences entre les variantes.

À partir des approches précédemment énoncées, nous proposons deux contributions :

(1) **Des facilités pour supporter l'approche *top-down*.** Comme première contribution, nous proposons un ensemble de facilités pour supporter la construction de lignes de langages à travers de l'approche top-down. En particulier, nous proposons une approche de modularisation des langages qui permet la décomposition des langages dédiés comme modules de langages interdépendants. En plus, nous introduisons une

stratégie de modélisation pour représenter la variabilité dans une ligne de langages.

Ces facilités sont accompagnées d'un ensemble d'outils méthodologiques pour guider le processus de développement.

(2) **Des techniques d'ingénierie inverse pour supporter l'approche *bottom-up***. Comme deuxième contribution, nous proposons une technique d'ingénierie inverse pour construire, de manière automatique, une ligne de langages à partir d'un ensemble de variantes de langages existantes. La technique proposée inclut : (1) un mécanisme pour récupérer un design et une implémentation modulaire pour la ligne de produit á partir des concepts de langages existants dans les variantes, et (2) une approche pour synthétiser un modèle de variabilité qui représente les contraintes dont les modules des langages peuvent être combinés.

Contexte d'application

Cette thèse a été rendu possible grâce à plusieurs partenariats. Les besoins de ces partenariats ont guidé les contributions de la thèse. En outre, ces besoins ont été utiles pour la validation des idées proposées.

Le premier partenariat c'est le projet **VaryMDE**[1], qui est une collaboration industrielle entre l'INRIA et Thales. Le but du projet est de concevoir des mécanismes pour gérer la variabilité qui émerge pendant le processus d'ingénierie logicielle chez Thales. En particulier, le project étude la variabilité au niveau des modèles utilisés pendant le processus de développement logiciel, ainsi comme la variabilité au niveau des langages utilisés pour écrire les modèles. Tandis que la thèse de Ferreira [16] propose une solution pour le premier type de variabilité, dans cette thèse nous proposons une solution pour le deuxième.

Le deuxième partenariat c'est le projet **GEMOC**[2]. Le but du project est de fournir un ensemble d'outils pour supporter la coordination entre les différents langages utilisés dans la construction d'un même système. Notre interaction avec le projet GEMOC nous a permit de partager des idées avec des experts dans le domaine de l'ingénierie des langages.

Finalement, le troisième partenariat c'est le projet **RELATE**[3]. RELATE est un projet Marie Curie et son but c'est de former des docteurs au tour des technologies d'informatique dans les nouages. La contribution la plus importante de ce partenariat c'était sa plateforme pédagogique qui nous a fourni un environnement assez riche à travers de plusieurs séminaires dans des universités européens différentes.

[1] Website of the VaryMDE project: http://varymde.gforge.inria.fr/
[2] Website of the GEMOC initiative: http://gemoc.org/ins/
[3] Website of the RELATE project: http://www.relate-itn.eu/

Chapter 1

Introduction

1.1 Research Context

The increasing complexity of modern software-intensive systems has motivated the need of raising the level of abstraction at which software is designed and implemented [1]. The use of domain-specific languages (DSLs) has emerged in response to this need as an alternative to express software solutions in relevant domain concepts, thus favoring separation of concerns, and capitalising the knowledge adquired in a particular domain [3]. The adoption of this language-oriented vision has motivated the construction of DSLs for a large variety of purposes such as construction of user interfaces [4], specification of security policies [5], or mobile applications' prototyping [6].

Fig. 1.1 presents an overview of language oriented development [7], which includes three types of stakeholders: language designers, systems engineers, and final users [3]. Language designers are computer scientists who own the technical skills needed in the construction of DSLs. They know the techniques and tools available to conduct language development processes. Through a domain analysis process, language designers materialize the domain knowledge in the form of language constructs [8]. Systems engineers are software and hardware developers who use DSLs to build a particular system. Final users are those who thake advantage of the functionalities provided by the systems.

Note that the notion of "system" is larger than just software applications. In this thesis, we adopt a systems engineering perspective where DSLs are used not only to specify software, but also the behavior of cyber physical systems such as aircrafts or trains.

1.2 Problem Statement

Despite the advantages furnished by DSLs in terms of abstraction, separation of concerns, and improvement of productivity, the language-oriented development approach has important drawbacks that put into question its benefits [9]. One of those drawbacks is associated

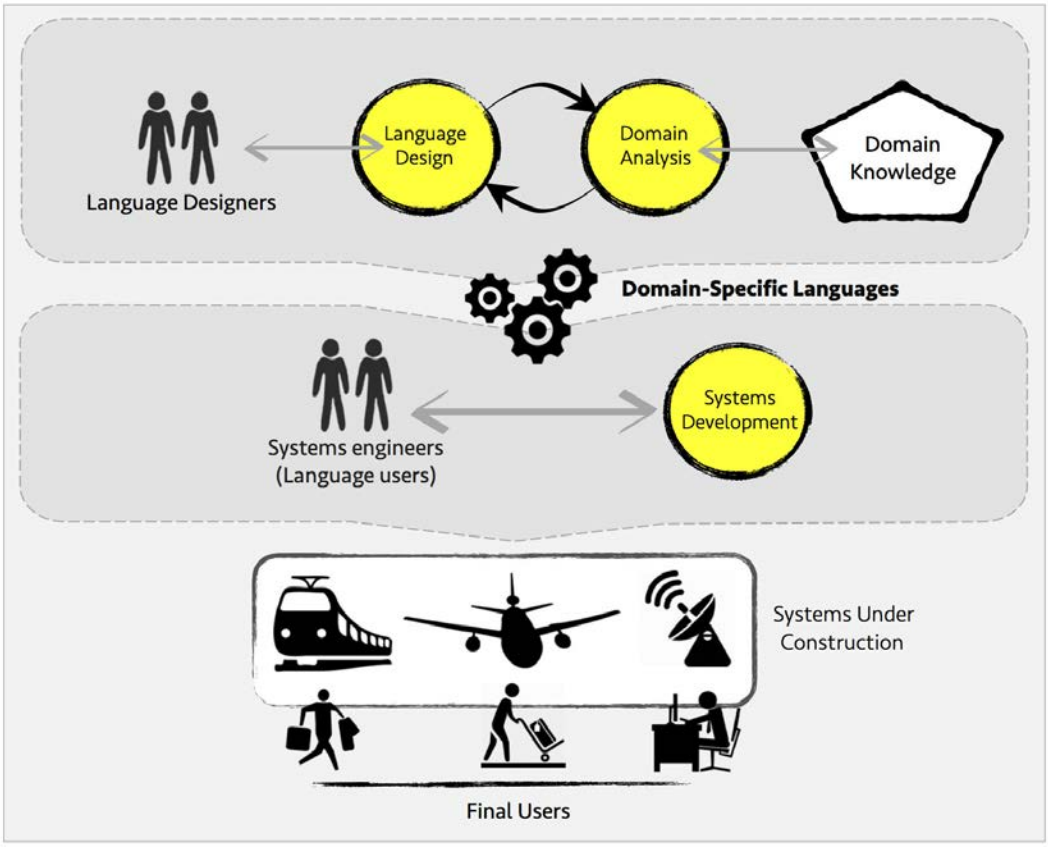


Figure 1.1: Overview of the language-oriented development approach

to the elevated costs of the language development process. The construction of DSLs is a time consuming activity that requires specialized background [3]. Language designers must own solid modeling skills and technical knowledge to conduct the definition of complex artifacts such as metamodels, grammars, interpreters, or compilers [3]. Is it worth the effort to build a DSL? That question is not always easy to answer [9].

The development of DSLs becomes even more complex when we consider that DSLs often have different *dialects*. A language dialect is a variation of a given DSL that introduces certain differences in terms of syntax and/or semantics [10]. This type of variations appear under two situations. The first situation is the use of well-known formalisms through different domains. Consider the case of finite state machines (FSMs), which have been used in a the construction of DSLs for a large spectrum of domains such as definition of graphical user interfaces [4] or games prototyping [11]. Those DSLs share typical state machine constructs such as states or transitions. However, each DSL adapt those abstractions to address the particularities of its domain.

The second situation that favors the existence of DSL variants is when the complexity

of a given domain motivates the construction of several DSLs to develop separately the various system concerns. In such a case, the tooling that supports the domain is composed of several DSL variants that share certain domain abstractions. For instance, suppose two DSLs: the former is a DSL for specification and verification of railway scheme plans [12]; the latter is a DSL for modeling and reasoning on railway systems' capacity and security [13]. These DSL share certain domain abstractions i.e., railway management. However, they both require different semantics and specialized constructs to achieve their purposes.

The phenomenon of DSL variants reflects the abstraction power of certain well-known formalisms –such as state machines or petri nets– that, with proper adaptations, can fit various domains. Besides, it shows how different concerns in a same domain can be addressed by different and complementary DSLs. However, the construction of a set of DSL variants (a.k.a., a language family) implies two challenges. The first challenge corresponds to the reuse of the commonalities existing among the variants: language designers should reuse, as much as possible, formerly defined language constructs to minimize implementation from scratch [14]. The second challenge corresponds to the management of the differences of each variant: language designers should guarantee that each variant provides the particularities related to their application context.

In this thesis, we aim to answer the following research question:

Research Question: How to increase *reuse* and manage the *variability* during the construction of families of DSLs?

1.3 Contributions

The research community on software language engineering has proposed the use of Software Product Lines Engineering (SPLE) to deal with the phenomenon of DSL variants [17]. This led to the notion of *Language Product Lines (LPL)*, i.e., a software product line where the products are languages [14, 15]. The main principle behind language product lines is to implement DSLs through *language features*. A language feature encapsulates a segment of a language specification that represents certain functionality [18]. Those features can be combined in different manners to produce different DSL variants.

Similarly to software product lines, language product lines can be built through two different approaches: top-down and bottom-up [19] (see Fig. 1.2). In the top-down approach, a language product line is designed and implemented through a domain analysis process where the knowledge owned by experts and final users is used to define a set of language modules that implement the language features of the product line. Also, the domain knowledge is used to specify variability models capturing the rules that define the way in which the language features can be combined to produce valid DSL variants. In the bottom-up approach, the language product line is built up from a set of existing DSL variants through reverse-engineering techniques. Those techniques should provide mecha-

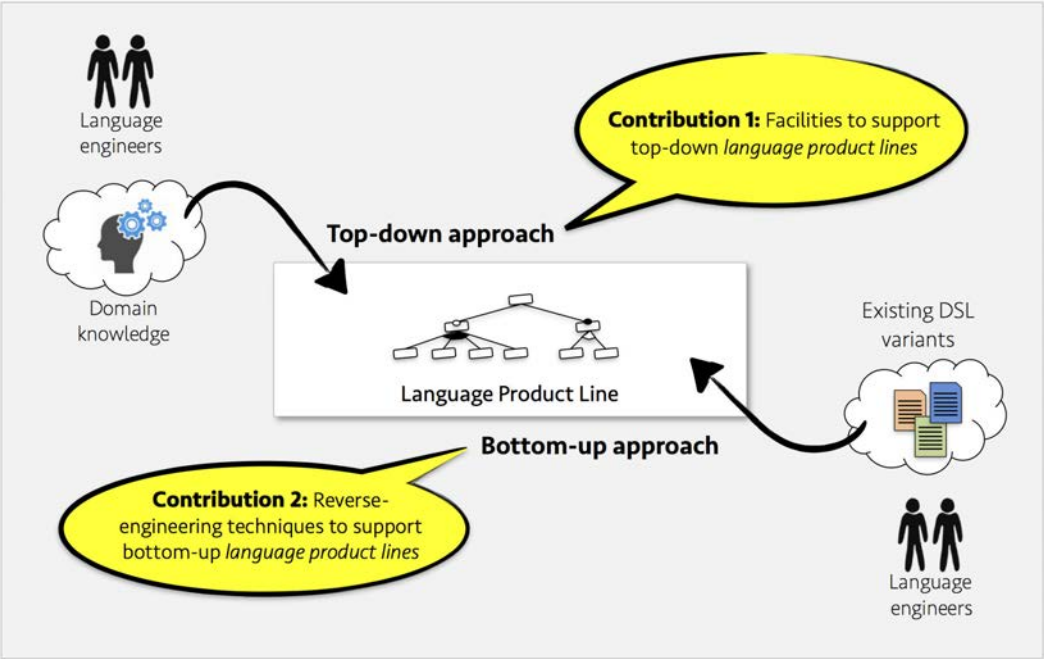


Figure 1.2: Two different approaches for Language Product Line Engineering

nisms for: (1) recovering of a language modular design and implementation including all the language constructs existing in the DSL variants; and (2) synthesis of the corresponding variability models.

Based on these different approaches for the construction of language product lines, we propose two contributions briefly explained below:

(1) **Facilities to support top-down language product lines.** As a first contribution, we introduce a set of facilities to support the development of top-down language product lines. Concretely, we provide a language modularization approach that permits the decomposition of DSLs into interdependent language features. We also introduce a modeling strategy to represent the variability of a language product line.

These facilities are accompanied with a set of methodological insights to guide language designers during the development process.

(2) **Reverse-engineering algorithms to support bottom-up language product lines**. As a second contribution, we introduce a reverse-engineering process to automatically build up a language product line from a set of existing DSL variants. This process encompasses: (1) the recovering of a language modular design and implementation that defines the features of the language product line; and (2) the synthesis of variability models that represent the corresponding variability.

1.4 Thesis' Realization Context

The development of this thesis involved the participation of several partnerships. Moreover, the needs of our partnerships have guided the contribution of the thesis and were useful during the corresponding validation.

The first partnership was the **VaryMDE** project[1], which is an industrial collaboration between INRIA and Thales. This project seeks to provide mechanisms to deal with the variability emerging during the systems engineering processes in Thales. Concretely, the project studies the variability at the level of the models used to represent the systems under construction, and at the level of the modeling languages used to build those models. Whereas the thesis of Ferreira [16] provides an answer to the first level, this thesis provides an answer to the second one.

The second partnership of this thesis was the **GEMOC** initiative[2]. In GEMOC, the participants work to provide a set of tools and methodologies to coordinate the use of several modeling languages in complex application contexts. Our interaction with GEMOC allowed us sharing ideas with experts in the field of software language engineers that are currently seeking solutions to similar problems.

The third partnership of this thesis was the **RELATE** project[3]. It is a Marie Curie project that aims to train doctors on cloud technologies. The stronger contribution of this partnership was its pedagogical infrastructure. RELATE provided a rich scientific environment through different workshops and internships in several universities in Europe that facilitate the exchange of ideas with other PhD students, experienced professors, and industrial stakeholders.

1.5 Outline

The content of this thesis is structured in four parts.

The first part, entitled **Preliminaries,** is dedicated to explain the present of language product line engineering through two chapters. Chapter 2 introduces background knowledge in terms of software product line engineering and domain specific languages. The purpose of his chapter is to introduce the vocabulary that we will use all along the document. Chapter 3 presents a study of the state of the art on the applicability of software product line engineering in the construction of domain specific languages. This chapter is structured in the form of a systematic literature review and closes with a set of open issues that we identified in the area of language product line engineering.

The second part of this thesis is entitled **Contributions**. It starts with a brief foreword, presented in Chapter 4, that highlights the subset of the open issues in the state of the art

[1] Website of the VaryMDE project: http://varymde.gforge.inria.fr/
[2] Website of the GEMOC initiative: http://gemoc.org/ins/
[3] Website of the RELATE project: http://www.relate-itn.eu/

that we will address in this thesis. Then, we introduce our contributions structured in two chapters. The first one, Chapter 5, presents a set of facilities for the construction of top-down language product lines. The second one, Chapter 6, presents a reverse-engineering strategy to support bottom-up language product lines.

The third part of this thesis is entitled **Implementation and Validation**. This part starts by describing the concrete tooling in which we implement our approach. In particular, in Chapter 7 we present Puzzle, an IDE built on top of the Eclipse Modeling Framework where we implemented the ideas introduced in this thesis. Then, in Chapter 8 we introduce three case studies that we use as validation.

Finally, the fourth part of this thesis, entitled **Closure** closes this document by presenting our conclusions and perspectives in Chapter 9.

Part I

Preliminaries

Chapter 2

Background

In this section, we introduce a unified vocabulary to facilitate the comprehension of the ideas presented in the rest of this document. In particular, we present a brief background in domain-specific languages and software product line engineering.

2.1 Domain-Specific Languages (DSLs)

In recent years, growing interest in domain-specific languages has led to the proliferation of formalisms, tools, and methods for software language engineering [2]. Hence, numerous techniques for implementing DSLs have emerged.

2.1.1 Implementation Concerns for DSLs

Just as traditional general purpose languages, domain specific languages are typically defined through three implementation concerns: abstract syntax, concrete syntax, and semantics [20]. The *abstract syntax* of a DSL specifies the set of language constructs that are relevant to its domain and the relationships among them. The *concrete syntax* of a DSL maps its language constructs to a set of symbols (either graphical or textual) that the users manipulate to crea te models and programs conforming to its abstract syntax. These representations are usually supported by editors that enable users to write programs using the symbols defined by the concrete syntax acting as the graphical user interface of the DSL. Finally, the *semantics* of a DSL assigns a precise meaning to each of its language constructs. More precisely, *static semantics* constrains the sets of valid programs while *dynamic semantics* specifies how they are evaluated at runtime.

2.1.2 Technological Spaces for the Implementation of DSLs

There are different technological spaces available for the realization of each of these concerns. The abstract syntax of a DSL can be expressed using grammars or metamodels. This

decision often depends on the culture of language designers. Grammars will most likely be favored by language designers with a strong background in language theory, parsers and compilers, while metamodels will most likely be favored by language designers with a strong background in object-oriented programming and model-driven engineering.

Regarding concrete syntax, DSLs can have either textual or graphical representations (or a mix of both). This decision is usually motivated by the requirements of final users, and the scenarios where the DSL will be used [21]. The implementation of a concrete syntax may for instance rely on the definition of a parser, or a projectional editor [22]. Since concrete syntax and semantics are usually defined as a mapping from the abstract syntax, the choice of the abstract syntax formalism strongly impact the choice of concrete syntax and semantics specification. In particular, the use of grammars to express abstract syntax implies that the DSL have a textual concrete syntax.

Regarding the specification of static semantics, there are not many design decisions to make beyond the constraints language to use. Usually, this selection is based on technological compatibilities with the formalism in which the abstract syntax is defined.

In turn, there are different methods for the definition of dynamic semantics: operational semantics, denotational semantics, and axiomatic semantics [23]. Operational semantics expresses the meaning of the language constructs of a DSL through the computational steps that will be performed during the execution of a program [23]. The definition of the operational semantics thus consists in an endogenous transformation that changes the execution state of conforming programs. Typically, the implementation of operational semantics corresponds to the definition of an interpreter.

Denotational semantics expresses the meaning of a DSLs through functions that map its constructs to a target formal language where the semantics is well-defined [24, 25]. When the target language is not a formal one (*e.g* another programming language with its own semantics), the term *translational semantics* is favored. The implementation of the translational semantics typically takes the form of a compiler.

Axiomatic semantics offers a mechanism for checking if the programs written in a DSL own certain properties. Examples of such properties are equivalence between programs or functional correctness (*e.g.* checking if the program is correct with respect to its specification in terms of pre- and post-conditions) [26].

The different methods for implementing the semantics of software languages are not mutually exclusive. There are authors [27, 26] that suggest that a DSL might own more than one type of semantics according to the needs of the final users.

Figure 2.1 sums up the discussed taxonomy in the form of a feature model [28]. Each feature represents a choice in terms of the technological space of a DSL. The relationships between features represent constraints on the combination of those choices. This taxonomy is consistent with the state of the art of language workbenches presented in [29]. Nevertheless, our taxonomy is more focused on the characteristics of the languages themselves rather than on the characteristics of the language workbenches. Our taxonomy is also con-

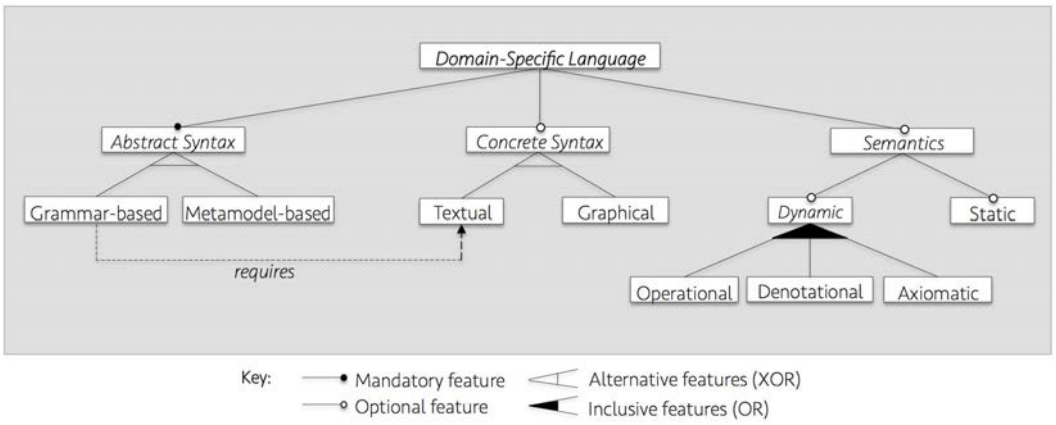


Figure 2.1: Technological spaces for domain-specific languages

sistent with the classification of DSLs introduced in [30].

2.1.3 External versus Internal DSLs

Another important decision when designing a DSL concerns the shape of the resulting language and the way to develop it. Language designers can choose to build either an external or an internal DSL[1]. The construction of an *external DSL* can be viewed as the creation of a new language [2] with its own dedicated infrastructure: editors, compilers and/or interpreters, tools, etc. In such a case, language designers must write a complete specification of their language using dedicated formalisms that offer the suitable expressiveness for defining each implementation concern. Since those formalisms are languages intended to specify languages, they are usually known as meta-languages and vary depending on the technological space chosen for the construction of the DSL.

In the case of *internal DSLs*, the principle is to take advantage of the infrastructure already provided by a host language [2]. The high-level domain concepts of the DSLs are implemented using the language constructs offered by the host language. Editors, parsers, or compilers of the host language can be reused as is, thus lowering the development costs compared to external DSLs. However, following this approach also implies that the capabilities of an internal DSL are restricted to the capabilities of the host language. The DSL must work with the programming paradigm, the type system, and the tooling provided by the host language. Besides, it is diffucult to forbit the use of the constructs of the host language, as well as providing feedback to the developers in terms of domain specific concepts. Because of all these reasons, an appropriate selection of the host language is of vital importance [32].

[1] Although the terms “internal” and “embedded” are sometimes used interchangeably, we use the term internal DSL to avoid the confusions sometimes associated with embedding as composition operator [31].

2.1.4 Language Workbenches

The notion of language workbench originates from the seminal work of Martin Fowler [33]. The main intent of language workbenches is to provide a unified environment to assist both the language designers in creating new DSLs. Modern language workbenches typically offer a set of meta-languages that the language designers use to express each of the implementation concerns of a DSL [34], along with tools and methods for composing and analyzing their specifications.

2.2 Software Product Lines Engineering (SPLE)

While traditional approaches to software development are intended to build individual software products, the SPLE approach proposes the construction of families of software products through a production lines' perspective [35]. A software product line is an infrastructure that enables to assembly several software products that share some commonalities with well-defined variations [35].

The central principle of the SPLE approach relies on the notion of *feature*. A feature encapsulates a characteristic that might be included in a software product. In that sense, a software product line can be viewed as a set of features available for the construction of a family of software products. Figure 2.2 shows the life-cycle of a software product line; it is divided into two phases: domain engineering and application engineering [35].

During the *domain engineering* phase, the objective is to build the product line itself (i.e., the infrastructure). This process includes the design and implementation of a set of common assets, as well as the explicit representation of the possible variations. The common assets of a software product line correspond to the software artifacts that implement the features. In turn, the possible variations of a software product line correspond to the combination of features that produce valid software products [36].

Since the notion of feature is intrinsically associated with encapsulation of functionality (i.e., characteristics), the implementation of the common assets requires a modular design of software artifacts that allows the definition of interdependent and interchangeable software modules. Those modules should be linked to the features they implement. In turn, the explicit representation of the variations requires a formalism to express the rules defining which are the valid combinations of features. Typically, those rules encode dependencies and/or conflicts between features. Feature models (FMs) [37] became the *"de facto"* standard to express these rules [38].

During the *application engineering* phase, the objective is to derivate software products according to the needs of specific customer segments [35]. Such derivation process comprises the selection of the features that should be included in the product, i.e., product configuration, as well as the assembly of the corresponding software modules, i.e., modules composition.

It is worth mentioning that both, the domain engineering and the application engineering

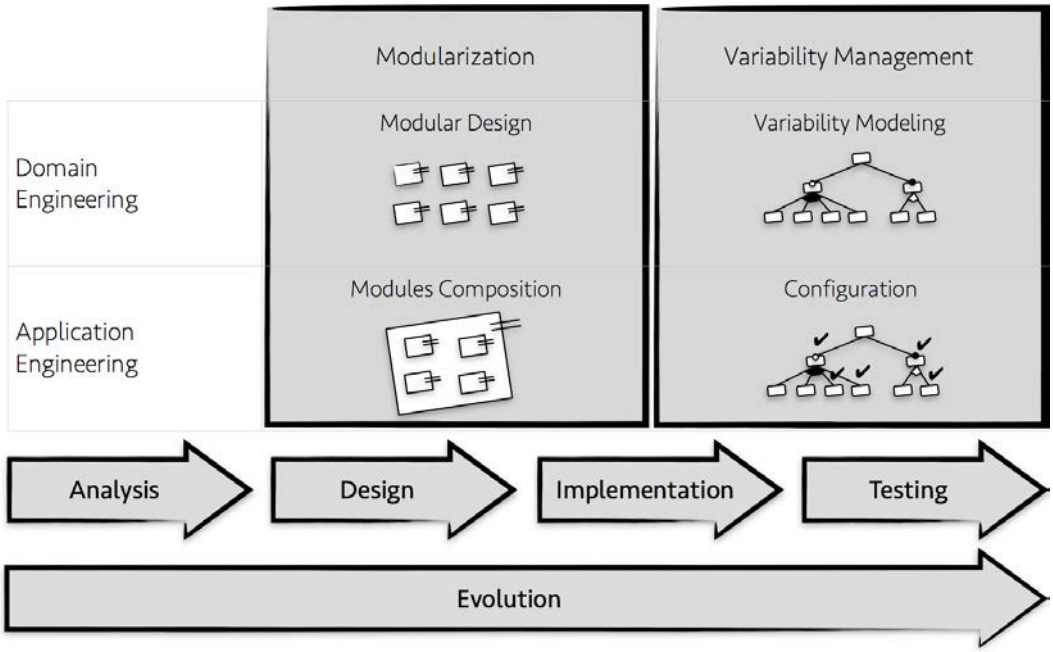


Figure 2.2: Phases of the SPLE's life cycle

phases are intended to be complete software development process. Hence, these phases require the typical steps towards the construction of software: requirements analysis, solution design, implementation, and testing [35]. Besides, software product lines are not static in time. The market needs evolve, and software product lines should support changes and adaptations to new business needs [39].

2.3 Summary

In this chapter, we introduced a unified vocabulary on the software engineering areas related to this thesis. We started by defining the notion of domain specific languages in terms of the implementation concerns and the diverse technological spaces available to conduct the language development process. We also discussed the relationship between languages and language workbenches. Afterwards, we introduced some vocabulary on software product lines engineering. We explained the domain engineering and the application engineering processes and the challenges to overcome during each of them.

Chapter 3

State of the Art: A Literature Review on Language Product Lines

As aforementioned, there is synergy between software product line engineering and the construction of DSLs [15]. It is possible to use the ideas provided the SPLE approach for systematic management of software variants to build DSL variants [17]. To this end, the life-cycle of software product lines should be adapted to the particularities of the language development process. Besides, language workbenches should provide the facilities that allow language designers to adopt those ideas [3].

Nowadays, we can find a number of approaches that directly or indirectly support this vision. Each approach provides certain facilities that can be used during the construction of a language product line. However, it is yet difficult for language designers to realize how those approaches can be used in a concrete language development project. This difficulty has two dimensions.

The first dimension is the partial coverage of the language product lines life-cycle. Not all the approaches address all the steps of such a life-cycle. Rather, they are often focused on a particular step (such as language modular design) without discussing the other ones. This can be explained by the fact that the approaches that result useful in language product line engineering were not necessarily conceived to this end. For example, not all the approaches in languages modular design are intended to support variability; many of them are motivated by other factors such as domain evolution and maintenance [40].

The second dimension is the potential misalignment between the technological space supported by each approach and the technological constraints of a particular language development project. Approaches in software language engineering are often conceived for a specific technological space which not always matches the requirements of a specific DSL development project. For example, an approach conceived for grammars-based DSLs with operational semantics may be difficult (or even impossible) to apply in a project where DSLs are meant to be metamodels-based with denotational semantics.

This chapter reports on an effort for organizing the literature on language product line engineering through a systematic literature review. The contributions of the chapter are two-fold. On one hand, we propose a definition to the life-cycle of a language product lines; we use it to analyze how current approaches support such a life-cycle. On the other hand, we establish a mapping between each approach and the technological space it supports.

In this sense, this chapter targets both researchers and practitioners. Researchers will find a comprehensive analysis of the life-cycle of language product lines, as well as a deep study of the strategies used in the state of the art to address such a life-cycle. In turn, practitioners will find in this chapter as a practical guide that they can use to find out the most convenient approach for a particular language development project according to the technological space used in a particular language development process.

Scope. In this thesis, we are concerned to the study of the variability in external DSLs. Hence, the study presented in this chapter is restricted to approaches supporting this type of DSLs.

3.1 Related Surveys and Literature Reviews

There are other literature studies in the field of software language engineering. Perhaps the most notable one is presented by Mernik et. al. [2] which provides a comprehensive analysis of the different development phases in the construction of DSLs: analysis, design, and implementation. Besides, the study introduces insights to identify the situations in which the development of a DSL is a correct decision, and discusses the capabilities of some of the language workbenches available in 2005.

Some years later, Kosar et al., [41] published a new research work in the form of a systematic mapping study which analyzes the trends of the research in DSLs from 2006 to 2012. The conclusions of the study permit to identify the issues that require more attention in the research of DSLs. For example, the authors clearly identify a lack of research on domain analysis and maintenance of DSLs.

A similar study is presented by Marques et al. [42]. In this case, the objective is to provide a systematic mapping study that allows to identify the tools and techniques used in the construction of DSLs. For example, the authors provide a comprehensive list of the host languages used in the development of internal DSLs. Besides, this work permit to understand in which domains the DSLs are being used. One of the conclusions in this regard is that the most popular domain for DSLs is the construction of Web-based systems. Other popular domains are embedded systems and networks.

Another relevant study on the literature of software language engineering is the one presented by Erdweg et al. [29]. More than studying research trends and techniques, this work focus on the analysis of language workbenches. The authors identify a comprehensive

set of features provided by the current language workbenches. Then, these features are used to compare the language workbenches among them. The technological spaces are viewed as features of the language workbenches.

Positioning. All the studies presented so far are intended to provide a general vision on the field of software language engineering. They analyze a large amount of approaches and offer offer different perspectives on the past, the present, and the future of the research in software languages. The literature review that we present in this chapter is intended to be more specific. Instead of global perspectives, we propose a detailed study in a localized issue: the use of software product lines techniques to increase reuse in the construction of DSL variants. In that sense, our survey can be compared with other surveys addressing localized issues. For example, the work presented by Ober et. al. [43] surveys different techniques to deal with interoperability between DSLs, and the work presented by Kusel et. al. [44] studies the approaches to leverage reuse in model-to-model transformations.

3.2 Research Method

In this section we provide the details about the methodology that we followed during the conduction of this systematic literature review. Concretely, we describe the search protocol that we used to find and select the articles included in the discussion. The search protocol is illustrated in Fig. 3.1. It was inspired on the guidelines for systematic literature reviews presented by Kitchenham et al [45].

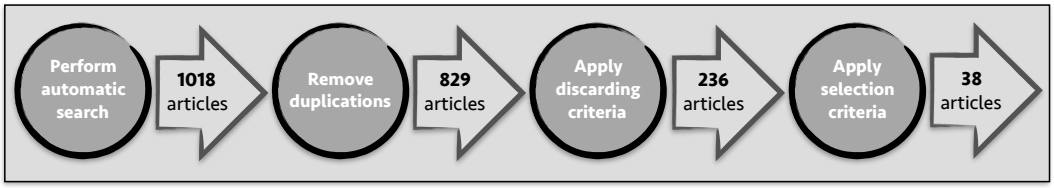


Figure 3.1: Protocol used to chose the articles included in the discussion.

Perform automatic search. The first phase of the protocol corresponds to an automatic search that collects a preliminary set of articles potentially interesting for the discussion. It was performed on four digital libraries: *ACM-DL*, *IEEEXplore*, *SpringerLink*, and *ScienceDirect*. These digital libraries where selected because they are used to publish the articles accepted in the conferences and journals typically targeted by the community of software language engineering. We decided to discard other sources such as *GoogleScholar* that do not guarantee that the indexed documents have been validated through peer-reviewing processes.

The automatic search was based on the following boolean expression: *(A* ***OR*** *B* ***OR*** *C)* ***AND*** *(D* ***OR*** *E* ***OR*** *F)* ***AND*** *(G)* where the corresponding strings are presented in the Tab

3.1. There might be several variants of these strings. For example, we can consider plurals and acronyms. However, this current setting retrieve all the all the works that we knew before conducting this literature review, which we consider as a good insight to believe that the automatic search is appropriated. This first phase resulted in 1.018 articles.

Research Area	Challenge	Scope
A: language engineering	**D:** variability	**G:** domain–specific languages
B: languages implementation	**E:** modularity	-
C: languages definition	**F:** composition	-

Table 3.1: Strings for the automatic search of the systematic literature review

Remove duplications. There are some cases in which an article is indexed by more than one digital library. As a consequence, some of the entries resulting from the first phase corresponded to the same article. Then, the second phase of our protocol was dedicated to remove those repetitions by checking the title of the paper as well as the target (conference or journal) in which it was published. This phase ends up with a set of 829 unique articles.

Apply discarding criteria. The keywords-based automatic search retrieved many articles. However, not all of them where relevant to the scope of the literature review. Therefore, we conducted a *discarding process* based on a two-fold *discarding criteria* presented below. Those criteria were applied on titles, abstracts and conclusions. At the end of this phase we obtained 236 articles.

- Discard the articles which do not deal with design and/or implementation of DSLs.
- Discard the articles which do not target any of the issues that we have identified as relevant languages product line engineering i.e., modularity, composition, and variability management.

Apply selection criteria. After applying the discarding criteria, we applied a second filter intended to select the articles will be definitely part of the discussion. To this end, we defined a two-fold *selection criteria* that we applied on the article's introductions. This phase resulted in 38 articles.

- Select the articles that have a clear contribution to one or several issues which are relevant on language product line engineering for external DSLs.
- Select the articles that present case studies if and only if they offer clear insights to address at least one issue of language product line engineering.

Final result. Fig. 3.2 presents the selected articles classified by year and type of publication. Of the 38 articles, 9 were published in journals, 17 in conferences, 11 in workshops. The figure shows an increasing interest on the subject represented in an increasing number of publications. The list of articles selected and discarded and in each step of the search protocol is available on-line[1].

[1] Survey's website: http://spltosle-survey.weebly.com/

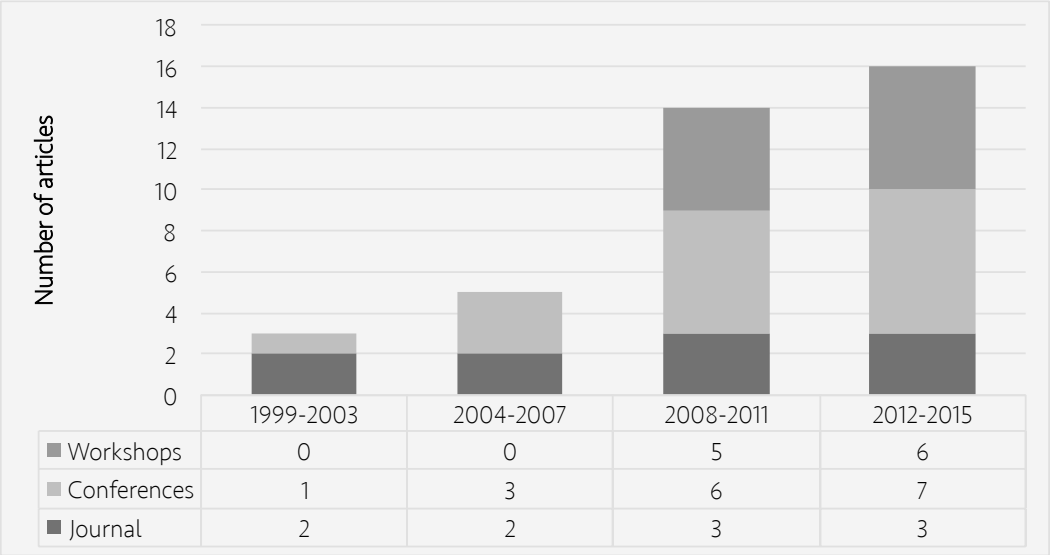


Figure 3.2: Number of articles per year and type of publication

Validation of the searching protocol. Despite the rigorous process that we followed to identify relevant articles, we wanted to reduce the risk of omitting some relevant article during the automatic search. So, we use a three-fold strategy to validate the automatic search.

First, before conducting this literature review, we established a set of articles that we knew in advance and that are relevant in this study. Then, we checked if those papers were included by the automatic search. The results were positive, all the papers in the predefined set were included in the automatic search.

Second, we collected the papers cited by the 38 articles finally included in the literature review. We select the references that we considered as relevant and we checked that they were also included in the automatic search. The results in this second validation strategy were positive as well; all these relevant articles were included in the automatic search.

Finally, we ask a variety of researchers to check our corpus and see if it has some missing works. We obtained several answers pointing out that the main works were considered.

3.3 Results

In this section, we present the results of our analysis of the articles obtained through the systematic process described above. Each article was read and analyzed according to the vocabulary presented in Section 2.

3.3.1 The Life-Cycle of a Language Product Line

Fig. 3.3 presents the life-cycle of a language product line. It addresses the same issues addressed by the life-cycle of a software product line introduced in section 2.2. However, there are certain particularities that should be considered. Those particularities come out from the specificities of the DSLs development process, and are discussed in several of the articles we selected during the survey protocol.

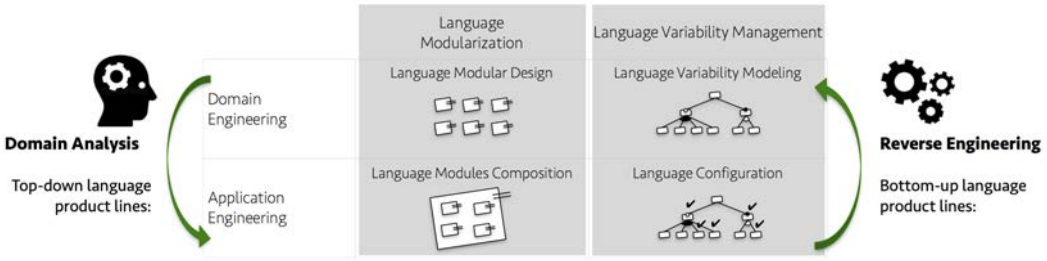


Figure 3.3: Phases of the life cycle of a language product line

Language Modular Design

Language modular desing supposes the intersection of two bodies of knowledge: software modularization and software language engineering as shown in Fig. 3.4. Hence, language modularization approaches inherit a combined complexity. On one hand, they have to support the modularization scenarios identified in the literature of software language engineering (i.e., extension, restriction, aggregation, unification, and extension composition). On the other hand, they are supposed to preserve basic principles behind software modularization (e.g., independent development, information hiding, and substitutability).

Languages modularization scenarios. A modularization scenario describes a situation where two language modules interact each other according to the nature of the dependencies existing between their language constructs. Those scenarios have been largely discussed in the literature [46, 22, 47] under different names. A unified vocabulary is presented in Table 3.3.1 and in the following we provide a brief description of each of them.

The modularization scenarios called self-extension and referencing, are out of the scope of the paper. The first one, because it is only applicable to the case of internal DSLs that will not be discussed in this paper. The second one because it refers more to the problem of languages integration and coordination than languages modularization.

- **Extension:** Extension is a modularization scenario where a *base language module* is enhanced with new capabilities provided by an *extension language module*. Such new capabilities can be either new language constructs or additional behaviors on top of the existing constructs [48].

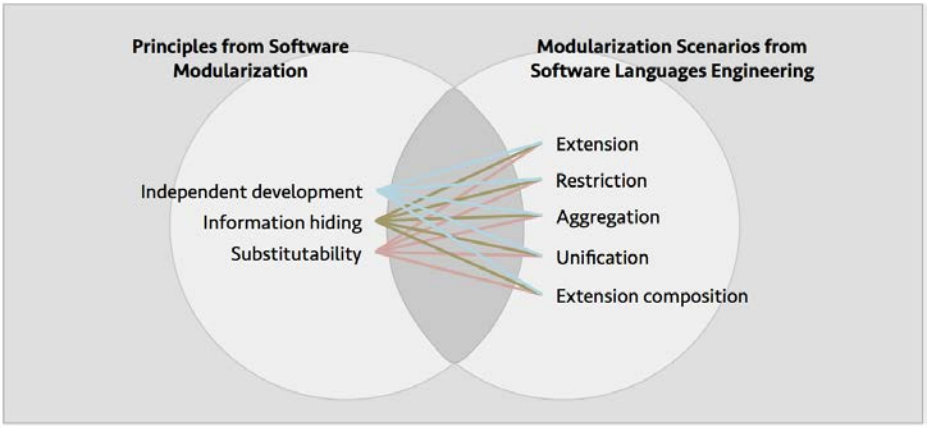


Figure 3.4: The relationship between software modularization and software language engineering

This article	Erdweg	Vöelter	Haber
	[46]	[22]	[47]
Extension	Extension	Extension	Inheritance
Restriction	Restriction	Restriction	-
Aggregation	-	Combination	Embedding
Unification	Unification	-	-
***Self-Extension**	Self-Extension	Embedding	-
***Referencing**	-	Referencing	Aggregation
Extension Composition	Extension Composition	-	-

Table 3.2: Language modularization scenarios in the literature. **(*) Out of the scope**

For instance, a language for expressing finite state machines can be extended to support hierarchical state machines by introducing the notion of composite state [49]. In such a case, the extension module introduces a new construct (i.e., CompositeState) completing the specification at the level of the abstract syntax, the concrete syntax, and semantics. The same language can be also extended with a pretty printing operation that returns a string representation of the entire state machine.

- **Restriction:** Unlike extension, restriction refers to the modularization scenario where capabilities of a *base language module* are reduced by a *restriction language module*. In other words, some of the constructs offered by the base language are disabled so they cannot longer be used. In [50] the author introduces an illustrative example for restriction where a base language for controlling a robot is restricted by removing some of the movement commands initially provided.

  Restriction is commonly identified as a particular case of extension [46, 22]. A language construct can be disabled by either overriding an existing language construct,

or introducing additional constraints that, in the validation phase, avoid the acceptance of models/programs which include the restricted construct. In this article, we consider extension and restriction as different modularization scenarios – they have not only different but also opposite purposes – that can be addressed by means of similar modularization techniques.

- **Aggregation:** Aggregation refers to the modularization scenario where a *requiring language module* uses (and incorporates) some language constructs provided by a *providing language module*. Consider for example the case where a language for modeling finite state machines uses the functionality provided by a constraint language for expressing guards in the transitions.

- **Referencing:** Similarly than aggregation, in referencing a requiring language module uses some constructs provided by a providing language modules. However, in this case the requiring language constructs are not incorporated by the requiring module but just referenced. Consider for example, the case in which a UML sequence diagram references the entities defined in a UML class diagram.

  Although this modularization scenario has been discussed in the literature of software language engineering, we did not found evidence that demonstrates its relevance in the language product lines life-cycle. This is because the objective of a language product line is to provide mechanisms to compose complete variants of a DSLs specification and, in this case, composition has a different meaning being more related to orchestration of models/programs.

- **Unification:** Unification refers to the modularization scenario where two independent languages, initially conceived for different purposes, are composed to produce a language with more powerful functionality. The main difference with respect to the other modularization scenarios introduced so far is that in this case there is not dependency between the involved languages. Rather, they are independent one from the other, and some glue code is needed for the composition. Note that in this case, the interface between the involved language modules is specified as a third language module containing the glue code.

  As an example of unification consider the research presented in [4] where a language for state machines is unified with the CSS (Cascading Style Sheets) language. The purpose is to facilitate the definition web interfaces. The work is based on the idea that a state machine can be used to represent user interactions whereas CSS can be used for expressing web pages' style.

The modularization scenarios presented so far can be applied in complex situations involving more than two language modules. This case is known as **extension composition**. Consider the case presented in Fig. 3.5 where the web styling language CSS is unified with language for expressing state machines that, in turn, uses the functionality of a constraints language (i.e., aggregation) and that is extended by the notion of composite states.

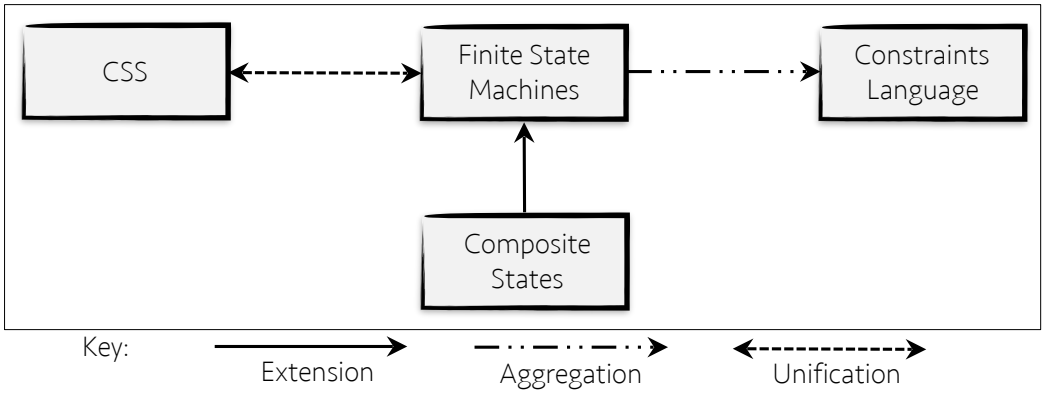


Figure 3.5: Example of extension composition

Principles from software modularization. The literature on software modularization is quite diverse. Many approaches and discussions have been proposed on modular design and implementation of software systems. However, there are three principles which have received special importance in the literature:

- **Independent development.** Independent development has been recognized as one of the advantages software modularization [51]. Software modules can be developed independently of each other even if they have dependencies with other modules [51]. However, this capability is not given for free. To support independent development, a modularization approach should provide a mechanism to express a set of assumptions a module under construction can consider as given. These assumptions are supposed to be fulfilled by other module. Software modules composition is the activity where the assumptions made a certain module are materialized in real functionality providing by another software module.

- **Information hiding.** Information hiding refers to the capability of explicitly selecting the information that a module exposes to other modules. Ideally, the exposed information should represent the functionality provided by the software component. The hidden information should correspond to the implementation details [52]. The importance of this principle relies on reducing the impact of module's evolution [53]. A module exposes the information which is supposed to be stable along the time (i.e., its functional specification) and hides the information which is more likely to change (i.e., its implementation) [52].

- **Substitutability.** Substitutability refers to the capability of interchange modules that provide the same functionality even if they are implemented differently. For example, there exists substitutability in a software system that can use different modules for LDAP authentication. As long as the modules provide the same functionality, a components model should permit to use any of them according to the needs in a

specific situation. The importance of substitutability is to reduce coupling between modules and to increase their potential reuse.

Language Modules Composition

One of the particularities of the DSLs implementation is that the tooling associated to a DSL (e.g., parsers or validators) is rarely built by language designers. Rather, such a tooling is automatically generated from the DSLs specification by language workbenches. For example, the parsers of the DSLs are often generated from a BNF-like grammar; those parsers might include capabilities such as syntax coloring or auto-completion.

As a consequence of this particularity, language modules composition can be performed either at the level of the specification [46, 50] or at the level of the tooling [54]. In the first case, the principle is to compose the specifications of each language module thus producing one joint specification that is used to automatically generate the tooling of the entire DSL. In doing so, the composition phase should compose the implementation artifacts containing the language modules specifications while clearly defining the semantics of the composition so the language constructs can correctly interact among them. In the second case, the principle is to first generate the tooling corresponding to each language module, and enable mechanisms to support the interaction between those "tooling modules".

Multi-Dimensional Variability Modeling

The variability existing between DSLs should be explicitly represented in order to identify the combinations of language modules that, once assembled, will produce valid DSLs. The fact that a DSL is specified in different implementation concerns implies different dimensions of variability [55, 56]. Let us summarize each of these dimensions.

- **Abstract syntax variability.** One of the motivations for the construction of language product lines is to offer customized languages that provide only the constructs required by a certain type of users. The hypothesis is that it will be easier for the user to adopt a language if the DSL only offers the constructs he/she needs. If there are additional concepts, the complexity of the DSL (and the associated tooling) needlessly increases and *"the users are forced to rely on abstractions that might not be naturally part of the abstraction level at which they are working"* [14].

  Abstract syntax variability refers to the capability of selecting the desired language constructs for a particular type of user. In many cases, constructs are grouped in *language features* to facilitate the selection. Such grouping is motivated by the fact that selecting constructs can be difficult because a DSL usually has many constructs, so a coarser level of granularity is required.

- **Concrete syntax variability.** Depending on the type of user, the use of certain types of concrete syntax may be more appropriate than an other one. Consider, for example, the dichotomy between textual or graphical notations. Empirical studies such as

the one presented in [57] show that, for a specific case, graphical notations are more appropriate than textual notations whereas other evaluation approaches argue that textual notations have advantages in cases where models become large [58]. Concrete syntax variability refers to the capability of supporting different representations for the same language construct.

- **Semantic variability.** Another problem that has gained attention in the literature of software language engineering is the semantic variation points existing in DSLs. A semantic variation point appears where the same construct can have several interpretations. Consider, for example, the semantic differences that exist between state machines languages explored in [59]. In that case, a state machine can either comply with the run-to-completion policy or accept simultaneous events. In the first case, events are processed sequentially (one after the other and one at a time) even if two events arrive at the same time. In the second case, simultaneous events can be attended at the same time. Semantic variability refers to the capability of supporting different interpretations for the same language construct.

These dimensions of variability are not mutually exclusive. There are cases in which several types of variability appear at the same time in the same language product line. In such cases, an approach for multi dimensional variability modeling [60] is required; it should take into account the fact that decisions taken in the resolution of the abstract syntax variability may affect decisions taken in the both concrete syntax and semantics variability.

Fig. 3.6 illustrates multi dimensional variability in the case of state machines. Each dimension of variability is expressed as a sub-tree. In the case of abstract syntax variability, a DSL for state machines is a mandatory feature that requires of an expression language. Timed transitions can be optionally selected as an extension of the DSL for state machines. The semantic variability dimension represents the decisions with respect to the behavior of the state machine. In this example, semantic choices regard to the perfect synchrony hypothesis (an event takes zero time for being executed) and events concurrency. Finally, the concrete syntax variability dimension presents the choice between graphical or textual DSLs for state machines.

Multi-staged Language Configuration

Once the variability of the language product line is correctly specified, and as long as the language features are correctly mapped to language modules, language designers are able to configure and derive DSLs. There are two issues to consider. First, the multi dimensional nature of the variability in language product lines, supposes the existence of a configuration process supporting dependencies between the decisions of different dimensions of variability. For example, decisions in the abstract syntax variability may impact decisions in semantic variability. Second, language product lines often require multi staged languages configuration. That is, the possibility of configuring a language in several stages and by different stakeholders.

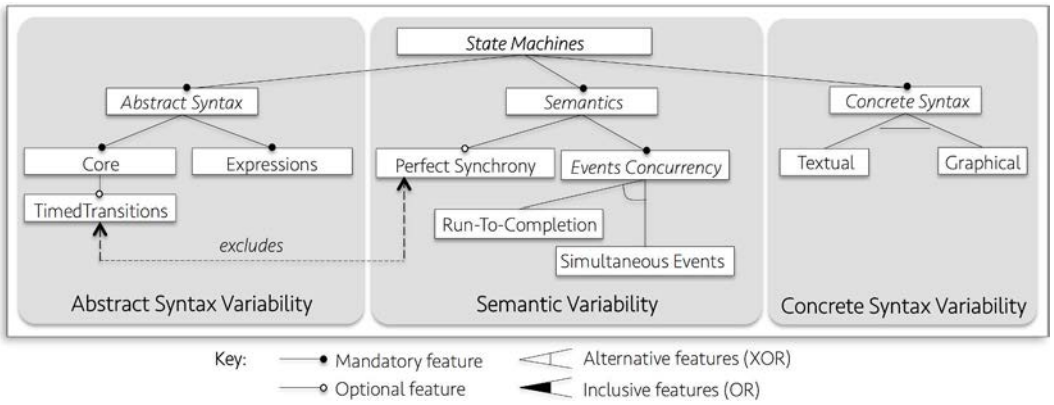


Figure 3.6: Example for multi-dimensional variability in language product lines

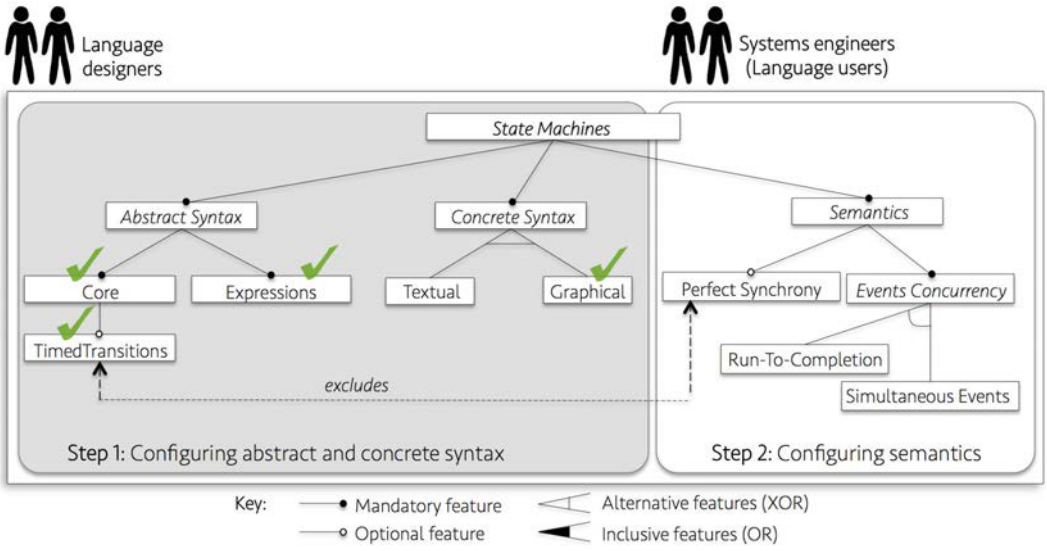


Figure 3.7: Example for multi-staged configuration in language product lines

Multi-staged configuration was introduced by Czarnecki et al. [61] for the general case of software product lines, and discussed by Dinkelaker et al. [62] for the particular case of DSLs. The main motivation to support such functionality is to transfer certain configuration decisions to the final user so he/she can adapt the language to exactly fits his/her needs [62]. In that case, the configuration process is as follows: the language designer provides an initial configuration. Then, the configuration is continued by the final user that can use the DSL as long as the configuration is complete. In doing so, it is important to decide what decisions correspond to each stakeholder.

As an example, suppose the multi staged configuration scenario presented in Figure 3.7. In that scenario, the language designer configures the abstract and the concrete syntax of a

DSL for finite state machines. Using those decisions, the language designer can produce a parser and an editor for the DSL. However, the semantics of the DSL remains open so the final user can configure it according to his/her modeling needs. Under the literature, this capability is known as *late semantic adaptation* [62]. It is important to mention, however, that this configuration scenario is just an example that illustrates the complexity of the configuration process associated to a multi dimensional variability modeling approach.

The Flow of the Life-Cycle: Top-down vs. Bottom-up

So far, we have presented the stages that compose the life-cycle of a language product line. We reviewed the main challenges that language designers have to overcome in terms of languages modularization and variability management through the domain engineering and the application engineering phases. Now, we will discuss the order in which those challenges are addressed during the development process, for which there are two different perspectives: top-down and bottom-up [19] as shown in Fig. 3.3.

In the top-down perspective, the domain engineering phase is performed first. Then, the produced artifacts are used to conduct the application engineering phase. Language engineers use domain analysis to design and implement a set of language modules and variability models from some domain knowledge owned by experts and final users. Those artifacts can be later used to configure and compose particular DSLs. This top-down approach is appropriated when language designers know in advance that they will have to build many variants of a DSL, and they have some clues indicating that the effort of building a language product line will be rewarded.

Differently, in the bottom-up perspective, the application engineering phase is performed before the domain engineering phase. Language designers start by building different DSLs that address different needs of final users. Then, when language designers realize that there is potential enough to build up a language product line from a set of existing DSLs, these DSLs are analyzed to extract that commonalities and variability that, with appropriated mechanisms, can be used to reverse engineer language modules and variability models.

3.3.2 Current Support for the Language Product Lines' Life-Cycle

After reading the articles obtained from the survey protocol, we identified a set of approaches supporting (partially or completely) the language product line's life-cycle. Those approaches are listed in table 3.3. In this section, we analyze how the aforementioned approaches support each step of the language product line's life-cycle.

There is a clarification to point out in this table. There are two approaches i.e., Neverlang+AiDE and ASF+SDF+FeatureHouse that are more than single approaches are the combination of several approaches. Whereas Neverlang and ASF+SDF are approaches for the construction of DSLs, AiDE and FeatureHouse are tools for variability management. In this literature review, we group those approaches since they have been used together

Name	Articles
LISA	[63, 64, 65, 66, 50]
Melange	[67, 68, 69]
Keywords-based modularization	[70]
Meta Programming System, MPS	[71, 22]
Modularization on top of ATL	[72]
Modularization on top of MetaDepth	[73, 74]
Gromp	[75]
Domain-concepts based modularization	[76, 77]
Interfaces-based modularization	[78]
Components-based LR parsing	[54]
Roles-based modularization	[79, 80]
MontiCore	[81, 82, 31, 55]
Neverlang+AiDE	[83, 40, 84, 85, 86]
ASF+SDF+FeatureHouse	[87, 88, 89, 18]

Table 3.3: Current approaches supporting language product lines

to support language product line engineering. This decision facilitates the study of the approaches.

Support for Languages Modular Design

Languages modularization has been largely discussed in the literature. Indeed, the most part of the approaches we survey in this literature review aim to support languages modularization. As a result of the analysis of those approaches, we have identified two *modularization techniques* intended to support modular languages design. Those techniques vary with respect to the way in which bindings between language modules are expressed; they are explained below:

- **Endogenous modularity.** In endogenous modularity, bindings between language modules are defined as part of the modules themselves. Usually those bindings are direct references between language modules such as the `import` clause. One important characteristic of endogenous modularity is that, because the modules are linked each other, the importing module has direct access to all the definitions provided by the imported one. As a result, the importing module can easily extend or use these definitions.

  This approach results quite useful from the language designer's point of view because it is straightforward, and it enables IDE facilities such as auto-completion. Contrariwise, the disadvantage of endogenous modularity is that it does not favor language modules substitutability because dependent modules are strongly linked each other. Replacing one language module for another one requires some refactoring to change the direct reference and, in many cases, adapt to the definitions of the

new imported module. This form of modularization favors high coupling between modules.

- **Exogenous modularity.** In exogenous modularity, bindings between language modules are defined externally. Usually, approaches based on exogenous modularity provide mechanisms (for example composition scripting languages) to describe those bindings in third-party artifacts that are the input of the composition process. In this case, language modules do not know the language modules they will be composed with. Hence, they cannot directly use foreign language constructs. To deal with this problem, language modules declare a set of requirements that are intended to be fulfilled in the composition phase. Those requirements are indirect references to language constructs that are defined in another module.

  Note that this approach favors language modules substitutability. Since there are not direct references between language modules, the bindings can be changed in the external artifact without modifying the modules themselves. Besides, because the dependencies between language modules are expressed as declarations, modules can be interchanged by any module that provide language constructs compatible with the declarations. The disadvantage of this approach is that it introduces additional complexity in the development process. Language designers need to consider not only the construction of the modules, but also the binding artifacts and manage indirect references.

The importance of these modularization techniques relies in two issues. First, they influence the way in which the approaches support language modularization scenarios and its subsequent composition. Second, they constraint the way in which the approaches address software modularization principles.

Modularization techniques vs. modularization scenarios. In the following we discuss how the modularization scenarios are addressed through the modularization techniques. Afterwards, we analyze the composition strategies required in each case. Note that e do not include the modularization scenario called referencing in our analysis because, as we said earlier, we did not find evidence of its relevance in the language product lines life-cycle.

- **Endogenous modularity to support *extension* and *restriction*.** Endogenous modularity is useful to support extension and restriction of language modules via the `import` clause. Base modules are *imported* by extension modules; then extension modules can access and enhance the definitions of the base modules while introducing new constructs, behavior or constraints.
- **Exogenous modularity to support *extension* and *restriction*.** There are approaches based on exogenous modularization to support extension and restriction of language modules. In this case, the extension modules do not import a specific base module. Rather, the binding between the extension and the base module is specified eternally, so extension modules have not direct access to the constructs of the base module.

In the composition phase, both base and extension modules are composed thus introducing to the base module the constructs, behavior, or constraints defined in the extension module. Following this strategy, the base module can be easily substituted by another one as long as it provides the constructs used as extension points.

- **Endogenous modularity to support *aggregation*.** Endogenous modularity is also useful to support aggregation of language modules. As we said before, in aggregation of language modules we have a requiring language module that uses the language constructs provided in a providing language module. In the case of endogenous modularity, the requiring language module imports the providing language modules thus having access to all its language constructs.
- **Exogenous modularity to support *aggregation*.** Aggregation can be supported via exogenous modularity. In this case, the requiring module declares a set of language constructs that are supposed to be implemented in a providing language module.

  One might think that there is not difference between endogenous and exogenous aggregation. However, the fact that the requiring language module does not reference directly the providing language module results quite useful for facilitating modules substitutability. Note that the providing language can be replaced for any other language that implements the construct Expression. To do so, language developers need just to change the binding and execute the composition. As a matter of fact, we can find approaches in which the substitutability is even more favored by introducing some mechanisms that permit to declare those requirements in a more abstract way.
- **Endogenous modularity to support *unification* and *extension composition*.** Endogenous modularity is also useful to support unification of two language modules (a.k.a., left/right modules). As aforementioned, ideally these language modules should not be modified during the unification process, so the most common solution is to create a third language module (a.k.a., integration module) containing the glue code that specifies the semantics of the integration. In the case of endogenous modularity, the integration module directly imports the left/right modules and uses (or extends) their language constructs to define a unified language.
- **Exogenous modularity to support *unification* and *extension composition*.** Exogenous modularity is also useful to support unification of language modules. In this case, the relationship between the third module and the other two is not direct importing but indirect and specified in a composition artifact.

Support for Language Modules Composition

As aforementioned, language modules composition can be performed either at the level of the specification (i.e., specification composition) or at the level of the tooling (i.e., tooling composition). The first strategy is most common; in most of the approaches reviewed in this article language modules composition produces a unified specification from a set of

language modules. Differently, tooling composition is rarely mentioned in the literature of software language engineering. Indeed, we found only one approach using tooling composition [54].

The solution strategy to implement tooling composition can be compared with the classical mechanisms to achieve software composition. After all, parsers, interpreters, or compilers are software tools that can use classical composition strategies such as interfaces. The article presented by Wu et al., [54] introduces a new parsing algorithm that supports modular parsing. In this approach, the parser of a DSL can be defined as a set of interdependent parser modules, and the complete parsing process is supported by the parsing algorithm that can "visit" several parser modules. In the case of specification composition, we found two different techniques used for the composition techniques:

- **Direct linking to compose endogenous modules.** In endogenous modularization, implementation artifacts are physically related via direct linking realized through the `import` clause. Direct linking corresponds to include all the content of the referenced artifact at the beginning of the referencing one.
- **Artifacts merging to compose exogenous modules.** In exogenous modularization, implementation artifacts are completely independent so the have no direct links between them. Hence, their content should be unified during the composition phase, thus producing a unique artifact containing a joint specification.

Many of the approaches studied in this literature review propose composition strategies based on direct linking exclusively. Indeed, despite the limitation of direct linking with respect to the substitutability principle, it has demonstrated to be useful to support the modularization scenarios presented in Section 3.3.1. This is because the importing language module can access all the language constricts of the imported one, and establish any type of relationship among them. However, there are other approaches whose composition strategy is based on some composition operators, which formally define the semantics of the composition. Such operators are optional in the case of direct linking, but mandatory in the case of artifacts merging. In the following we explain the composition operators that we found during the conduction of this survey.

- **Inheritance.** Inheritance is a mechanism to exploit reuse coming from object oriented programming. It has demonstrated to be useful as composition operator for language modules [50]. Generally, approaches that use inheritance as composition operator are based on endogenous modularity. This can be explained by the nature of the inheritance relationship, which is intended to reuse the specification provided in a concrete implementation artifacts for which direct linking results quite useful.

  In inheritance, the composition rules are based on the notions of `extending` and `overriding`, which are useful to compose the interfaces of the modularization scenarios introduced in section 3.3.1. In the case of extension, the extension point is a language construct in the base language that is `extended` by some language

construct in the extension module. In restriction, the restriction point is a language construct in the base language that is `overridden` by some language construct in the restriction module. In aggregation, the provided module accesses the requiring module through an inheritance relationship. Indeed, if the requiring module inherits the providing one, then it will be permitted to access (and use) all their constructs. In the cases of unification and extension composition, the glue code can be implemented by a language module that inherits all the modules involved in the composition. Naturally, multiple inheritance is needed to support this scenario.

- **Merge.** Merging can be defined as the combination of two artifacts where *"the common elements are included only once, and the other ones are preserved"* [90]. In the case of language modules composition, merging is an additive operator that sums the constructs provided by the language modules involved in the composition while avoiding repetition. Due to the capability of merging to integrate independent artifacts, it is generally used by approaches based on exogenous modularization.

  When using the merge operation, the "common elements" become quite important. They represent the interfaces between the language modules, and can be used to address all the modularization presented in section 3.3.1. In the case of extension, the extension module declares language constructs corresponding to the extension point as part of their definitions. In the composition phase, the declaration of the extension point provided by the extension module is merged with the implementation of those constructs provided by the base module. A similar approach is used in the case of aggregation. The requiring language module declares the language constructs that it uses, and in the composition phase these declarations are merged with the actual implementation of the constructs provided by the providing language module. This idea can be generalized for the composition of more than two language modules to support both unification and extension composition.

- **Superimposition.** Superimposition is a particular case of merging. Indeed, the superimposition operator is defined as the merge of two implementation artifacts. As a result, its applicability for language modules composition is quite similar than the one for merging. The difference between merge and superimposition is that, in the later, the implementation artifacts are intended to be modeled in a tree-based structure. This is because the superimposition operator is recursive, and their semantics are formalized as composition of trees. Similarly than in the case of merging, superimposition is used in approaches based on exogenous modularization.

- **Weaving.** Weaving is another operator has been used in the literature to support languages modularization. It supposes the existence of a base module and an aspect [90]. The base module is enhanced by the features introduced by the aspect. Language designers must define the exact point (i.e., the join cut) in which those features will be injected. The definition of weaving let it open to be applicable in both endogenous and exogenous modularization. That means that the binding between the aspect and

the base module can be defined either in the aspect itself or in an external artifact.

The nature of the weaving operator makes it appropriate to support languages extension and restriction. In that case, the extension point is the point cut which is enhanced with the functionality implemented in the advice. During the conduction of this literature review we did not find any evidence that indicates that weaving can be used to support the other modularization scenarios.

Modularization techniques vs. software modularization principles. The modularization technique used by an approach also impacts its capability to address the software modularization scenario. We observed that the support for software modularization principles is deeply associated with the shape of the interfaces between language modules, which differ in the case of endogenous and exogenous modularization. When the approach uses endogenous modularity, this interface is a direct link between the module and the software modularization principles are hardly addressed. Differently, when the approach uses exogenous modularity, it must provide a mechanism to define the dependencies between language modules in a way that favor software modularization principles.

- **Endogenous modularization and *independent development*.** The use of endogenous modularization constitutes a barrier for addressing independent development. This is because, by definition, direct linking between two language modules implies the existence and accessibility of the imported module. A module `A` can import a module `B` if and only if the module `B` is already implemented and it can be accessed. Hence, the development of module `A` depends on the development and release of module `B` so it is not true that the module `A` can be developed independently.
- **Exogenous modularization and *independent development*.** Unlike endogenous modularization, exogenous modularization favors independent development. When there is not a direct references between language modules, the modularization approach must provide a mechanism (for example a required interface) to allow language designers the expression of the needs of a language module with other modules. This mechanism permits the development of language modules independently from their needs while assuming that those needs will be eventually fulfilled.
- **Endogenous modularization and *information hiding*.** The use of endogenous modularization is a potential barrier for addressing independent development. This is because direct linking between language modules often permits to access to all the information of the providing module. To avoid this capability, the modularization approach should provide a mechanism to protect specification elements in the imported module. The protected clause used by Java is an example of such a mechanism.
- **Exogenous modularization and *information hiding*.** The use of exogenous modularization is neither a barrier to the information hiding principle nor a guaranty of its support. At a first view, the fact that the language modules are not directly referenced between them protects in somehow their specification elements since the interaction

between the modules is expressed through explicit interfaces that define the needs that one module has from others. However, those needs are not necessarily constrained. Hence, a language module could access forbidden information from other modules. To guarantee the information hiding principle, a modularization approach should offer a mechanism (e.g., provided interfaces) to filter out the information that can be accessed by other language modules.

- **Endogenous modularization and *substitutability*.** To support substitutability, an approach should provide two capabilities. First, the binding between the needs of a requiring module and the functionality offered by the providing one should be expressed in such a way that it can be easily re-directed. In this sense, the use of endogenous modularization is a barrier for addressing substitutability. Replacing one language module for another one requires some refactoring to change the direct reference and, in many cases, adapt to the definitions of the new imported module.
- **Exogenous modularization and *substitutability*.** It is easier to address substitutability when the approach uses exogenous modularization. The bindings are defined externally and they can be re-directed without modifying the language modules themselves. Ideally, an approach to language modularization that aims to address substitutability should provide a mechanism that permits to compose two language modules even if the requiring and providing constructs are implemented differently.

Mapping current approaches with language modularization capabilities. Table 3.4 shows how current approaches support languages modularization. For each approach, the figure indicates the modularization scenarios it supports and the software modularization principle it addresses. Besides, the table shows the modularization techniques and the corresponding composition operator.

Note that endogenous and exogenous modularization are not mutually exclusive. Indeed, in the cases of Modularization on top of MetaDepth, the approaches use endogenous modularization to support language modules extension and restriction and exogenous modularization to support language modules aggregation. In both cases, independent development and substitutability are partially addressed in the sense that they are possible for the case of aggregation but not for the case of extension neither for restriction. Those approaches does not provide any mechanism to express provided interfaces so they do not guarantee the information hiding principle.

Support for Multi-Dimensional Variability Modeling

In contrast to the large amount of articles on languages modularization, we found very few articles addressing languages variability management. In the following we discuss the current advances in this regard.

Support for multi dimensional variability modeling. All of the current approaches supporting languages variability modeling are based on feature models. However, they dif-

	LISA	Mélange	Keywords-based modularization	Meta Programming System, MPS	Modularization on top of ATL	Modularization on top of MetaDepth	Gromp	Domain concepts-based modularization	Interfaces-based modularization	Components-based LR parsing	Roles-based modularization	MontiCore	Neverlang+AiDE	ASF+SDF+FeatureHouse
Modularization capabilities														
Modularization scenarios														
Extension	●	●	-	●	-	●	●	●	-	●	-	●	●	●
Restriction	●	●	-	●	-	●	●	●	-	●	-	●	●	●
Aggregation	●	●	●	●	●	●	-	●	●	●	●	●	●	●
Unification	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Extension composition	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Software modularization principle														
Independent development	-	●	●	-	-	❍	●	-	●	❍	●	❍	●	●
Information hiding	-	-	-	-	-	-	-	-	●	●	●	-	-	-
Substitutability	-	●	●	-	-	❍	❍	-	❍	❍	❍	❍	❍	❍
Modularization mechanisms														
Modularization technique														
Endogenous modularization	●	-	-	●	●	●	-	●	-	●	-	●	-	-
Exogenous modularization	-	●	●	-	-	●	●	-	●	-	●	●	●	●
Composition strategy														
Specification composition	●	●	●	●	●	●	●	●	●	-	●	●	●	●
Tool composition	-	-	-	-	-	-	-	-	-	●	-	-	-	-
Composition operator														
Inheritance	●	-	-	-	-	-	-	-	-	-	-	●	-	-
Merge	-	●	●	-	-	-	●	-	●	-	●	●	●	-
Superimposition	-	-	-	-	-	-	-	-	-	-	-	-	-	●
Weaving	-	●	-	-	-	-	-	-	-	-	-	-	-	-

Key: ● **YES** - **NO** ❍ **PARTIALLY**

Table 3.4: Mapping current approaches and language modularization capabilities

fer in the modeling approach they use to represent the variability. We found three different modeling approaches explained below:

- **Feature models supporting abstract syntax variability.** Wende et al. [79] use feature models as a documentation artifact to present a catalog of language features that can be combined each other to produce DSL variants. This modeling approach is illustrated in Figure 3.8. Each feature is associated to a language module that is completely specified in terms of abstract syntax, concrete syntax, and semantics.

  This approach is useful in language product lines with abstract syntax variability.

Each language feature can be viewed as a fully specified set of language constructs that will be selected or not according to the needs of the final user. However, support for concrete syntax variability and semantic variability is limited. For example, if a language designer needs to represent semantic variability, he/she will have to define two language modules with the same abstract and concrete syntax but with different semantics. In doing so, language designers would introduce specification clones (repeated segments of specification) all along the language product line, thus increasing maintenance costs.

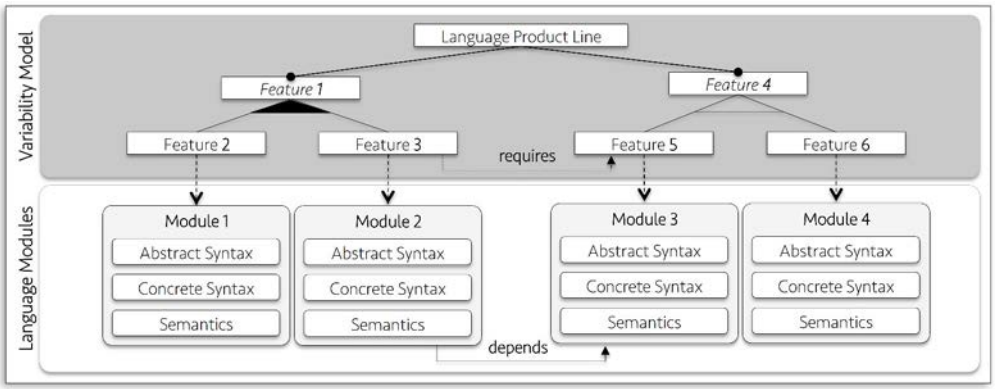


Figure 3.8: Boolean feature models for representing abstract syntax variability

- **Multi-dimensional variability with concern-specific features.** The approach presented in [91] proposes to deal with variability management on top of Neverlang using the COMMON VARIABILITY LANGUAGE. This approach considers not only abstract syntax variability, but also concrete syntax and semantic variability. To this end, the approach proposes to use feature models by following the modeling strategy illustrated in Figure 3.9 where abstract features are used to represent a segment of abstract syntax that may vary in terms of concrete syntax or semantics. The children features represent the possible variations.

  Consider, for example, the feature called "Feature 1" that represents a language feature with a variation point in the concrete syntax. The "Feature 1" is mapped to a language module that encapsulates the corresponding abstract syntax and semantics (that do not vary). Besides, there are two children that indicate the different representations of the feature. Each of these children is mapped to a language module that implements the corresponding concrete syntax.

- **Multi-dimensional variability with concern-specific subtrees.** The approach to support variability management is based on feature models to represent multi dimensional variability [55]. In other words, this approach supports not only abstract syntax variability, but also concrete syntax and semantic variability. To support multi dimen-

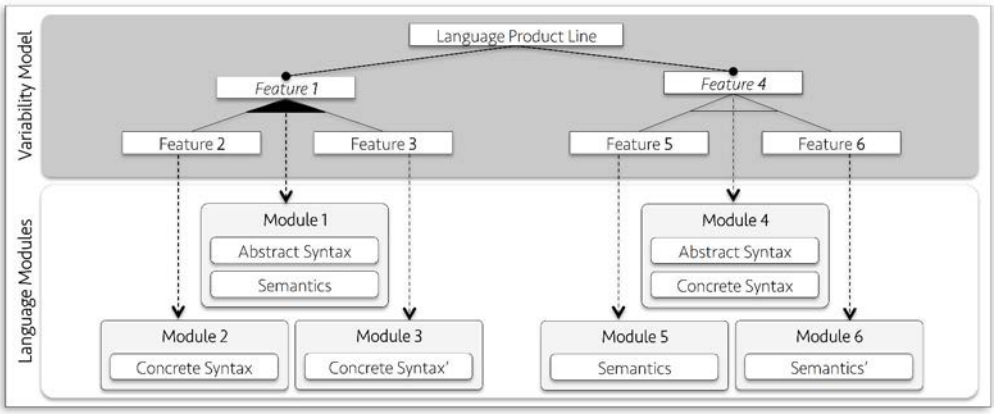


Figure 3.9: Boolean feature models for representing syntactic and semantic variability

sional variability, the authors propose an approach in which the variability model has (at the first level) one child feature of the root for each dimension of variability. Figure 3.10 illustrates this modeling approach. In that example, the sub-tree concrete syntax has two language features (i.e., feature 1 and feature 2) that represent two different representations of a particular construct (or set of constructs). Each feature is implemented in a language module that implements the corresponding functionality in the corresponding implementation concern. 3.10.

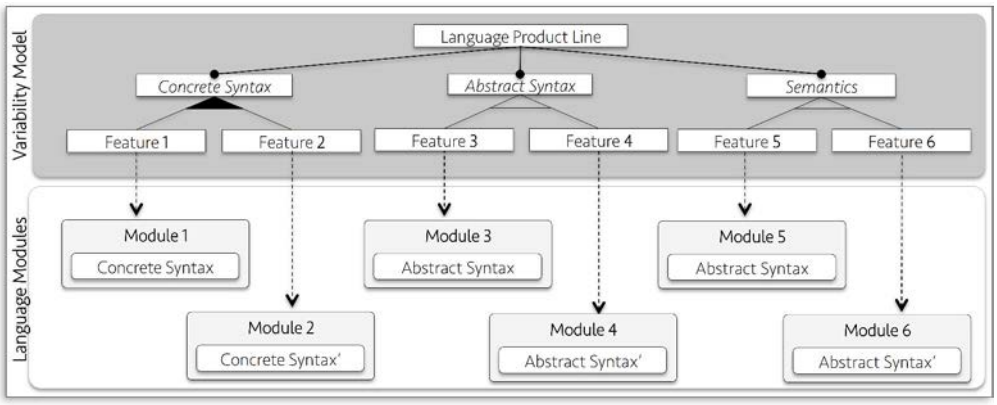


Figure 3.10: Boolean feature models for representing multi-dimensional variability

On the other hand, the approach presented in [18] proposes the use of FEATUREHOUSE to deal with languages variability management on top of ASD+SDF. It supports abstract syntax variability, syntactic, and semantic variability. The modeling strategy is the same than the one used by MontiCore (Figure 3.10), and provides facilities to configure and derivate languages from the feature model.

Note that multi-dimensional variability supposes that the modularization approach supports the definition of each implementation concern (i.e., abstract syntax, concrete syntax, and semantics) in a different language module. In the cases where this separation is not provided, multi dimensional variability is not possible.

There are some approaches that go further in the study of variability management in language product lines by proposing to automatically infer the variability models. The approach presented in [92] proposes a search-based technique to find a features model that represents the variability existing in a set of language modules while optimizing an objective function. This approach uses an ontology that describes the domain concepts of the language product line. The second approach (presented in [15]) refines the former by removing the ontology. This improvement is motivated by the difficulty behind the construction of such ontology. Then, the authors propose to annotate the BNF-like grammar with certain information that is used to create a variability model.

Is worth highlighting that these approaches support not only abstract syntax variability but also concrete syntax and semantic variability. In the first case, since the ontology represents the domain from both the syntax and semantic point of view, then it is possible to use it to identify all the existing variation points. In the second case, the annotations provide the expressiveness enough to address all these dimensions of the variability.

Support for Multi-Staged Language Configuration

The use of feature modeling to represent the variability of language product lines entails some support for languages configuration. Indeed, the most part of the tools for feature modeling, provide capabilities to create different configurations from a given variability model. Once the variability of a language product line is modeled in a tool-supported feature model, language designers have some facilities to produce configurations that can be used in to select the corresponding language components and start the composition process.

Once language designers have a tool for configuration of feature models, they should define a multi staged configuration process involving final users if needed. As the reader might imagine, multi staged configuration is more an organizational capability that defines the configuration decisions each stakeholder should make and basic configuration tool support can be used to this end. There are, however, some approaches that enable certain tool features to ease this process. In particular, the approach presented in [55] proposes to express configurations of a feature models configuration files. Each file is a refinement of of the configuration decisions then providing certain traceability and more clear assignation of the configuration decisions. This is relevant when feature models become bigger.

Mapping current approaches and capabilities on variability management. Table 3.5 shows how current approaches support languages variability management. In particular, the figure presents a table that, for each approach, indicates the capabilities it provides in terms of multi-dimensional variability modeling and multi staged languages configuration.

	LISA	Mélange	Keywords-based modularization	Meta Programming System, MPS	Modularization on top of ATL	Modularization on top of MetaDepth	Gromp	Domain concepts-based modularization	Interfaces-based modularization	Components-based LR parsing	Roles-based modularization	MontiCore	NeverLang+AiDE	ASF+SDF+FeatureHouse
Variability modeling														
Functional variability	-	-	-	-	-	-	-	-	-	-	●	●	●	●
Multi-dimensional variability	-	-	-	-	-	-	-	-	-	-	-	●	●	●
Languages configuration														
Features selection	-	-	-	-	-	-	-	-	-	-	-	●	●	●
Multi-staged configuration	-	-	-	-	-	-	-	-	-	-	-	●	●	●

Table 3.5: Mapping current approaches and variability management capabilities

3.3.3 Mapping Approaches and Technological Spaces

In the following, we describe the technological space supported by each of the approaches studied in this survey.

- **LISA.** LISA supports the construction of textual DSLs where the abstract syntax, concrete syntax, and semantics are specified through attribute grammars (which is usually associated to a form of denotational semantics [93]). LISA supports modular language design and language modules composition. To this end, this approach uses ideas from object-oriented programming. In particular, it introduces the notion of inheritance in attribute grammars. In LISA, language modules are defined as attribute grammars that can have inheritance relationships among them.

- **Mélange + Kermeta.** Mélange supports the construction of DSLs where the abstract syntax is specified in metamodels, static semantics are defined as class invariants, and the dynamic semantics is defined operationally as aspects in the Kermeta meta-language [68, 69]. Melange supports modular languages design and language modules composition.

- **Keywords-based modularization.** Keywords-based modularization supports the construction of textual DSLs where the abstract syntax is defined in an object-oriented model (a sort of metamodel) and the semantics is defined denotationally through transformations. Keywords based modularization supports modular languages design and language modules composition. To this end, this approach introduces the notion of *keyword*. A keyword is a language module that contains an object-oriented

model to express abstract syntax, a regular expression to express concrete syntax, and a localized transformation to express semantics.

- **Meta Programming System (MPS).** MPS supports the construction of graphical and textual DSLs whose abstract syntax is defined in metamodels, the concrete syntax is defined through projectional editors, and the semantics is defined operationally in Java programs. This approach supports modular languages design and language modules composition. To this end, MPS enables modularization of the metamodels and provides mechanisms to propagate such modularization at the level of the concrete syntax and semantics.

- **Modularization on top of ATL.** Modularization on top of ATL supports the construction of DSLs where the abstract syntax is defined in metamodels and the semantics is defined denotationally through transformations. The approach supports modular language design and language modules composition by introducing modularization on top of the ATL transformation language [94].

- **Modularization of top of MetaDepth.** MetaDepth supports the construction of textual DSLs where the abstract syntax is defined in metamodels, static semantics is defined in constraints, and dynamic semantics is defined denotationally through transformations. MetaDepth supports modular languages design and language modules composition. To this end, this approach is based on metamodels extensions, and structural concepts.

- **Gromp.** Gromp supports the construction of graphical DSLs whose abstract syntax is defined in metamodels, and the concrete syntax is defined in PICTURE (a platform for the definition of graphical DSLs built on top of EMF). Gromp supports modular languages design and language modules composition. To this end, this approach provides a composition language that allows language designers to manually describe the composition of several language modules.

- **Domain-concepts based modularization.** Domain-concepts based modularization supports the construction of graphical DSLs whose abstract syntax is defined in metamodels and dynamic semantics is defined denotationally through transformations. This approach supports modular design and language modules composition. To this end, the approach provides a pool of composition operators that can be used for expressing the composition of language modules (which are referred to as domain-concepts). This approach is applied to a case study that is presented in [77].

- **Interfaces-based modularization.** Interfaces-based modularization supports the construction of metamodels based DSLs. Neither concrete syntax nor semantics are addressed in this case. This approach supports modular languages design and language modules composition. To this end, the approach follows a principle based

on language interfaces. The authors define metamodel interfaces to metamodel fragments that can be later composed according to some predefined operators.

- **Components-based LR parsing.** Components-based LR parsing is an approach that supports the modular definition of parsers defined through grammars. From all the articles reviewed in this literature review, this is the only one that uses tooling composition as composition strategy. The authors justify their decision by arguing that it favors low coupling in language modular design. Semantics are not addressed by the approach.

- **Roles-based modularization.** Roles-based modularization supports the construction of textual DSLs where the abstract syntax is defined in metamodels, the concrete syntax in BNF-like grammars, and semantics is specified operationally in Java programs. The ideas proposed in this approach are implemented in the LanGems workbench [80]. This approach supports modular languages design and language modules composition. Additionally, the authors propose a first step towards variability modeling. Languages configuration and derivation is not addressed.

- **MontiCore.** MontiCore supports the construction of textual DSLs where the abstract and concrete syntax are defined in BNF-like grammars, and semantics is defined denotationally. MontiCore provides an extended format for grammars that enhances the classical context-free grammar notation with some mechanisms offered by metamodels (e.g., data types, inheritance, interfaces, and associations). This approach supports modular languages design and language modules composition. Additionally, we found an approach to address variability modeling and languages configuration on top of MontiCore [55].

- **Neverlang+AiDE.** Neverlang supports the construction of textual DSLs where the abstract and concrete syntax are specified in BNF-like grammars, static semantics is specified as validation programs, and dynamic semantics is defined operationally in Java programs. This approach provides support for modular languages design and languages composition. Additionally, we found several approaches that support variability management on top of Neverlang.

- **ASF+SDF+FeatureHouse.** ASF+SDF+FeatureHouse supports the construction of textual DSLs where the abstract and concrete syntax are specified in BNF-like grammars, and semantics is specified denotationally through transformations. This approach is a tool chain composed of ASF, SDF, and FEATUREHOUSE. ASF+SDF provides support for modular languages design and languages composition [87, 88, 89]. In turn, the approach presented in [18] proposes the use of FEATUREHOUSE as a languages variability management framework on top of ASD+SDF.

Table 3.6 introduces a summary of the discussion presented in this section.

	LISA	Mélange	Keywords-based modularization	Meta Programming System, MPS	Modularization on top of ATL	Modularization on top of MetaDepth	Gromp	Domain concepts-based modularization	Interfaces-based modularization	Components-based LR parsing	Roles-based modularization	MontiCore	Neverlang+AiDE	ASF+SDF+FeatureHouse
Abstract syntax														
Grammars-based	●	-	-	-	-	-	-	-	-	●	-	●	●	●
Metamodels-based	-	●	●	●	●	●	●	●	●	-	●	-	-	-
Concrete syntax														
Textual	●	-	●	●	-	●	-	-	-	●	●	●	●	●
Graphical	-	-	-	●	-	-	●	-	-	-	-	-	-	-
Semantics														
Static	●	●	-	●	-	●	-	-	-	-	-	●	●	●
Operational	-	●	-	●	-	-	-	-	-	-	●	-	●	-
Denotational	●	-	●	-	●	●	-	●	-	-	-	●	-	●
Axiomatic	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 3.6: Mapping current approaches and technological spaces

3.4 Open Issues in Language Product Line Engineering

During the conduction of this literature review, we found an important amount of approaches to support the implementation of language product lines. We provide evidences to state that, with the current meta-languages, it is possible to build a language product line in several technological spaces. Concretely, we analyzed a considerable amount of approaches for languages modularization, and we show that the definition of language modules is possible as well as the use of feature models to represent variability in DSLs. We also presented approaches to automatically generate a first version of those feature models.

Despite all these advances, there are still some open issues on both top-down and bottom-up language product line engineering. Fig. 3.11 shows a mind-map that summarizes those open issues, which are explained in the reminder of this section.

3.4.1 Open Issues in Top-Down Language Product Lines

Most of the approaches that we found during the conduction of this literature review are focused in the construction of top-down language product lines. They provide facilities in terms of language modularization and variability management that, in most of the cases, are supported by language workbenches. Using those approaches, a language designer can define a language modular design, as well as represent the corresponding variability.

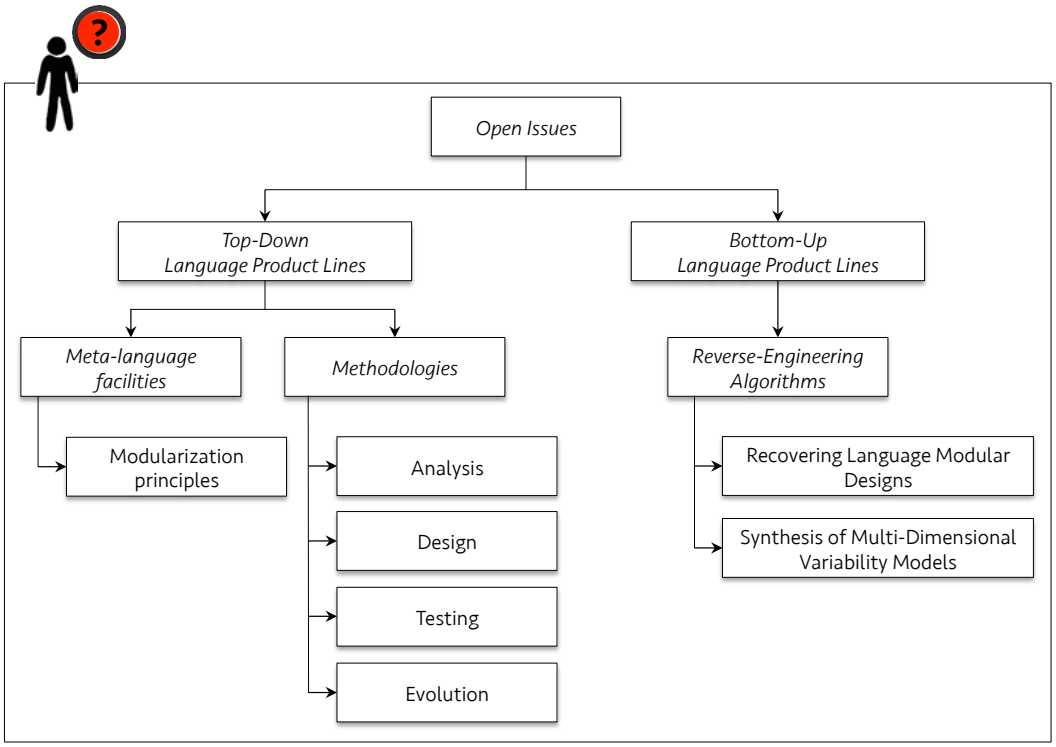


Figure 3.11: Open issues in language product line engineering

However, current approaches in language product lines engineering have certain limitations that can be classified in two categories: limitations from the point of view of the meta-language facilities themselves, and limitations from the point of view of the methodology needed to build a language product line.

Limitations From the Point of View of Meta-Language Facilities

We observed that there are still some limitations in terms of meta-languages supporting language product lines, particularly in context of languages modularization. Despite the community on software language engineering has put many attention on modularization issues and that current approaches support very well languages modularization scenarios; very few approaches are successful on addressing software modularization principles (i.e., independent development, information hiding, and substitutability).

This is due to the fact that not all the language modularization approach propose appropriate interface mechanisms that preserve these principles. We claim that the notion of language interfaces should be better developed to achieve languages modularization approaches that effectively address software modularization principles.

Limitations From the Point of View of the Methodology

In terms of methodology, there is a long path to follow. The methodological aspects of the construction of a language product line are rarely mentioned. The analysis, design, testing, evaluation, and evolution of language product lines are open issues that should be addressed to provide appropriate support to language designers. In the reminder of this section, we discuss the open issues and questions that should be addressed to facilitate the construction of language product lines.

Analysis in language product line engineering. In the development of software product lines, the requirements analysis phase is dedicated to the identification and documentation of the common and variable requirements of the product line's final users [35]. Naturally, this analysis is quite important in the case of language product lines as well. Language designers must clearly identify and classify the final users of the products of the DSLs that will be produced by the product line, and define their common and variable requirements.

An example of this requirements analysis phase in the context of language engineering is presented by Cazzola and Olivares [95]. They identify common and variable requirements for programming languages in the context of education in computer science. In this case, the final users are the students that are classified according to their level in a programming course. The language product line permits to incrementally introduce language features to the students according to their evolution in the learning process.

The analysis phase in language product lines has certain particularities that should be better investigated, however. For example, the notion of requirements in DSLs should be better defined because there is no consensus about what it means. According to the example mentioned above, a requirement in language engineering is associated to the language constructs. Contrariwise, Kolovos et al. [96] associate DSLs requirements to properties such as supportability, orthogonality, or simplicity. A definition of requirements in DSLs development should be accompanied with methodologies to identify them, as well to documentation well practices.

Design in language product line engineering. In the development of software product lines, the design phase is dedicated to the definition of a *modular design* that establishes the set of software modules provided by the product line and the interfaces between them. Besides, in this phase software engineers perform the construction of variability models that capture the variations required by the product line [35]. To address this activity, software engineers often use design patterns and modularization properties (such as low coupling and high cohesion) [97] to define an appropriate modular design that support the product line's variability while favoring quality attributes such as extensibility and maintainability.

In the case of language product lines, however, the definition of an appropriate modular design is rarely discussed. As a matter of fact, current approaches are mostly focused on providing the tooling (i.e., language workbenches) to define and compose language modules, and the design itself has been put aside. As a result, language engineers still have

problems at the moment of breaking down a language into interdependent language modules to support variability. More concretely, some of the questions that language designers must face are: What is the current level of granularity at which language modules should be defined? How to modularize a language to support the three different types of variability? Is it possible to define design patterns and modularization principles to facilitate modular languages design? Are the properties of low-coupling and high-cohesion relevant in the development of DSLs? If so, how to realize and measure those properties in a given language product line?

Testing in language product line engineering. In the development of software product lines, the testing phase is dedicated to the validation of the implementation artifacts that compose the product line's infrastructure [35]. To achieve such validation, software engineers must test both the software modules and the variability models.

On one hand, testing interdependent software modules is usually performed in three phases: unit testing, integration testing, and system testing [98]. In unit testing, each language module is validated independently to guarantee that the functionality it provides is correctly implemented. In integration testing, the interaction between modules is validated to guarantee that the contracts between modules are respected. In system testing, the system as a whole is validated. Naturally, these basic testing steps should be adapted to the fact that language modules are pieces of abstract syntax enhanced with concrete syntax and semantics.

On the other hand, testing the variability models correspond to verify that the configurations that can be obtained from the feature models produce valid products. To this end, software engineers must design an appropriate set of test scenarios and execute them on the possible configurations. This process can be extremely expensive when variability models become bigger, and prioritization strategies might be required [99].

All the issues mentioned above are still open in language product lines. Language designers usually face questions such as: How a language module can be tested independently to verify its localized functionality? How to perform integration tests to validate the interaction between several language modules? How to test entire DSLs produced as a composition of language modules? The difficulty of answering those questions relies on the fact that current approaches in languages testing (such as [100], or [101]) are intended to test completely specified languages. When a language is partially identified – a language module is a partial language – it cannot be compiled/interpreted and current testing approaches fail. Some research is needed to find out mechanisms that permit to express the requirements that a language module has with its environment (i.e., the required interface) and artificially simulate these requirements as done by mock objects in object oriented programming. It is worth mentioning that in doing so, researchers should into consideration the different implementation concerns of DSLs; not only the abstract syntax but also concrete syntax and semantics [29].

Evolution in language product line engineering. Because of the dynamism of business needs, requirements in software products are constantly changing, and evolution is a recurrent concern in software development. The situation is not different in software product line engineering. When the requirements of the stakeholders of a product line change, there is an impact on the product line's infrastructure, and some adaptations might be needed [102].

Evolution in software product lines supposes several challenges and depending on the nature of the evolution in the requirements the infrastructure might change differently [103]. Those changes can be relatively simple to manage (such as the introduction of a new feature without impacting existing ones), or quite painful (such as split or combination of existing features, which supposes adaptations in the variability model re-modularization of the common assets). Besides, requirements evolution might impact not only the implementation artifacts but also documentation and tests.

Evolution is also a recurrent issue in the development of DSLs due to the domain evolution problem. Changes in the domain rules, or simply improving the domain understanding might an impact on DSLs specifications [104]. As a result, the problem of evolution in language product lines is also a concern that language designers must address. Some of the questions to deal with are: How to re-modularize a components-based DSL? How to capture changes in the domain as evolution in variability models?

3.4.2 Open Issues in Bottom-Up Language Product Lines

Differently to the case of top-down language product lines, the bottom-up perspective has received little attention. Few works have proposed strategies and algorithms to address such a perspective, most of them are focused on the synthesis of the variability models. We can identify two main limitations of current approaches.

Limitations while Recovering Language Modular Designs

As largely discussed so far, the very first step towards the construction of a language product line is to achieve a appropriate language modular design. This is not different in the case of bottom-up language product lines. The first challenge of a re-engineering process is to recover the definition of a set of language modules that capture the commonalities and variability existing between the DSLs under study. Indeed, recovering modular designs is one of the main subjects of study when reverse engineering software product lines from software products.

This particular issue has not received proper attention in the literature. Indeed, the only approach that has proposed something in this direction is the one presented by Kuhn et al., [15]. In that work, each language construct is defined as a language module. That means that the reverse engineering process will result in a language product line containing as many features as language constructs exist in the input DSLs.

Note that this approach permits to exploit the variability in the language product line in

the sense that the level of granularity is quite high. Hence, language designers can make decisions with an important level of detail. However, the complexity of the product line might increase unnecessarily. From the point of view of language users, there are clusters of language constructs that always go together thus separation is not needed. For example, in our illustrative scenario on state machines, the concepts of `StateMachine`, `State`, and `Transition`, go always together since they correspond to a commonality of all the input DSLs. Separating these constructs in different features is not necessary in this case and this increases the complexity of the variability models. This can be a real issue if language designers decide to apply automatic analysis operations on those models.

Limitations on Synthesis of Multi Dimensional Variability Models

The synthesis of variability models has been largely studied in the literature. Some of those approaches have been adapted for the particular case of variability in the context of language product lines engineering. We mentioned those works along this literature review.

There is, however, an important limitation in those approaches. Although at the modeling level, feature models have shown their capabilities to represent multi dimensional variability and it has been validated for language product lines, there is not support for effectively reverse-engineering such multi dimensional variability in the language product lines. Indeed, the solution provided by current approaches is to synthesize variability models where each feature capture both the abstract syntax of the language constructs and their semantics. Using this strategy, a language construct that has different semantics interpretations is represented as two language features. Those features have the same abstract syntax (a repeated definition of the specification) and their corresponding semantics.

The problem with this strategy is that it couples abstract syntax variability with semantics variability, which limits multi-staged configuration. The scenario in which language designers configure only the abstract syntax, and final users configure their semantics is not supported since the configuration of the semantics depends also to configure a segment of the abstract syntax. We claim that, in order to support multi staged configuration, the abstract syntax variability should be defined separately from the semantic variability.

3.5 Threats to Validity

In this section, we discuss the possible threats to validity of the literature review introduced in this chapter. Concretely, we discuss three of the different types of validity proposed by Wohlin et al. [105]: construct validity, internal validity, and external validity.

Construct validity. Construct validity evaluates the quality of the methodology followed to obtain the income of the study. In the case of this literature review, this process corresponds to the research method we used to obtain the set of articles included in the discussion (described in Section 3.2). Does our study include all the relevant articles existing in the literature on language product line engineering?

To answer this question, we used a three-fold strategy intended to validate our research method. Such strategy was explained at the end of Section 3.2; it includes the participation of experienced researchers in the area. Despite such a rigorous process, our methodology could miss some relevant articles. This limitation comes up from four aspects. First, the automatic search phase of our process is based on arbitrary strings. Those strings were carefully selected through a criteria; however there is still a risk of missing some papers that do not fit in the search expression. Second, we performed the automatic search in a set of four digital libraries while excluding other potential sources such as Google Scholar. This decision is also well argued in Section 3.2; however those sources might also contain relevant articles that we are missing in this study. Third, both selection and discarding processes were performed by only one of the authors of this article. Hence, potential errors might appear at the moment of applying the selection/discarding criteria specially due to the large amount of articles provided by the automatic search. Finally, the discarding and selection process where conducted by reading only titles, abstract, introductions, and conclusion sections. This decision permitted us to deal with the large amount of articles resulting from the automatic search; however, we might be missing some details in the body of the articles that contribute to the discussion.

Internal validity. Internal validity concerns the process used to extract the results from a given income. In the case of this literature review, evaluating internal validity corresponds to evaluate whether our results are consistent with respect to the actual content of the articles included in the discussion.

The most important risk in terms of internal validity for this study is the low level of agreement in terms of vocabulary used in the articles. The same word is often used in different articles to refer to different concepts. To deal with this issue, we introduced a background section intended to unify the vocabulary that we used in this paper. Besides, we provide equivalences of vocabulary when needed to clarify the concepts (such as the one presented in Table 3.3.1). Still, the vocabulary that we are using might be conflictive with respect to pre-conceived ideas of some of our readers.

External validity. External validity evaluates whether the results obtained in the study can be generalized to closely related areas of endeavor. In the case of this literature review, the evaluation of the external validity corresponds to verify whether our methodology and results can be generalized to other areas of application of Software Product Lines Engineering. For example, we might ask us if the life-cycle of *Language* Product Lines can be generalized to describe the life-cycle of other product lines such as *Games* Product Lines [106] or *Embedded Systems* Product Lines [107].

The study provided in this article is based on an abstraction of the generalities of Software Product Lines Engineering. In section 2.2, we introduced a general life-cycle for software product lines while doing abstraction of the type of products. This permitted to map such a life-cycle to the case in which those products are DSLs. However, while doing such a mapping, we realized that there are certain particularities to consider coming from

the specificities of DSLs with respect to other types of software produces. In that sense, we claim that, even if some coarse grained phases of the life-cycle of a software product line can be generalized, there are still some important details to take into consideration coming up from the particularities of each type of software product.

3.6 Summary

This chapter reports on an effort for organizing the literature on language product line engineering. It provides a detailed study that shows how the ideas on software product lines have been adapted to support the emerging notion of language product lines. To this end, we conducted a literature review, which permitted to select an important amount of articles, that we analyze in a systematic way.

Part II

Contributions

Chapter 4

Foreword to the Contributions

In the last chapter, we presented a literature review on language product lines engineering. It closes with a discussion on the open issues that we found as part of the analysis of the results. In this chapter, we use the definitions and open issues introduced in the literature review to clearly point out the scope of the thesis. Such scope is presented at two levels. On one hand, we present the *scientific scope*, which corresponds to a subset of open issues that we address. On the other hand, we present the *technological scope*, which corresponds to the technological space in which our ideas can be applied.

4.1 Scientific Scope: Addressed Open Issues

This thesis aims to contribute to the field of language product lines engineering by addressing a subset of the open issues identified before. This subset of open issues are shown in Fig. 4.1. Our selection of the open issues is guided by two considerations. First, we aim to contribute in both top-down and bottom-up language product lines. This decision allows us to cover a larger spectrum of development scenarios. Second, we focus on the open issues regarding the design and implementation phases. Evolution and testing are quite complex problems that require a deep reflection. We consider that all those problems are hard to address in a single thesis at the same time. The reminder of this section is dedicated to give a brief summary of how we deal with those open issues.

Contributions on top-down language product lines. Chapter 5 presents our contributions in top-down language product lines. In this context, we are focused on extending current meta-languages in order to improve the capabilities in terms of languages modularization. In particular, we aim to provide a modularization approach which not only supports all the language modularization scenarios (i.e., extension, restriction, aggregation, unification, and extension composition), but also addresses the software modularization principles (i.e., independent development, information hiding, and substitutability). The inclusion of these software modularization principles represents our delta with respect to the state of the

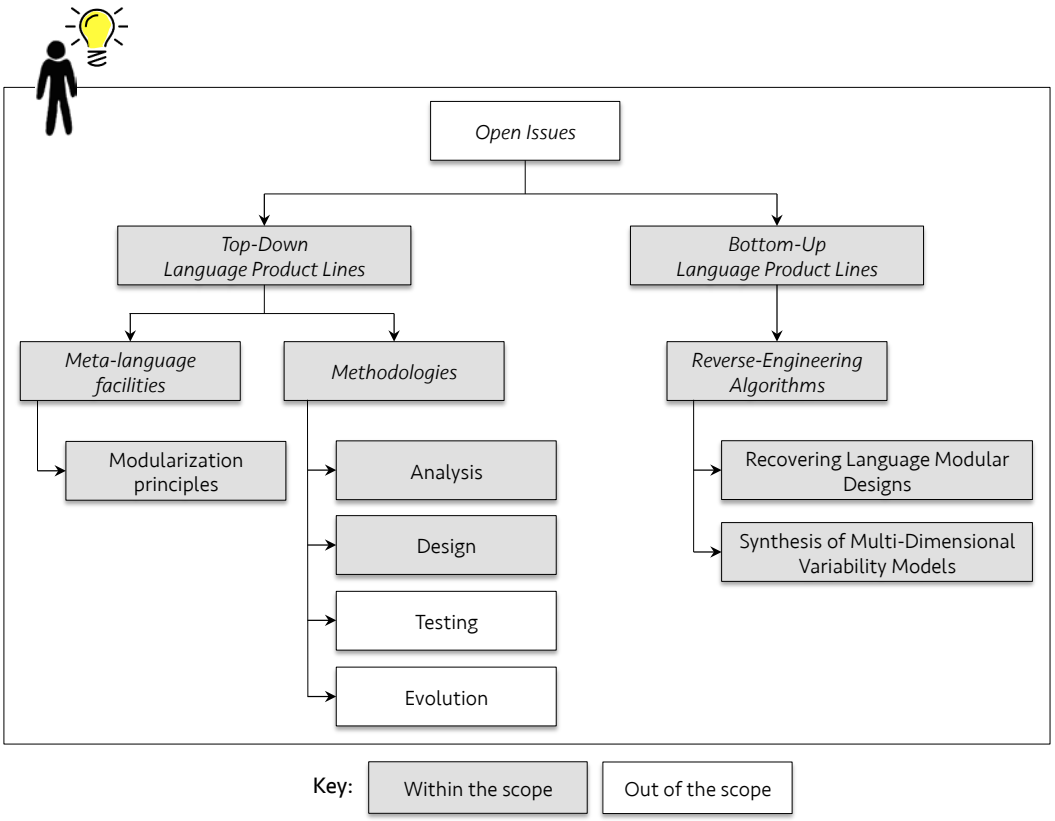


Figure 4.1: Scientific scope of the thesis: Addressed open issues

art. We also present an approach to manage languages variability. It proposes the combination of two formalisms –i.e., feature modeling and orthogonal variability modeling– to facilitate variability modeling as well as multi-staged configuration.

We close our contributions in top-down language product lines with a set of methodological insights to facilitate the phases of analysis and design of language product lines. These insights can be used in the planning of the daily activities of a team of language designers.

Contributions on bottom-up language product lines. Chapter 6 presents our contributions on bottom-up language product lines. In particular, we introduce a reverse engineering technique to automatically build up a language product line from a set of existing DSL variants. Our technique consists of an algorithm to recover a language modular design and a mechanism to synthesize language variability models that enable configuration and derivation of concrete DSLs. These variability models include both abstract syntax and semantic variability. Those models are generated in a separate way, thus answering the claim we introduced before in this regard.

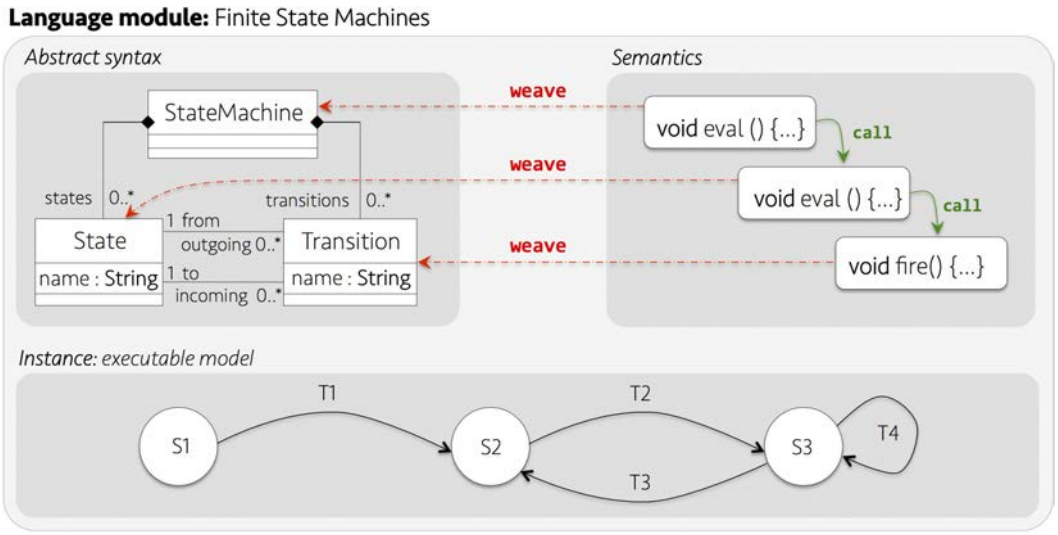


Figure 4.2: A simple DSL for finite state machines

4.2 Technological Scope: Supported Technological Space

Technological spaces to build DSLs are quite diverse. This diversity is evidenced not only in terms of the amount of existing technological spaces, but also in the differences among them. Although the purpose of all those technological spaces is the construction of DSLs, the concepts and definitions they use is quite different; for instance, the definition of a grammar-based DSL whose semantics is specified denotationally by means of attribute grammars, is quite different from the definition of a metamodel-based DSLs where the semantics are specified in a model-to-model transformation using the ideas of Model-Driven Engineering. This diversity makes it difficult to generalize the ideas of software language engineering to all the technological spaces. As a result, we need to clearly specify the scope of this thesis by indicating the technological space we address.

All the ideas that we will present in the remainder of this thesis are scoped to executable domain-specific modeling languages (xDSMLs) where the abstract syntax is specified through *metamodels*, and the dynamic semantics is specified operationally as a set of *domain-specific actions* [108]. Domain-specific actions are java-like methods that introduce behavior in the meta-classes of a given metamodel. Such injection is performed via weaving as the same as in aspect oriented programming [69]. Concrete syntax, and hence concrete syntax variability, are out of the scope of this thesis.

Fig. 4.2 illustrates this type of DSLs through a simple example on finite states machines. In that case, the metamodel that implements the abstract syntax contains three metaclasses: *StateMachine*, *State*, and *Transition*. There are some references among those metaclasses representing the relationships existing among the corresponding language constructs. The domain-specific actions at the right of the Figure 4.2 introduce the operational semantics to

the DSL. In this example, there is one domain-specific action for each metaclass. The interactions among domain-specific actions can be internally specified in their implementation by means of the *interpreter pattern*, or externalized in a model of computation [108].

Chapter 5

Facilities to Support Top-Down Language Product Lines

The objective in top-down language product line engineering is to transform the domain knowledge owned by domain experts into a language product line that permits to easily prototype diverse DSL variants. Each variant is intended to address the needs of a group of final users. In this chapter we present some extensions to the existing meta-languages to facilitate the construction of language product lines. Concretely, those facilities support language modularization and variability management. Besides, we introduce some methodological insights that guide language designers through the development process.

5.1 Meta-language Facilities for Language Product Lines

5.1.1 Supporting Languages Modularization

As aforementioned, current approaches on languages modularization focus on supporting languages modularization scenarios for different technological spaces. However, they fail in satisfying three main principles largely discussed in the literature on software modularization, namely: information hiding, independent development, and substitutability. In this section we revisit modularization of DSLs by proposing an approach that overcomes this limitation. Our approach covers both abstract syntax and semantics.

Supporting the Software Modularization Principles

In order to support software modularization principles in languages modularization, we propose the definition of certain *language interfaces*. In particular, we adapt the classical notions of required vs. provided interfaces to the construction of DSLs. As shown in Fig. 5.1, each of the elements of the approach is intended to support one of the software modularization principles.

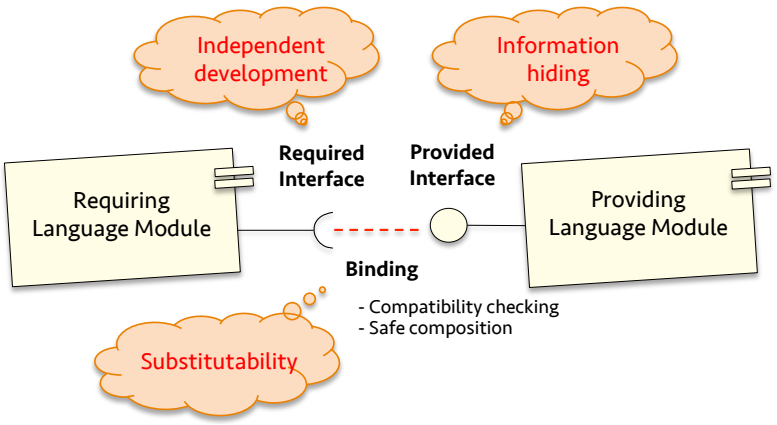


Figure 5.1: Interfaces for languages modularization

Required interfaces to support independent development. The purpose of a required interface is to support independent development of language modules. In that sense, a required interface is a mechanism that allows language designers to declare the needs that a language module has towards other modules while assuming that their needs will be eventually fulfilled. Suppose for example the development of a language module for finite state machines. This language module needs some additional abstractions such as constraints to express guards in the transitions. Using a required interface, language designers can declare those needs as a set of required constructs (e.g., *Constraint*) and focus on the definition of the constructs which are proper to finite state machines (e.g., *State*, *Transition*, *Triggers*).

The needs of a language module can be materialized in the form of required constructs. In assuming so, a module's specification would be composed of a set of actual constructs, which are being implemented by the module; and a set of required constructs, which represent needs to other modules. This approach results useful to support the modularization scenario called aggregation where the needs of language modules are entire constructs implemented in foreign modules. However, a finer level of granularity might be necessary to support other modularization scenarios such as extension where the needs are not necessarily entire constructs but finer elements such as properties or operations.

Based on this reasoning, we propose a mechanism to enable the capability to distinguish whether a given language specification element (i.e., meta-class, property, operation, parameter, enumeration, etc) corresponds to an actual implementation or a required declaration. The proposed mechanism is an extension to the EMOF meta-language that introduces the notion of *virtualization* (see appendix A, section A.1 for further details). Using this extension, language designers can define virtual specification elements expressing needs of the module. Non-virtual specification elements are actual implementations of the language module. The required interface of a language module is the set of virtual elements it contains within its specification.

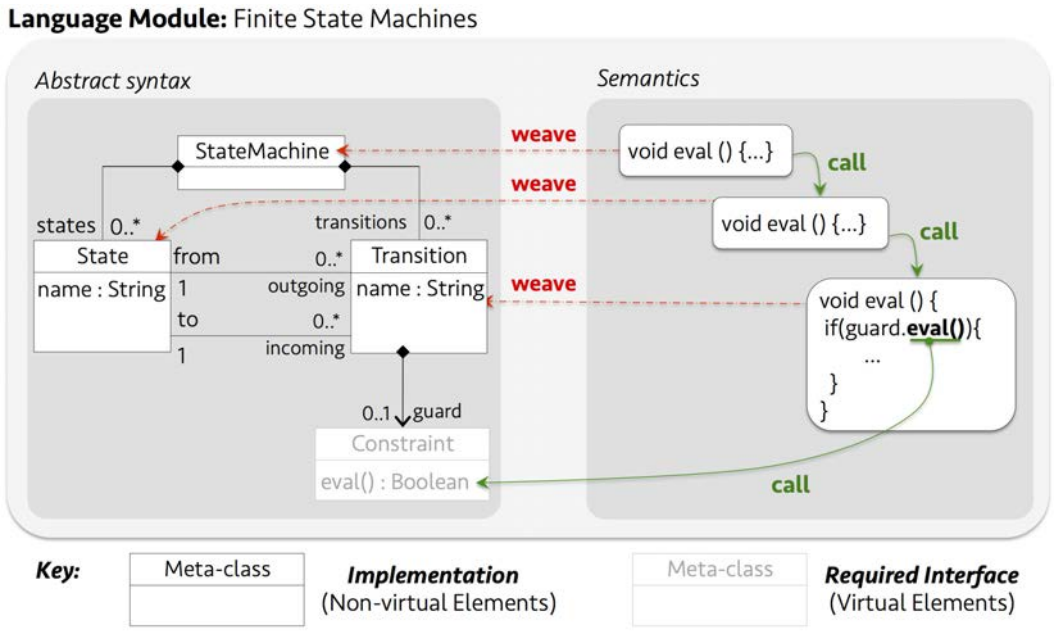


Figure 5.2: Example of the use of required interfaces

Fig. 5.2 illustrates the use of required interfaces through the example introduced before regarding a language module for finite state machines that requires a constraint language. In that example, the constructs proper the state machines are non-virtual elements since they correspond to actual implementation of such abstractions. In turn, the constraints to express the guards in the transitions are expressed as a virtual construct called *Constraint* that provides a virtual operation action called *eval()* which is used in the specification of the semantics of the meta-class *Transition*.

Provided interfaces to support information hiding. The purpose of provided interfaces is to enable information hiding in modular development of DSLs. That is, to distinguish between the information that specifies the functionality offered by the language module from the information corresponding to the implementation details behind such functionality. Consider a language module that offers the capability to express and evaluate constraints. Using a provided interface, language designers can express the essential functionality of the module i.e., expression and evaluation of constraints; and hide the implementation details and auxiliary concepts needed to achieve such functionality.

To support the definition of provided interfaces, we propose to extend EMOF with the notion of *module visibility* (see appendix A, section A.2 for further details). This extension allows to classify a certain specification element as either *public* or *private* according to its nature. For example, a language designer can classify a metaclass as *public* meaning that it represents essential functionality of the module so can be used by external modules

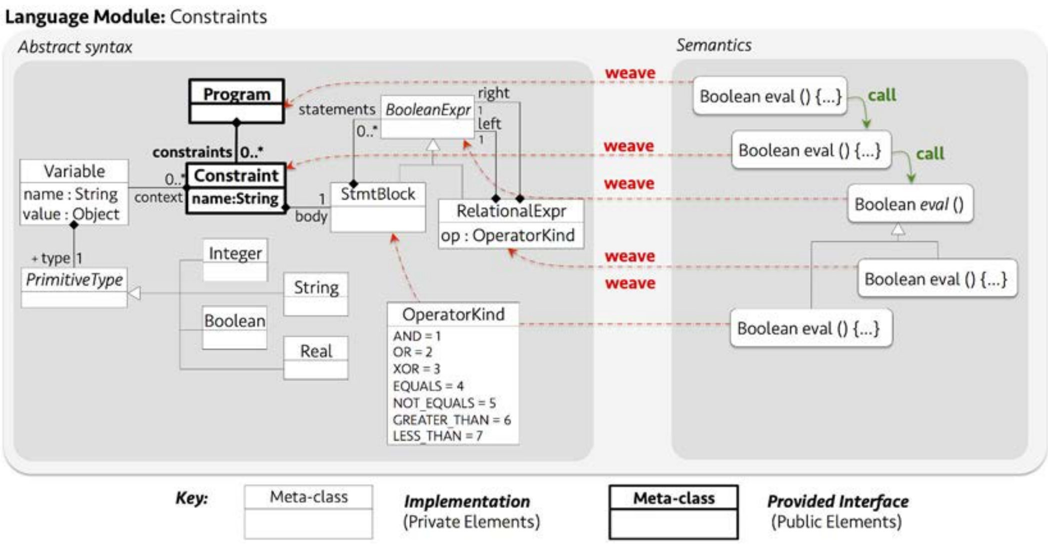


Figure 5.3: Example of the use of provided interfaces

and it belongs to the provided interface. Naturally, if the meta-class is classified as *private* it cannot be used by external modules and it cannot be considered as part of the provided interface. Note that the notion of module visibility is different from the notion of visibility already defined in EMOF. The later is associated to certain access constraints of model elements with respect to the package in which they are implemented.

Fig. 5.3 illustrates the use of provided interfaces through the example introduced before regarding the constraints module. Since the main functionality of the module is to define and evaluate constraints, the meta-classes included in the provided interface (so those one defined as *public* in terms of module visibility) are: *Program* and *Constraint* including the operations that implement their semantics. Note that the references from the metaclass *Constraint* to other metaclasses (i.e., Constraint.body and Constraint.context) should be defined as *public* if and only if they represent exposed functionality. In such a case, the target metaclasses should be also defined as public. In our example, those references are not exposed so they remain private as well as their target metaclasses.

Interfaces binding to support substitutability. Now we need to establish the way in which those interfaces interact each other at the moment of the composition. In doing so, it is important to conciliate two different (and potentially conflicting) issues. Firstly, safe composition should be guaranteed; we need to check the compatibility between a providing and a requiring language module by verifying that the functionality offered by the former actually fulfills the needs of later. Secondly, there must be some place for substitutability; the compatibility checking mechanism should offer certain flexibility that permits to perform composition despite some differences in their definitions. This is important because when language modules are development independently of each other, their interfaces and

implementations not always match [51].

To deal with the aforementioned issues, we propose an approach for compatibility checking which is strict enough to guarantee safe composition, and at the same time, it is flexible enough to permit substitutability under certain conditions. To this end, we propose to extract both required and provided interfaces in the form of *model types* [109]. The model type corresponding to the required interface contains the virtual specification elements of a language module, and the model type corresponding to the provided interface the model type contains its public specification elements. The relationship between a model type and a language module is called *implements* and it is introduced by Degueule et al. [67].

To perform compatibility checking, we use the sub-typing relationship between the model types corresponding to the provided and the required interfaces [110]. This relationship imposes certain constraints that guarantee safe composition while permitting some freedom degrees thus introducing some flexibility. In particular, under this definition of sub-typing the most obvious manner to guarantee safe composition is to check two conditions: (1) all the needs expressed in the requiring model type are furnished in the providing model type (*total* sub-typing); and (2) the two model types have exactly the same shape (*isomorphic* sub-typing). However, this definition of sub-typing also provides two dimensions of flexibility: *partial* sub-typing and *non-isomorphic* sub-typing.

The main principle behind partial sub-typing is that not all the needs expressed in the required model type must be provided by the provided one. In that case, compatibility checking corresponds to verify that the sub-set of elements that match in the model types are compatible. Then, the result of the composition is a third language module with a resulting required interface that contains those needs that have not been satisfied by the providing language module.

As an example suppose a language module for finite state machines that needs not only constraints for expressing guards in the transitions, but also action scripting constructs to express the behavior of the states. In such a case, a constraint module will fulfill the first need but not the second one. Thanks to partial sub-typing, we can perform compatibility checking only on the constructs associated to the constraints and, if they are compatible, then compose those language modules. The result will be a language module having the constructs for state machines and constraints but that still needs action scripting constructs defined in its required interface.

The principle behind isomorphic sub-typing is that the needs in the requiring interface are not always expressed exactly as the functionality offered by the providing module is expressed in the providing interface. For example, model type in Fig. 5.2 expresses the needs in terms of constraints of state machines through a class *Constraint* with an operation *eval()*. If we want to use OCL to satisfy these needs, then we will find that there is not a class constraint but *OclExpr*. Besides, the operational semantics might be implemented differently. In this case, we need an adapter that permit to find the correspondences among the elements of the model types.

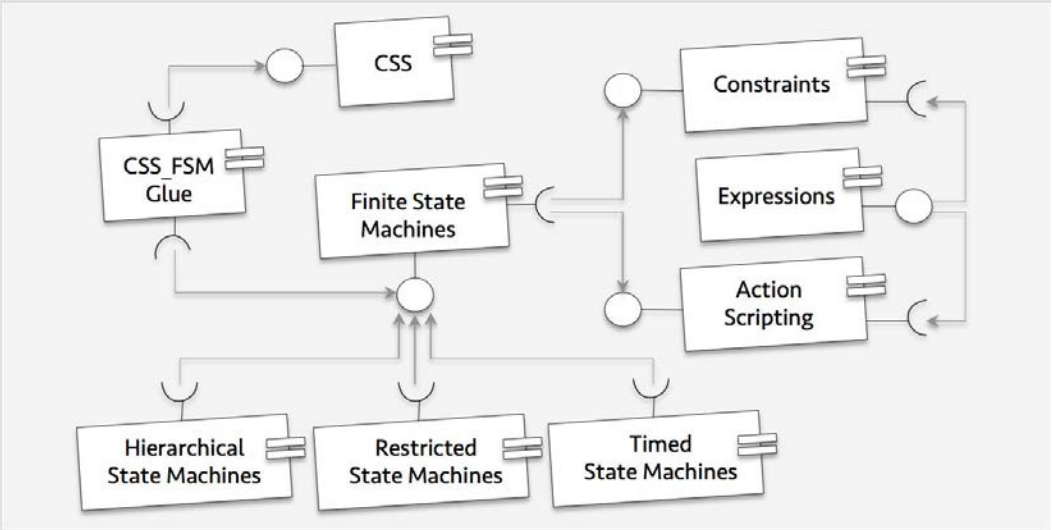


Figure 5.4: Running example: A modular DSL for finite state machines

At the implementation level, in the current state of our approach we support partial sub-typing and some particular cases of non-isomorphic sub-typing. We still need some research to fully support non-isomorphic sub-typing.

Supporting the Language Modularization Scenarios

Let us now to explain how our approach addresses the language modularization scenarios. To this end, we will use as running example the language for finite state machines that we have used so far. Fig. 5.4 summarizes such example; it is composed of several language modules with some dependencies among them. Each of those dependencies exemplifies a certain modularization scenario.

Extension. There are several ways to extend language modules. Our approach supports two of them: *sub-classing* and *open-classes* [111]. They are explained in the following.

Extension through ***sub-classing***. In this case, the extension module introduces one or more sub-classes which specialize the metaclasses defined in the base module. Consider a language module called *TimedStateMachines* that extends a base module called *FiniteStateMachines* with time counters on the transitions. The *TimedStateMachines* module introduces a new metaclass called *TimedTransition* which is a sub-class of the *Transition* metaclass. This new sub-class implements the structural features (i.e., properties) and behavior (i.e., operations) needed to support the time counters' functionality. To implement this particular case of extension by using our approach, a language designer should:

(1) Add the *Transition* metaclass to the provided interface of the *FiniteStateMachines* –i.e., define its module visibility as *public*–, thus indicating that it can be extended

by foreign modules. The provided interface should also include all the properties and operations of the *Transition* metaclass that represent its functionality.

(2) Add the *Transition* metaclass to the required interface of the *TimedStateMachines* module –i.e., define it as a *virtual* metaclass–, thus indicating that it corresponds to a requirement that should be eventually fulfilled by a certain base module. This declaration should include not only the metaclass itself, but also the properties and operations needed to implement the extension. For example, if the *TimedTransition* metaclass is intended to override the *fire()* operation, then it should be declared in the required interface.

(3) Define the *TimedTransition* metaclass as a subclass of the *Transition* metaclass defined in the required interface. Then, implement all the functionality corresponding to times counters.

*Extension through the **open-class** pattern.* In the case of extension through the open-class pattern [111], the idea is to enhance a meta-class of the base module by directly injecting new properties and/or operations without adding a new sub-class. The base class is actually re-opened and modified according to the needs of the requiring module. Suppose for example a language module called *HierarchicalStateMachines* that extends the *FiniteStateMachines* module to support hierarchical states –i.e., states that contain other states–. In that case, a new containing reference should be added to the *State* meta-class indicating that it can have sub-states. In addition, the operation *eval()* should be re-defined in order to include the new behavior. To implement this particular case of extension by using our approach, a language designer should:

(1) Add the *State* meta-class to the provided interface of the *FiniteStateMachines* module. Include the properties/operations that represent its functionality.

(2) Declare the *State* meta-class in the required interface of the *HierarchicalStateMachines* module. This declaration should include not only the meta-class itself, but also the properties and operations needed to implement the extension. In particular, because the behavior of the meta-class *State* should be re-defined, the *eval()* operation should be included in the required interface.

(3) Complete the virtual meta-class called *State* with the new functionality corresponding to hierarchical states. The language designer should add a non-virtual containing reference to the *State* meta-class representing the sub-states. Besides, the the *eval()* operation should be overridden with the enhanced behavior. As we will explain later in this paper, in our approach we provide a mechanism based on annotations to override operations defined in the required interface.

Note that extending a language module through the open-class pattern results in virtual meta-classes containing non-virtual elements. After composition, the resulting meta-class will be a unique *State* meta-class that contains all the properties already defined by the base module as well as the elements injected in the extension one.

Restriction. Language modules restriction can be achieved in several ways. For example, language designers might abuse of the open-class pattern to access an existing metaclass of the base module and then remove some of its elements[1]. The problem with this approach is that safe composition is not longer guaranteed; deletion of specification elements can introduce some conflicts when composing more than two modules.

Suppose for example that the we want to compose the *FiniteStateMachines* module with *HierarchicalStateMachines* and the *RestrictedFSM* –the later is a restriction to the *FiniteStateMachines* intended to remove the *Fork* and *Join* pseudostates–. In that scenario, after the composition of the modules we will have a compilation error in the implementation of the *HierarchicalStateMachines* if it uses the *Fork* and *Join* pseudostates; those pseudostates will not longer exist. As a matter of fact, another problem of allowing deletions of specification elements in a given module is that the results of the composition might be different depending on the order in which the composition is performed.

Based on this reasoning and inspired by the work of Vacchi et al [86], we decide to use additive extensions to support restriction of language modules. We propose to support restriction by enhancing the invariant of the metaclasses thus enforcing the validation phase of the language in such a way that all the programs that use the disabled constructs will be rejected. In our example about disabling pseudostates, all the state machines containing *Fork* and *Join* will be rejected in the validation phase. Note that in this case there will be no conflicts at the moment of the composition because the psedostates still existing in the definition of the modules. To implement this particular case of restriction by using our approach, a language designer should:

(1) Add the *StateMachine* meta-class to the provided interface of the *FiniteStateMachine* module. All the elements relative to the pseudostates we want to remove should be also included: the corresponding enumeration or meta-classes.

(2) Add the *StateMachine* metaaclass to the required interface of the *RestrictedFSM* module.

(3) Enhance the invariant of the *StateMachine* meta-class by adding a new constraint that states that there cannot exist pseudostates of type *Fork* or *Join*. In doing so, we are using the open-class pattern in an additive way to implement a restriction of the *FiniteStateMachines* module.

Aggregation. As aforementioned, aggregation is the modularization scenario where a requiring language module uses some elements specified in a providing one. As an example of aggregation, we can retake the situation explained in section 5.1.1 when explaining the notions of required and provided interfaces. There is a language module called *FiniteStateMachines* that implements basic functionality to express finite state machines, and which depends on two other modules: a first one providing constraint language and a

[1] We use the expression "*abuse* of the open-class pattern" because such pattern is typically used to add new elements in the open-class; removals are often forbidden.

second one for action scripting. There are two modules i.e., *Constraints* and *ActionScripting* that provide each of those functionalities but that, in turn, depend on an Expressions module that provides the basic functionality for specify arithmetic and relational expressions as well as to declare variables. To implement this particular case of aggregation by using our approach, a language designer should:

(1) Add the *Constraint* meta-class to the provided interface of the *Constraints* module.

(2) Add the *Program* meta-class to the provided interface of the *ActionScripting* module.

(3) Declare the meta-classes *Constraint* and *Program* in the required interface of the *FiniteStateMachine* module. This declaration should include not only the meta-classes themselves, but also the required properties and operations. In particular, in this case we need not only the meta-classes, but also their operational semantics implemented in the *eval()* operation of each of them.

(4) Implement the *FiniteMachinesModule* with the abstractions needed to support the definition of state machines by referencing the meta-classes *Constraints* and *Program* when needed.

Unification. Language modules unification is often achieved through the creation of glue modules that implement the behavior of the composition of two initially independent language modules. In order to address the example for unification regarding the integration of finite state machines to CSS to specify final user interfaces, a language designer should:

(1) Create a glue-code module (we called it FSM-CSS).

(2) Add the meta-classes *StateMachine* and *State* from the *FiniteStateMachine* module to the required interface of *FSM-CSS*.

(3) Add the meta-class *StyleSheet* from the *CSS* module to the required interface of *FSM-CSS*.

(4) Create a meta-class called *StyledStateMachines* in the module *FSM-CSS* as sub-class of *StateMachine*. A styled state machine implements the navigation of a web-site.

(4) Create a meta-class called *StyledState* in the module *FSM-CSS* that as a sub-class of *State*. Then, create a containment reference from *StyledState* to *StyleSheet* to indicate that each state of the state machine represents one page in the web-site, and that the corresponding style sheet implements the page's style.

Extension composition. To support extension composition, we need to provide a mechanism to specify a set language modules in terms of their implementation artifacts and interfaces. Doing the parallel with software architecture, what we need to support extension composition is an Architecture Description Language (ADL) [112]. In our approach, we use Melange [67] to this end. Melange is a tool-supported meta-language that can be

used to model the relationships in the large of a set of artifacts associated to the construction of software languages. The details of the use of Melange to this end will be introduced later in this document when we explain the implementation details of our approach.

Language modules composition

So far, we have explained the way in which language designers can define a set of interdependent language modules. we explained the way in which these dependencies can be expressed to support language modularization scenarios and respecting certain software modularization principles. Now, we explain the composition that permits to unify a set of language modules to produce a unique DSL specification.

Language modules composition starts with a preliminary phase of compatibility checking, which guarantees safe composition. It is based in checking the sub-typing relation between the model types representing required and provided interfaces of the involved language modules. Once the compatibility is correctly checked, the next step is to compose the language modules to integrate their functionality i.e., the needs of the requiring module are fulfilled with the services offered by the provided one.

In our approach, this composition is performed in two phases. First, there is a matching process that identifies one-to-one matches between virtual and public elements from the required and provided interface respectively. This match can be identified automatically by comparing names and types of the elements (where applicable). However, the match can be also specified manually in the case of non-isomorphism. Once the match is correctly established, the composition process continues with a merging algorithm that replaces virtual elements with public ones. That means also to replace all the possible references existing to the virtual element to point out to the corresponding public element.

Once the process is finished, we re-calculate both provided and required interfaces. The provided interface of the composition is re-calculated as the sum of the public elements of the two modules under composition. In turn, the required interface of the composition is re-calculated as the difference of the required interface of the required module minus the provided interface of the providing module.

5.1.2 Supporting Languages Variability Management

As we said earlier, the challenge towards representing the variability existing in a language product line is that such variability is multi dimensional. Because the specification of a DSL involves several implementation concerns, then there are several dimensions of variability that we must manage: abstract syntax variability, concrete syntax variability, and semantic variability [55, 56]. As the same as our approach to language modularization, our approach to variability management is scoped to abstract syntax and semantics; concrete syntax –and hence, concrete syntax variability– is not being considered in the solution.

Modeling multi-dimensional variability

A solution to represent abstract syntax variability and semantic variability should consider two main issues. Firstly, the definition of the semantics has a strong dependency to the definition of the abstract syntax –the domain-specific actions that implement the semantics of a DSL are weaved in the meta-classes defined in the abstract syntax–. Hence, these dimensions of variability are not isolated each other. Rather, the decisions made in the configuration of the abstract syntax variability impact the decisions that can be made in the configuration of the semantic variability.

The second issue to consider at the moment of dealing with language variability management is that a semantic variation point might be transversal to several meta-classes. Moreover, if the involve meta-classes are introduced by different language modules in the abstract syntax, then the semantic variation point depends of two features. As a result, the relationship between a feature in the abstract syntax and a semantic variation point is not necessarily one-to-one.

Currently, we can find several approaches to support multi dimensional variability (e.g., [60]). Moreover, in chapter 3 we show how those approaches have been applied concretely to language product lines. The most common practice is to use feature models to represent all the dimensions of variability. Each dimension is specified in a different tree and dependencies among decisions in those dimensions are expressed as cross-tree constraints. In this thesis, we propose a different approach based on the combination of feature models with orthogonal variability models. Feature models are used to model abstract syntax variability and orthogonal variability models are used to model semantic variability.

Fig. 5.5 illustrates our approach. At the top of the figure, there is a feature model in which each feature represents a language module. As aforementioned, each language module is composed of a metamodel and a set of domain specific actions. Hence, such a feature model is enough for language product lines where there is not semantic variability i.e., each language module has only one set of domain specific actions. Differently, when there are one or more language modules containing several sets of domain specific actions, then we have semantic variability that must be represented in the variability model. To represent such a variability, we include an orthogonal variability model as illustrated at the bottom of Fig. 5.5 which contains a variation point for each feature that represents a language module with more than one set of domain specific actions.

Why orthogonal variability models? An inevitable question that we need to answer at this point is: why we use orthogonal variability models instead of using feature models as proposed by current approaches? The answer to this questions is three-fold:

(1) The structure of orthogonal variability models is more appropriated. As explained by Roos-Frantz et al. [113], feature models and orthogonal variability models are similar. However, they have some structural differences. One of those differences is that whereas a feature model is a tree that can have many levels, an orthogonal variability model is a set of

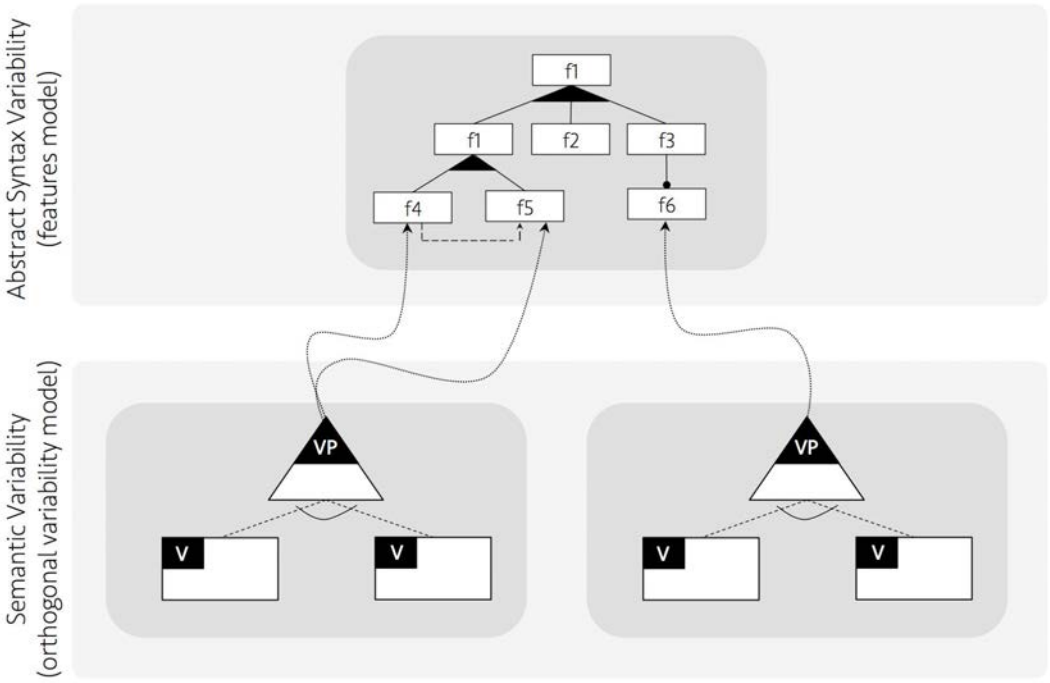


Figure 5.5: Approach to represent multi-dimensional variability in language product lines

trees each of which has two levels. Each tree represent one variability point and its children represent variants to that variation point.

Semantic variation points are decisions with respect to a particular segment of the semantics of a language. Although those decisions can have some dependencies among them, they can hardly be organized in a hierarchy. Indeed, we conducted an experiment where we use feature models to represent semantic variation points, and we always obtained two-level trees: the first level corresponds to the name of the variation point and its children represent the possible decisions. This fact suggests that orthogonal variability models are more appropriated than feature models to represent semantic variability.

(2) The meaning of orthogonal variability models is more appropriated. According to [18], a language feature is a characteristic provided by the language which is visible to the final user. This definition can be associated abstract syntax variability and the use of feature models can be appropriated to represented it. All the approaches on language product line engineering use feature models to this end showing that it is possible and appropriated.

The case of the semantic variability is different. A semantic decision is not a characteristic of a language that we can select or discard. The semantic of a DSL should be always specified if the DSLs is intended to be executable. Rather, a semantic decision is more a variation point that can have different interpretations captured as variants. This vocabulary

fits better in the definitions provided by orthogonal variability models. More than features, we have variation points and variants, which also suggest that the use orthogonal variability models is more appropriate to represent semantic variability.

Multi-staged Language Configuration

There are two issues to consider to support configuration of DSL variants in language product line engineering. First, the multi-dimensional nature of the variability in language product lines, supposes the existence of a configuration process supporting dependencies between the decisions of different dimensions of variability. For example, decisions in the abstract syntax variability may impact decisions in semantic variability. Second, language product lines often require multi-staged languages configuration. That is, the possibility of configuring a language in several stages and by different stakeholders.

Multi-staged configuration was introduced by Czarnecki et al. [61] for the general case of software product lines, and discussed by Dinkelaker et al. [62] for the particular case of DSLs. The main motivation to support such functionality is to transfer certain configuration decisions to the final user so he/she can adapt the language to exactly fits his/her needs [62]. In that case, the configuration process is as follows: the language designer provides an initial configuration. Then, the configuration is continued by the final user that can use the DSL as long as the configuration is complete. In doing so, it is important to decide what decisions correspond to each stakeholder.

Suppose the scenario introduced in Fig. 5.6 where the language designer is responsible to configure the abstract syntax variability whereas the language user is responsible to configure the semantics. When the language designer finishes its configuration process, the orthogonal variability models will be available so the final user can perform the configuration of the semantics. This orthogonal variability model will only include the variation points that are relevant to the features included in the configuration of the abstract syntax. Moreover, because each of the semantic variation points are represented separately in a different tree, then we can imagine a scenario where the language designer is able to configure not only the abstract syntax but also some semantic variation points, and then delegate to the final user only the decisions that he/she can take according to its knowledge.

Running example

Let us now to illustrate our approach in language variability management with the example of the state machines. Fig. 5.7 summarizes the situation. In that case, we have a language product line for state machines that offers a well-defined set of language features in for the abstract syntax, and two semantic variation points. For the case of the abstract syntax, there is a mandatory feature called Core that contains the main constructs needed to express finite state machines i.e.., *StateMachine*, *State*, *Transition*, *Region*, *Trigger*, and so on. Then there is a set of features representing the pseudosates. The initial pseudo state is mandatory, all the others are optional and should be included or not according to the level of the

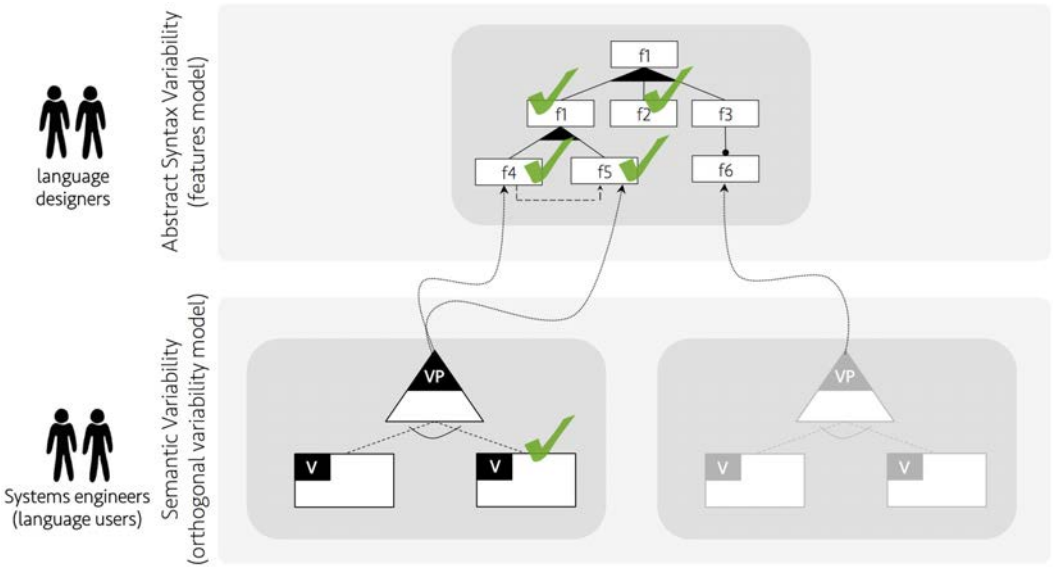


Figure 5.6: Approach to support multi-staged configuration of language product lines

expressiveness required for the DSL. We have also a feature called *TimedTransitions* that introduces the notion of time counters in the transitions. Finally, there is a feature called *HierarchicalStateMachine* that introduces support for composite states.

In addition of the abstract syntax variability, the language product line in our example also contains some semantic variability. In particular, there are two semantic variation points both of them deeply explained by Crane et al. [59]. The first one introduces two different ways of attending simultaneous events in a state machine. It is possible to actually support this simultaneous events in such a way that in a same step the state machine is reacting to two or more different triggers. The other possibility is to prohibit this behavior and comply the run-to-completion policy that states that only one event can be attended in one step. So, it does not matter if two events arrive at the same time to the state machine, there will be always a sequential reaction.

The second semantic variation point emerges as a result of selecting the feature called *HierarchicalStateMachines*. In such a case, it is possible to have conflicting transitions since, in the same step, many hierarchical states are activated. All the transitions which are outgoing the hierarchical states and that are associated to the same trigger are considered as conflicting. The first possibility to solve these conflicts is to fire the transition outgoing the state at the top of the hierarchy. The second possibility is to fire the transitions outgoing the state at the bottom of the hierarchy.

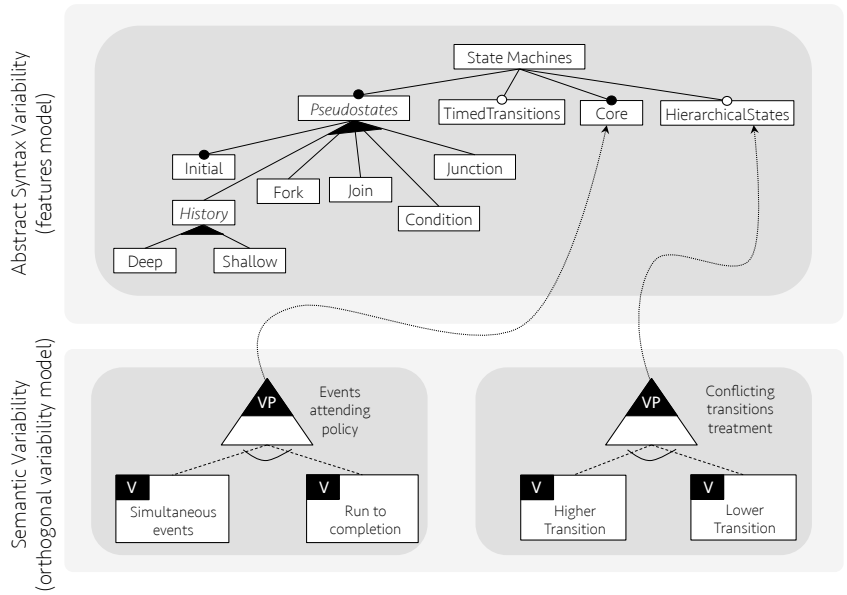


Figure 5.7: An example of multi-dimensional variability in language product lines

5.2 Methodologies for Top-Down Language Product Lines

As aforementioned, for the particular case of top-down **L**PLE, the development process begins with the domain engineering phase. Once it is completed, and the language product line is implemented, then the process moves on to the application engineering phase. As the reader might imagine, the domain engineering phase in **L**PLE is a complex development activity. Like during domain analysis in classical development of DSLs [114], the success of this phase depends on the expertise and the experience of domain experts and language designers. Whereas domain experts must own a deep understanding of the domain and solid communication skills, language designers should be able to filter relevant information from the domain and transform it into an appropriate design for the language product line.

Typically, the complexity of software development activities is addressed by means of well-defined methodologies that guide software developers through a process which is clearly defined in terms of key activities, inputs and outputs. The objective is to systematize the development activities in such a way that they can be monitored, evaluated, and continuously improved. The success of those methodologies relies on an equilibrium between *rigor*, which enables systematic development [115], and *flexibility*, which permits adaptation of the methodology to the particularities of each development process and its participants [116].

This section provides a methodological toolkit to support the domain engineering phase in top-down **L**PLE. It is meant to be rigorous enough to enable systematical development during domain engineering, and (at the same time) flexible enough to be easily adapted

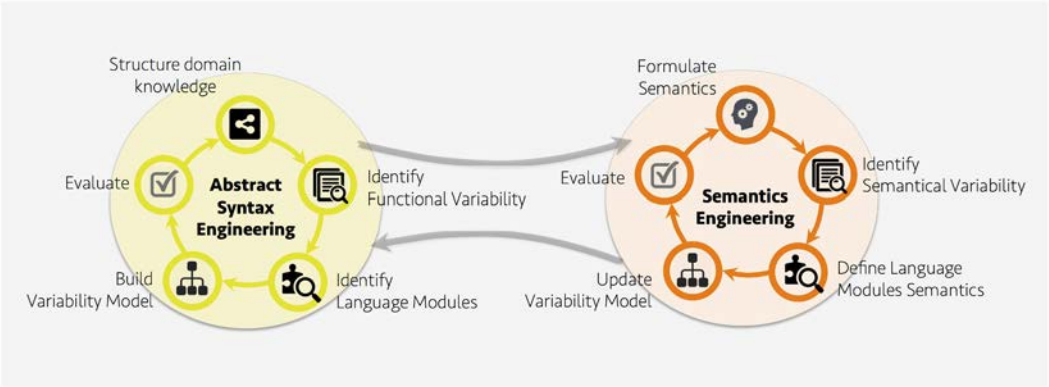


Figure 5.8: Staged process for domain engineering of language product lines

to the particularities of specific language development projects. In this methodological toolkit, we propose a set of key activities that guide language designers during the domain engineering phase. Each activity is specified in terms of input and outputs and provides some insights that help language designers during the process.

Our process, which is illustrated in Figure 5.8, is composed of two coarse-grained phases each of which is focused on engineering one implementation concern. The process starts with the engineering of the abstract syntax, then it goes through the engineering of the semantics. It should be noted that the proposed process is not monolithic. Rather, it proposes an iterative fashion to perform the activities thus favoring incremental implementation and continuous improvement. Besides, language designers can come back from the semantics engineering to the abstract syntax engineering (and vice-versa) if during the formulation of the semantics they realize that there are some missing elements in the abstract syntax. The reminder of this section is dedicated to deeply explain the process.

5.2.1 Abstract Syntax Engineering

The first activity of the domain engineering process is dedicated to the abstract syntax of the language product line. The input of this activity is the domain knowledge owned from domain experts and captured in existing DSLs targeting similar domains. The output of the activity is the definition of all the common assets of the language product line that are related to the abstract syntax. Hence, the main challenge during this phase is to transform domain knowledge (usually unstructured and vaguely described) into well-defined models conforming to some technological space appropriated to describe abstract syntax. To do so, language designers must complete a set of activities –described below– focused on structuring and formalizing the domain knowledge, identifying functional variability, and specifying language modules and variability models for abstract syntax.

Structure Domain Knowledge

The first step in abstract syntax engineering is to extract and structure the domain knowledge. At this point, language designers should interview domain experts and to explore potentially related DSLs in order to identify the concepts that are relevant for the domain targeted by language product line under construction. The relationships existing among the domain concepts should be also identified. Usually, this process is termed domain analysis [114, 117] which produces a *domain model* that will be the backbone of the rest of the domain engineering phase [118].

There are two important aspects to consider while building a domain model. On one hand, the domain model will constitute a communication bridge between domain experts and language designers. Then, it should be expressed in a formalism simple enough so it can be understood by domain experts –which in most of cases have no computer science background– and formal enough so it does not leave place for ambiguities.

On the other hand, the definition of the domain model should be as most precise as possible. That means that the vocabulary used to name the concepts must clearly reflect the elements of the domain they refer to. Besides, the relationships between the concepts should be correctly named and fully specified in terms of multiplicity and direction.

Using class diagrams to structure domain knowledge. There are very few modeling languages which are simple and formal at the same time. However, class diagrams are becoming a usual choice to this end; several methodologies for domain analysis use class diagrams as modeling formalism (e.g., [119, 120]).

In the case of language product lines, the use of class diagrams presents interesting advantages to express the abstract syntax. They can be easily mapped to metamodels, and with some effort, they can be also mapped in BNF-like grammars. Hence, we propose to use class diagrams as modeling languages for structuring domain knowledge. If the domain experts are reluctant to use class diagrams, ontologies can be used to communicate with domain experts, and lated transformed to class diagrams as proposed by Tairas et al. [117].

Consider for example the construction of a language product line of DSLs for finite state machines. The objective of the product line is to provide diverse DSLs for state machines that can be configured according to the needs of a particular system. To build this language product line, language designers interview the systems engineers and capture the needs they have with respect the state machines. In addition, there are several specification of state machines languages in the literature (e.g., UML state diagrams [121], Rhapsody [122], and Harel's state charts [123]) that can be used to enhance the domain knowledge.

Figure 5.9 proposes a domain model for state machines. It includes concepts typically associated to state machines (e.g., states, transitions, triggers) and some additional constructs that, although not directly related in the domain, result quite important for completely expressing a state machine. For example, there is the notion of *Constraint* needed to express guards in the transition, and the some action scripting instructions to express actions in

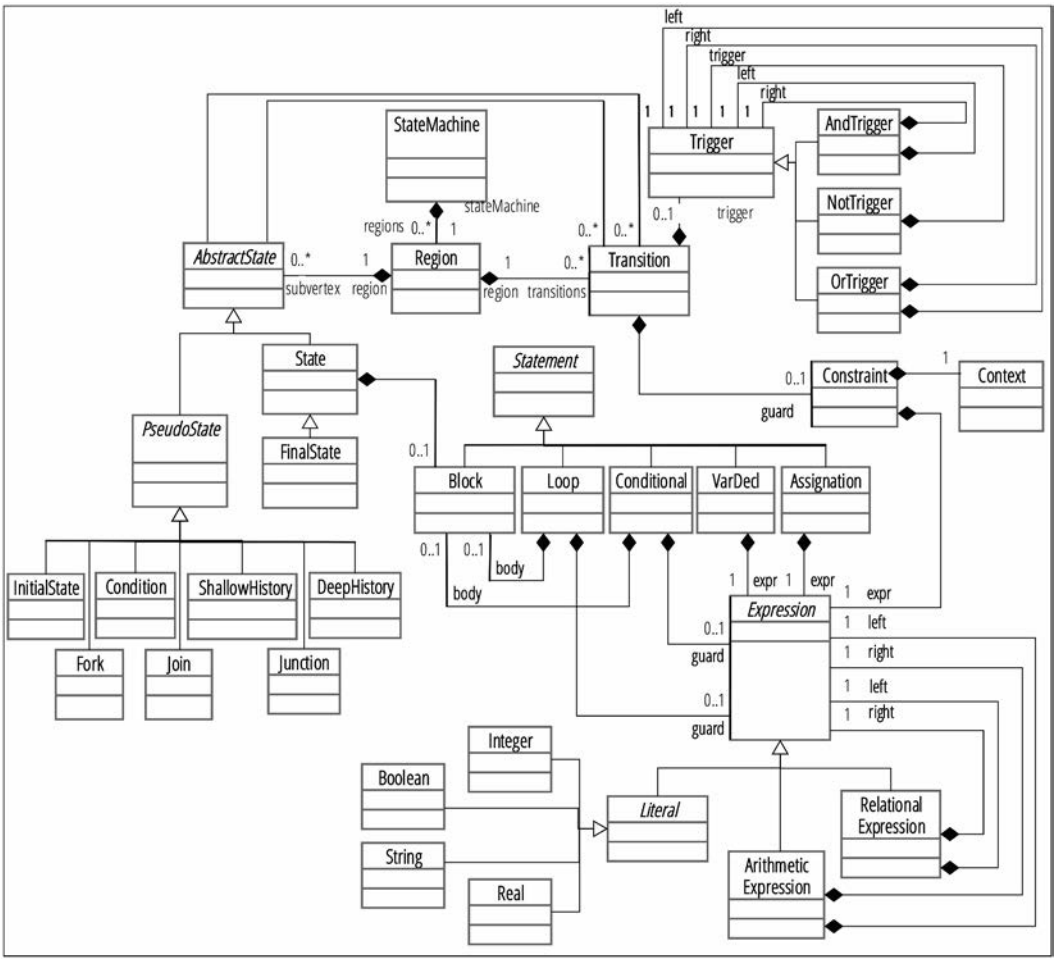


Figure 5.9: A domain model for finite state machines

the states. The purpose of this domain model is to illustrate our methodology. We will reference it in the rest of the chapter.

Identify Functional Variability

The second activity in abstract syntax engineering is to identify the functional variability supported by the language product line. The input of this activity is the domain model whereas the output is a second version of the domain model where each of the concepts are marked as mandatory or optional. The challenge of this phase, is to identify what are those concepts that must appear in all the DSLs as well to identify those concepts that are optional. Note that the mandatory concepts correspond to the commonality of the language product line whereas the optional concepts correspond to its variability.

Using the notion of "completeness" to identify functional variability. An effective way to define whether a concept is mandatory or not is by clearly establishing the *completeness* of a model i.e., to identify the minimum set of concepts needed to build model that can be used in the context in which the language product line will serve. In the case of the state machines, for example, the completeness criterion could be that the state machines are executable. Hence, a DSL for state machines is complete if and only if it provides the minimum elements to create state machines that can be executed. Intuitively, we can state that those elements are: (1) entry points (i.e., initial pseudostates) that indicate the initial point of the execution of the state machine; (2) exit points (i.e., final states) that indicate the end of the execution of a state machine; (3) triggers that allow the state machine to react to events in its environment; and (4) actions in the states to express the executability of each state at runtime. Note that if the state machines are meant to be used only as modeling artifacts describing some behavior with no real execution expected, then the completeness criterion for the DSLs would change and the analysis of functional variability would be different.

Figure 5.10 shows the analysis of functional variability for the example of the state machines. In that figure, optional concepts are marked with the **(?)** symbol. From the analysis above we can state that, to be executable, a state machine needs core concepts such as *StateMachine*, *Region*, *AbstractState*, *State*, *Transition*, *Trigger*, and *PseudoStates*. In addition, the concepts *InitialState* and *FinalState* are mandatory since they represent entry and exit points respectively. In addition, all the concepts associated to action scripting (i.e., *Statement*, *Block*, *Loop*, *Literal*, etc) are needed to permit the expression of actions on the states. Contrariwise, the different types of triggers (i.e., *AndTrigger*, *NotTrigger*, and *OrTrigger*) have not necessarily to be available for expressing an executable state machine. They can be viewed as facilities that enhance a given functionality. A similar situation occurs for the different types of pseudostates which enhance the expressiveness of a state machine, but are not fundamental for the executability of a state machine. Note also that the completeness criterion does not demand the existence of guards in the transitions. A state machine can be executable even if the transitions are always lunched automatically without checking the state of the environment. As a result, we will define constraints as optional even if we suspect that it is a functionality that will be always desired by final users.

Identify Concept Clusters

The next activity in the abstract syntax engineering process is to design a first version of the modular design for the language product line. Concretely, during this activity language designers should identify *clusters* of language concepts each of which will constitute the abstract syntax of a language module in the modular design. Those concept clusters will be enhanced with semantics during the semantics engineering process thus becoming fully specified language modules. The identification of concept clusters is not a trivial task, however. This process should take into account at least three considerations:

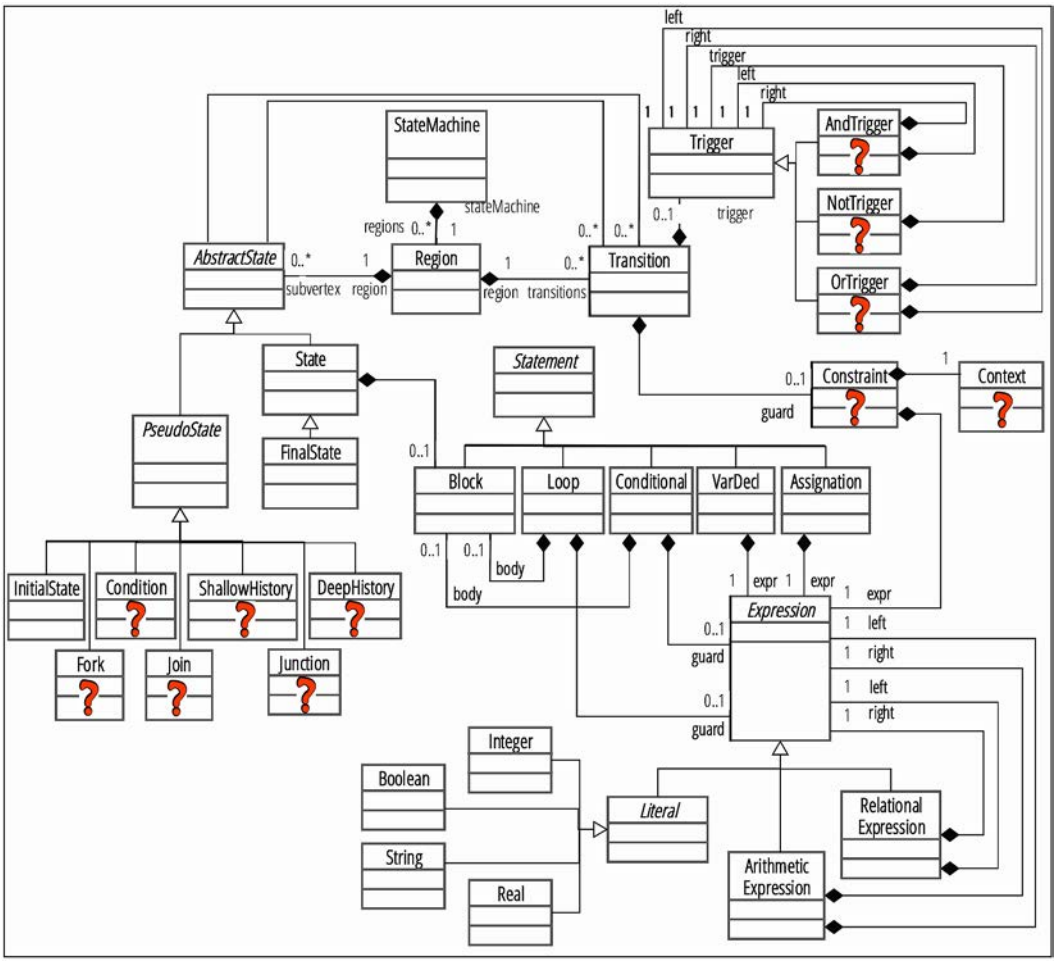


Figure 5.10: A domain model with optional/mandatory concepts for finite state machines

- **Alignment with the domain knowledge.** Because concept clusters will constitute the abstract syntax of language modules, and those modules implement language features, the identification of concept clusters will be visible in the feature models. In other words, the identification of concept clusters is not only a first step in the definition of the modular design, but also a first step to the identification of language features of the feature model. In turn, in software product lines engineering feature models are intended to capture domain knowledge [37]. Hence, the identification of concept clusters should consider the domain knowledge. Language designers should be sure that concepts grouped in the same cluster have certain relationship among them so they can be conceived as part of the same atomic feature with a well-defined objective.
- **Alignment with components-based design principles.** Besides being aligned with

the domain knowledge, the modular design of a language product line must own certain quality degree that guarantees its effectiveness to address the needs of final users under real conditions. Concretely, the modular design should be easy to maintain, reuse, and evolve so it can be easily adapted to dynamic domains. As a result, concept clusters should be defined while respecting well-known design principles (such as high-cohesion and low-coupling) that maximize the modular design' quality [124, 125].

- **Alignment with the functional variability.** Finally, concept clusters should reflect the functional variability previously identified. For example, concepts marked as mandatory should be separated from concepts marked as optional to guarantees that, at configuration time, there are no modules that must be selected because they include a mandatory concept and that includes an optional concept that, in that case, must be always selected so it is not actually optional.

Using hierarchical domain analysis to identify concept clusters. So far, we have discussed some considerations that should be taken into account during concept clusters identification. Let us now discuss some insights that language designers can use to conduct this activity. In particular, there is an observation that results quite useful to this end: the domains associated to a DSL are not necessarily isolated and completely independent [126, 22]. Contrariwise, the domain of a DSL is typically composed of several interdependent and self-contained sub-domains.

Consider for example the domain model for the case of the state machines previously introduced in Figure 5.9. Certainly, all the concepts appearing in the model are relevant in the domain of finite state machines. However, one can identify certain sub-domains that can be analyzed independently. For instance, concepts such as *Expression*, *ArithmeticExpression*, *Literal*, *Integer*, *Boolean*, or *String* are useful for expressing finite state machines but they can be grouped in a the sub-domain of the arithmetic expressions which is a complete domain itself. A similar situation occurs for the domain concepts *Constraint* and *Context*. Those concepts belong to the domain of the constraints but are also relevant for the domain of state machines since they permit to express guards in the transitions.

This observation about sub-domains has been previously exploited in software engineering. Indeed, we can find some research works (e.g., [127, 128]) that analyze the hierarchy of the domain associated to a software system to solve problems such as the recovery of software architectures from legacy systems [127] or the construction of software component libraries [128]. The main idea is to define a *relatedness metric* to quantify the degree at which two domain elements are related each other. Then, this metric is used to group the domain elements in hierarchical clusters that are viewed as sub-domains which provide useful information about the system under study.

We propose the use *hierarchical domain analysis* for the identification of concept clusters. If we are able to build a domains hierarchy from the concepts provided in the domain model, then we can identify some atomic sub-domains that can be defined as clusters whose

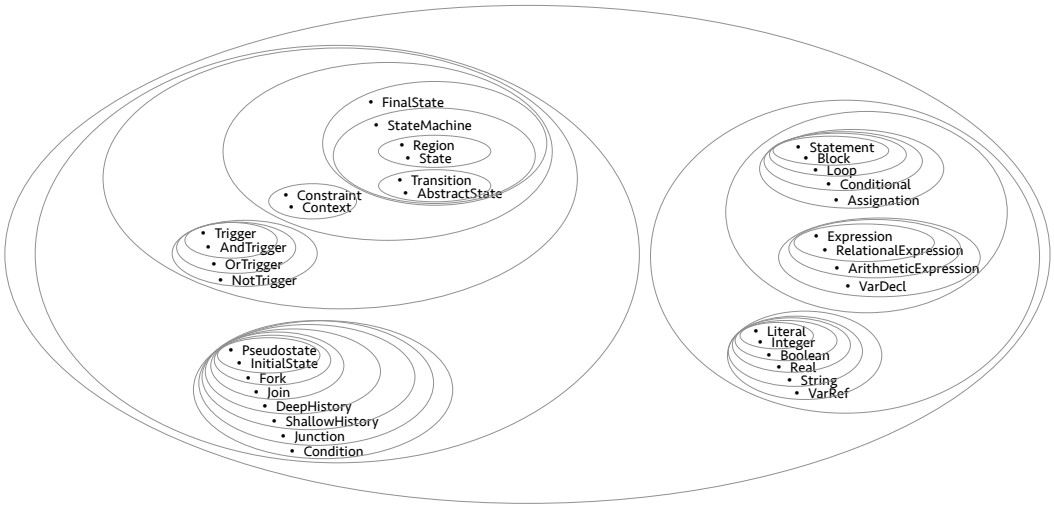


Figure 5.11: Venn diagram for the hierarchical domain analysis for state machines

concepts will be highly related. We propose to use *cohesion* as relatedness metric between language concepts. This makes sense since the notion of cohesion is associated to the "degree to which the elements of a module belong together" [124]. When the software under study is a domain-specific language, cohesion refers to the relatedness of the language constructs composing a language module.

The result of the hierarchical domain analysis is a Venn diagram such as the shown in Fig. 5.11 for the example of the state machines. Note that it results quite useful to understand the degree of relatedness among the language constructs. Indeed, using this result we can propose a concrete set of language modules. Concretely, we can identify two main sub-domains: the one at the right of Figure 5.11 including all the concepts related to the expression of state machines themselves, and the one at the left of the same figure including all the concepts related to the expression of action scripts. Furthermore, the sub-domain of the state machines can be, in turn, decomposed in several sub-domains. For example, we can see that the pseudostates from a sub-domain as well as the triggers. The constraints are also part of a sub-domain whereas the core concepts of state machines i.e., *StateMachine*, *Region*, *State*, *Transition*, *AbstractState*, and *FinalState*, form another sub-domain. A similar situation occurs for the sub-domain of action scripting. We can identify three sub-domains: literals, expressions, and control structures.

The details of how to apply hierarchical domain analysis in languages modularization are deeply explained in Appendix B.

Build Variability Model

The third step in the abstract syntax engineering is to build a variability model capturing the functional variability of the language product line. As explained before, we use feature models to this end. Hence, the output of this step is a feature model supporting the functional variability identified in section 5.2.1, and which is aligned with the concept clusters identified in section 5.2.1. It is worth nothing that, besides supporting functional variability and being aligned with concept clusters, feature models should own quality attributes such as analyzability, changeability, and understandability [129].

To achieve those quality attributes, there must exist a design process where the knowledge and experience of software engineers (language designers in this case) is of vital importance. Although it is difficult to define a precise methodology or algorithm to build feature models achieving all those quality attributes, we can use hierarchical domain analysis to provide a first version of the feature model that can be later improved/refined by software engineers according to the specific needs. The hypothesis is that if we use the a hierarchical analysis of the domain knowledge, then the hierarchy of the features in the feature models will correspond to a real domain hierarchy, and feature models will be easy to read for the engineers who will configure the DSLs. To this end, we propose to follow the steps below:

(1) **Create a root.** We need to start the construction of the variability model by creating a root. To this end, we propose to create an artificial root implemented as an abstract feature that will have the name of the language product line. In our case, we will call it *State Machines*.

(2) **Translate the hierarchical domain analysis into a tree-like structure**. The challenge now is to use the information provided by the hierarchical domain analysis to build the a feature model capturing the functional variability of the language product line. To this end, we propose to produce a simple variability model where each concept cluster is represented in one and only one language feature. The idea is to analyze those clusters according to their position on the Venn diagram in figure 5.11, and use that information to find a good position in the hierarchy for the corresponding feature.

Figure 5.12 shows an example of a feature model created from the hierarchical domain analysis. Note that, in the general case, the fact that a sub-domain is contained by another sub-domain is reflected by a parent-children relationship in the hierarchy of the feature model. However, we decide to pull up the feature *Constraints* to the second level of the tree since it seems to be a DSL at the same level of state machines and action scripting.

Note also that we include some abstract features abstract features i.e., *Pseudostates* and *Triggers* to improve the readability of the model.

(3) **Obtain the dependencies graph and add the *implies* constraints.** To finish the

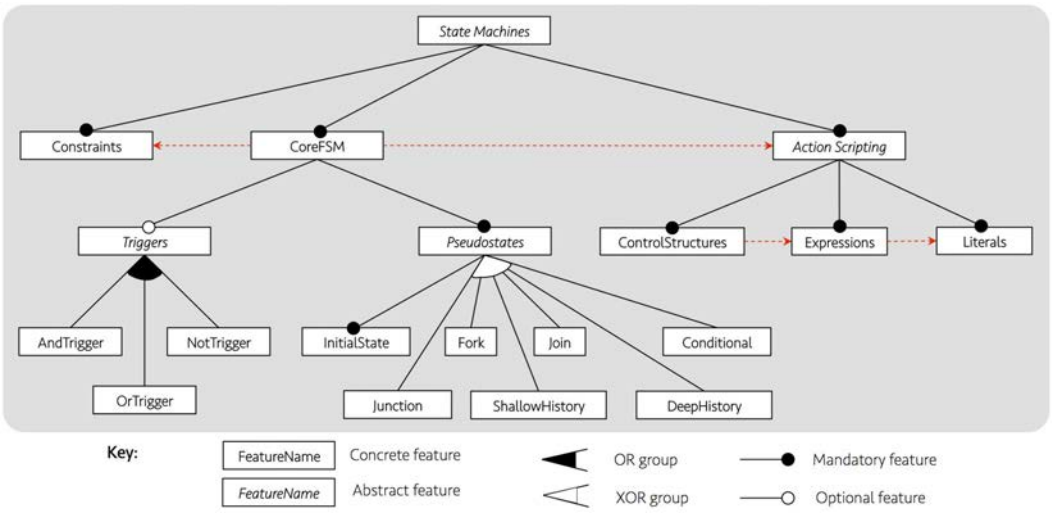


Figure 5.12: Features model for a language product line of finite state machines

construction of the variability model, we need to be sure that it encodes all the dependencies among the concept clusters. To do so, we need to obtain the dependencies graph from the concept clusters of the language modular design.

In this graph, each vertex represents a concept cluster and each arc represents a dependency between two concept clusters. Then, we need to check (one by one) if each dependency is encoded by the hierarchy of the feature model –note that a child feature can be selected if and only if its parent feature is selected so hierarchy is a mechanism to encode dependencies–. If there are non-encoded dependencies, they should be included in the model in the form of *implies* dependencies.

Evaluate

Once we have a first version of the language product line, we need to evaluate it in order to guarantee their quality and envision possible improvements. As the reader might imagine, such evaluation is as complex as the evaluation of software product lines. It should take into account not only functional aspects of the reusable assets but also quality attributes such as usability, maintainability, and so on.

In the case of domain specific languages, both functional testing and quality attributes evaluation are still emergent subjects of research. Although there are some advances in the literature for unit testing [100, 130], and debugging [131], the definition of a complete evaluation framework covering all the aspects of testing of DSLs implemented in the technological space used in this work is still an open issue. Naturally, this fact limits the evaluation of language product lines.

For the scope of this work, however, we propose a simple evaluation of language product

lines based on the *structural soundness* of its modular design. The structural soundness of a modular design is evaluated in terms of the principles of high-cohesion and low-coupling. These principles have demonstrated benefits in terms of understandability, maintainability, and reuse in object oriented software systems [124, 125]. If we admit that there is some synergy between object oriented programming and language engineering (specially when using metamodeling techniques) we may assume that those benefits are also appearing within the development of domain-specific languages. We envision an empirical study to find out evidence that support such statement as part of the future work of this thesis.

To evaluate cohesion and coupling in a modular design, it is important to define metrics that indicate the level at which each principle is achieved. In such a way, language designers can objectively evaluate their clusters with respect to cohesion and coupling, and then improve their designs. In this work, we propose to measure cohesion and coupling as follows:

- **Measuring *cluster cohesion*:** A first approach to measure cohesion was introduced in section 5.2.1 for hierarchical domains analysis. That metric, however, is a pairwise measure that computes the cohesion of two language concepts with respect to the domain model they belong to. For the case of modular design' evaluation, we need to measure the cohesion at the level of the concept clusters. That is, we need to find a measure of cohesion of each cluster in terms of the relationships existing among their concepts. To this end, we define a new metric termed *cluster cohesion*. Using the notion of *inter-connectivity* provided by Mancoridis et al. [132], we propose to compute cohesion the cluster Cl_i as follows:

  Definition The cohesion of a cluster Cl_i is computed as the quotient between the amount of relationships (references + inheritances) among the concepts of the cluster and the square of the number of concepts of the cluster. □

- **Measuring inter cluster coupling:** The notion of coupling is associated to how strong is the dependency between two software modules [125]. When the software under study is a domain-specific language, we propose to measure coupling in terms of the amount of cross-cutting relationships existing between two language clusters. To this end, we adopt the *inter-connectivity* measure introduced by Mancoridis et al. [132]. Concretely, for each a pair of language clusters (Cl_i, Cl_j) we compute coupling as follows:

  Definition The coupling between two clusters Cl_i and Cl_j is computed as the quotient between the number of cross-cutting relationships and two times the factor between the amount of concepts in the involved clusters. □

Figure 5.13 illustrates structural soundness for modular design in terms of coupling and cohesion. An optimal design is one in which the average coupling of its clusters is minimal whereas the average cohesion of its clusters is maximal. During the development process, language designers could refine the design thus improving the metrics and hence optimizing

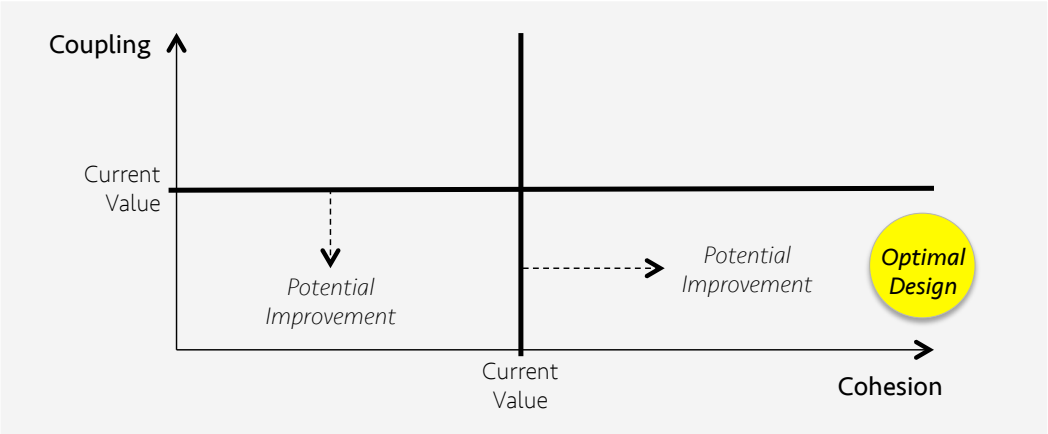


Figure 5.13: Clusters for a DSL for finite state machines

the structural soundness of the modular design. To this end, they can re-define the concept clusters by increasing/decreasing the granularity level in which the hierarchical domain analysis is used. For example, whereas in the current design we decided to define the clusters *Literals*, *Expressions*, and *ControlStructures* separately, we could also define one single cluster called *ActionScripting* (note that Figure 5.11 shows that these three clusters conform one sub-domain themselves). This decision will reduce the number of modules, and certainly modify the values for cohesion and coupling. This decision, however, will have an impact on the definition of the variability models so they should be co-evolved with the new version of the concept clusters.

5.2.2 Semantics Engineering

The second activity of the domain engineering process is dedicated to the implementation of the semantics of the language product line. The input of this activity is the domain model annotated with variability, as well as the first version of the language modular design and variability models produced in the abstract syntax engineering. The output of this activity is the language product line enhanced with semantics. The steps we propose to address this activity are roughly presented in Fig. 5.8. In the reminder of this section we explain them in detail.

Formulate Semantics

The first step in the semantics engineering phase is to formulate the semantics of each language module identified in the abstract syntax engineering phase. This task should be performed by language designers in junction with the domain experts, which are the stakeholders owning a deep understanding of the domain. The work of language designers

is to fully understand those semantics and formulate them in some concrete artifacts –as most formal as possible– that reduce misunderstandings.

Because of the technological space we are addressing of this thesis, this step of semantics formulation phase can be inspired from object oriented technologies. Indeed, there are many synergies between executable meta-modeling and object-oriented programming; each language construct is represented by a meta-class and the operational semantics of the DSL is implemented in the form of java-like methods in those meta-classes (a.k.a., domain-specific actions). We propose to take advantage of these synergies and use strategies such as design by contracts in the formulation of the semantics for the language modules.

Since design by contracts has been largely discussed in the literature, we are not going to go further in the explanation of such approach. Rather, in the following we will use it to specify the semantics of one language module of our example of state machines as an illustrative exercise.

Design by contracts for the *CoreFSM* module. Let us use design by contracts to formulate the semantics of the *CoreFSM* module. This module contains the main functionality of a state machine: evaluating a set of input events by activating and deactivating states that are linked each other by transitions. The idea of this exercise is to take this simple functionality and described it in terms of domain-specific actions and contracts.

Identify Semantic Variability

As part of the process of semantics formulation, language designers and domain experts might conclude that there are several interpretations for the semantics of certain constructs. Indeed, in many cases semantics variability is the main motivation to implement a language product line. There are several examples of this; consider the UML specification which is full of what they call ”semantic variation points” that refer to different possible ways to understand the same concept.

The challenge in this phase is to identify those possible variation points and select those that are potentially interesting for the final users. Once those variability points are identified, language designers need to specify the different versions of the methods and their contracts.

Language Modules Implementation

Once the contracts of the domain-specific actions are well-defined, the next step is to implement them by writing the corresponding code of the body of the methods. Although this task might be considered as straigforward after the definition of the methods, the truth is that is in the coding phase where language designers actually understand the deep details of the semantics. As a result, during this phase might appear more semantic variation points.

Update the Variability Model

Once the semantics is implemented and all the variation points are more or less stable, the next step is to update the variability model by enhancing it with the semantic variability. To this end, language designers should build the orthogonal variability model and link it with the corresponding language features corresponding to the modules they depend.

5.3 Summary

In this section we introduced a strategy to specify a language product line. Our strategy takes into account languages modularization and languages variability management. For the case of languages modularization, we propose a definition for required and provided interfaces that non only supports the modularization scenarios identified in the literature of software language engineering but also addresses some classical principles identified in the literature of software modularization. In the case of languages variability management, our strategy is based on the combination of feature models with orthogonal variability models. It results most convenient in terms of structure and meaning and favors multi-staged configuration. All these ideas are applicable for metamodels-based DSL whose dynamic semantics is specified through domain-specific actions.

Chapter 6

Reverse Engineering for Bottom-Up Language Product Lines

So far, we have introduced facilities supporting the construction of top-down language product lines. Besides, we proposed a methodology aimed to guide the domain engineering process. These facilities are useful when the scope of the development project is clear and language designers can justify the effort behind the construction of a language product line. Nonetheless, the construction of a language product line is not always a purpose itself at the beginning of a language development project. Rather, language designers are often asked to build an initial DSL with in a well-defined domain and for a specific purpose, and with the evolution of the project, some variants of the DSLs are needed in order to address new requirements. In such a case, language designers have to create a new development branch and perform the corresponding modifications. This process is repeated or each new variant of the DSL. At some point, it becomes impractical and the challenge is to perform a re-engineering process where the different variants of the DSLs are used to build a language product line from a bottom-up perspective.

The clone-and-own approach. We aim to contribute with a strategy to reverse engineering language product lines from sets of existing DSL variants which have been built through the clone-and-own approach. Under this approach, the DSL variants are created by cloning existing versions of a DSL and performing the corresponding adaptations. While several research works have shown that such a practice is quite common in software development projects [133, 134], in a recent work [135] we provided some empirical evidence showing that it is also a common practice in language development. We performed an analysis of a large pool of DSLs obtained from GitHub, and we detected a relevant amount of specification clones among them. The details of this study are presented in Appendix C.

A running example. Suppose a team of language designers working on the construction of the DSL for finite state machines. To this end, language designers follow the UML specification [121] thus defining language constructs such as states, regions, transitions,

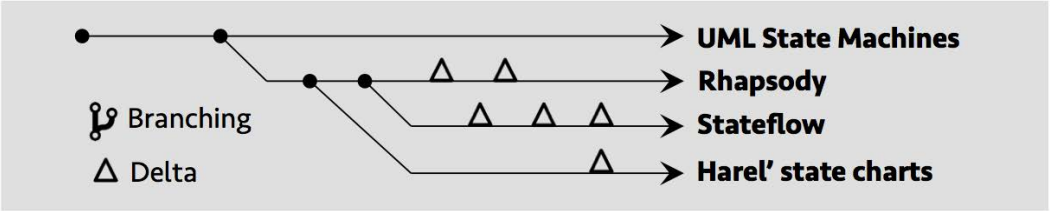


Figure 6.1: Example of clone-and-own pattern

triggers, and so on. Those language constructs are specified in terms of their syntax and semantics. So, at the end of the language development process, they release an executable DSL which behavior complies the UML specification.

Once this first DSL is released, the language designers are asked to build a new variant, which must comply the Rhapsody specification [122] (i.e., another formalism to finite state machines). This new variant shares many commonalities with UML state machines, but introduces differences at both syntax and semantics levels [59]. At this point, language designers face the problem of reusing as much as possible the constructs defined in the first DSL during the construction of this new variant.

One of the possibilities that language designers have is to use the clone-and-own approach by copy-pasting the specification of the first DSL in a new project (or cloning the repository), and then performing the needed adaptations as shown in Fig. 6.1. This approach permits a fast prototyping of the new DSL variant. After this process, language designers obtain two different DSLs implementing different formalisms of state machines. Those DSLs have some commonalities among them –those commonalities are materialized in terms of specification clones–. And at the same time, the DSL have some particularities that make them unique.

Suppose now that the final users need support for other formalisms for state machines. Hence, the team of language designers is asked to implement two more DSL variants: the first one complying the Stateflow specification [136], and the second one complying classical Harel state machines [123]. If the language designers use, again, the clone-and-own approach, then they will obtain set of four DSL variants. Those variants share specification clones, and own certain particularities.

The problem that language designers face at this point is two fold. First, they will have to produce a new variant for each FSM formalism. This becomes specially challenging when final users need to combine some specifications to define hybrid formalisms. Language designers will have to produce a new version of each DSL for each desired combination, and the clone-and-own approach becomes impractical. Second, the existence of specification clones increases the maintenance costs of the involved DSLs. If language designers detect bugs in one of the specifications; they will have to check all the DSL variants. While several approaches have been proposed to exploit the notion of code clones to produce software product lines from existing product variants [137, 138, 139], in this chapter we exploit

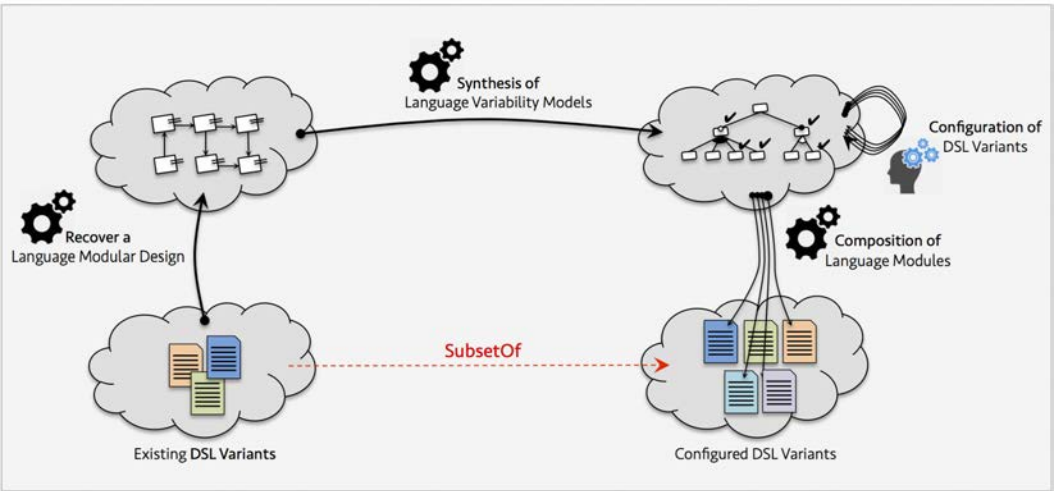


Figure 6.2: A reverse-engineering strategy to bottom-up language product lines

the notion of language specification clones to reverse engineering language product lines from existing DSL variants.

6.1 Approach Overview

In this section, we present our reverse engineering technique to support the construction of bottom-up language product lines. As shown in Fig. 6.2, the proposed technique is composed of four steps. During the first step, we automatically recover a language modular design for the language product line. Such a modular design is composed of a set of language modules and a set of dependencies among them. During the second step, language modules' dependencies are used to synthesize a variability model that can be used, during the third step, to configure concrete DSL variants. Finally, during the forth step the DSL variant is assembled by composing the involved language modules.

In the following, we explain the way in which we perform the first and second steps (i.e., recovering language modular design and synthesis of variability models). Since the resulting language product line is specified in terms of the meta-languages introduced in Chapter 5, the third and fourth steps (i.e., configuration and composition) can be performed as explained in sections 5.1.2 and 5.1.1, and will not be discussed in this section.

6.2 Recovering a Language Modular Design

Let us start the description of our reverse engineering technique by explaining the way in which we identify the set of language modules and dependencies that constitute the language modular design of the product line. Roughly speaking, we propose to take all

the given DSL variants and unify them in a unique DSL. Then, the unified DSL is broken down into several interdependent language modules. This process results in a factorization of the specification clones existing in the DSL variants. The purpose of this strategy is to remove as many specification clones as possible, thus reducing the maintenance costs. Our factorization strategy is based on four principles explained in the following:

Principle 1: *DSL specifications are comparable. Hence, specification clones can be detected automatically.* To detect the specification clones in a given set of DSL variants, we need to compare those specification clones and find out the repeated specification elements. For the technological space discussed in this thesis, specification elements for the abstract syntax correspond to metaclasses whereas specification elements for the semantics correspond to domain specific actions. Hence, comparison of DSL specifications corresponds to compare metaclasses and domain specific actions.

Comparison of metaclasses. To compare metaclasses, we need to take into account that a metaclass is specified by a name, a set of attributes, and a set of references to other metaclasses. Two metaclasses are considered as equal (and so, they are clones) if all those elements match. Formally, comparison of metaclasses is defined by the operator $\doteqdot$.

$$\doteqdot : MC \times MC \rightarrow bool \tag{6.1}$$

$$\begin{aligned} MC_A \doteqdot MC_B = true \implies & \\ & MC_A.name = MC_B.name \wedge \\ & \forall a_1 \in MC_A.attr \mid (\exists a_2 \in MC_B.attr \mid a_1 = a_2) \wedge \\ & \forall r_1 \in MC_A.refs \mid (\exists r_2 \in MC_B.refs \mid r_1 = r_2) \wedge \\ & |MC_A.attr| = |MC_B.attr| \wedge \\ & |MC_A.refs| = |MC_B.refs| \end{aligned} \tag{6.2}$$

Comparison of domain specific actions. To compare domain specific actions, we need to consider that –similarly to methods in Java– domain specific actions have a signature that specifies its contract (i.e., return type, visibility, parameters, name, and so on), and a body where the behavior is implemented. Two domain specific actions are equal if their signatures and bodies are equivalent.

Whereas comparison of signatures can be performed by syntactic comparison of the signature elements, comparison of bodies can be arbitrary difficult. If we try to compare the behavior of the domain-specific actions, then we will have to address the semantic equivalence problem, which is known to be undecidable [140]. To address this issue, we conceive bodies comparison in terms of its abstract syntax tree as proposed by Biegel et al. [141]. In other words, to compare two bodies, we first parse them to extract their abstract syntax tree, and then we compare those trees. Note that this decision makes sense because

we are detecting specification clones more than equivalent behavior. Formally, comparison of domain-specific actions (DSAs) is defined by the operator $\coloneqq$.

$$\coloneqq : DSA \times DSA \rightarrow bool \tag{6.3}$$

$$\begin{aligned} DSA_A \coloneqq DSA_B = true \implies \\ DSA_A.name = DSA_B.name \wedge \\ DSA_A.returnType = DSA_B.returnType \wedge \\ DSA_A.visibility = DSA_B.visibility \wedge \\ \forall p_1 \in DSA_A.params \mid (\exists p_2 \in DSA_B.params \mid p_1 = p_2) \wedge \\ |DSA_A.params| = |DSA_B.params| \wedge \\ DSA_A.AST = DSA_B.AST \end{aligned} \tag{6.4}$$

Principle 2: *Specification clones can be viewed as sets' intersections, which is useful to factorization.* A DSL specification can be seen as a set of metaclasses and a set of domain specific actions. In doing so, specification clones correspond to intersections among those sets. Those intersection elements can be specified once and reused in several DSL variants [142, p. 60-61]. Hence, we can factorize specification clones by breaking down the intersections existing among DSL specifications.

Fig. 6.3 illustrates this observation through the running example on finite state machines. We show two Venn diagrams to represent both syntax and semantic intersections. The Venn diagram corresponding to the abstract syntax shows that the classical constructs for state machines such as StateMachine, State, and Transition are in the intersection of the three given DSL variants i.e., UML state machines, Rhapsody, and Harel's state machines. In turn, there are certain particularities for each DSL. For example, the concept AndTrigger is owned by UML and Harel state machines but not for Rhapsody. Concepts such as OrTrigger and NotTrigger are only provided by Harel state machines since the concept of Choice is exclusive of UML state machines.

For the case of semantic variability, the 3-sets intersection is empty meaning that there is not a common semantic for the three DSL variants. Rather, UML state machines and Rhapsody share the domain specific actions corresponding to the constructs of State Machine, State, and Transition. The implementation of Harel state machines is different.

Note that this way to conceive DSL specifications is useful to factorize specification clones as illustrated in Fig. 6.4. Each different intersection is separated in a separate subset that, as we will explain later, is encapsulated in a language module.

Principle 3: *Abstract syntax first, semantics afterwards.* The abstract syntax is the backbone of the DSL specification; it specifies its structure in terms of metaclasses and relationships among them whereas the domain-specific actions add executability to the metaclasses. Hence, the process of breaking down intersections should be performed for the abstract syn-

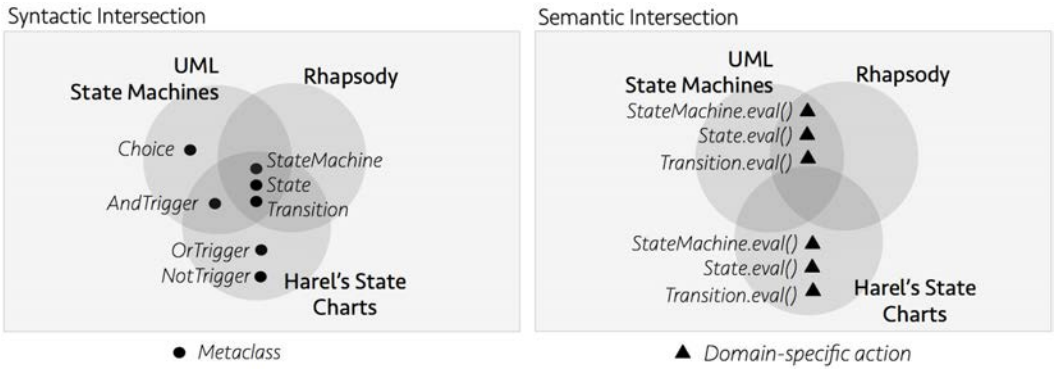


Figure 6.3: Syntactic and semantic intersections in a set of DSL variants

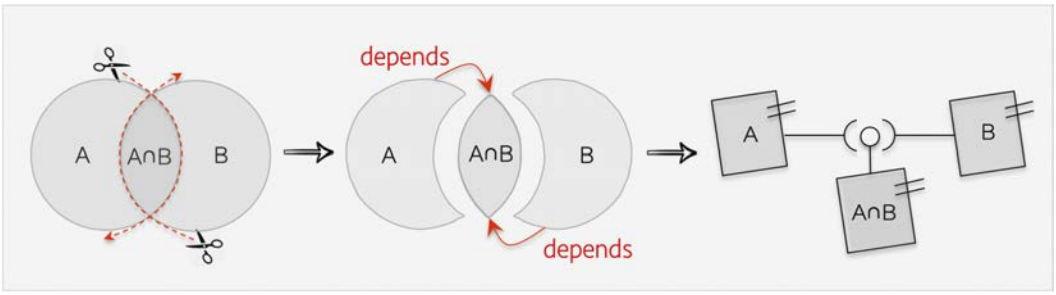


Figure 6.4: Breaking down intersections to factorize specification clones

tax first, thus identifying the way in which metaclasses should be grouped into the different language modules. Afterwards, we can do the proper for the semantics. In doing so, we need to take into consideration the phenomenon of semantic variability. That is, two cloned metaclasses might have different domain-specific actions. That occurs when two DSLs share some syntax specification but differ in their semantics.

Principle 4: *Breaking down a metamodel is a graph partitioning problem.* A metamodel can be seen as a directed graph $G =< V, A >$ where:

- **V**: is the set of vertices each of which represents a metaclass.
- **A**: is the set of arcs each of which represents a relationship between two metaclasses i.e., references, containments, and inheritances.

This observation is useful for breaking down metamodels, which can be viewed as a graph partitioning problem where the result is a finite set of subgraphs. Each subgraph represents the metamodel of a language module.

The principles in action. Fig. 6.5 shows the way in which we recover a language modular design through the principles explained above. It is composed of two steps: unification and breaking down.

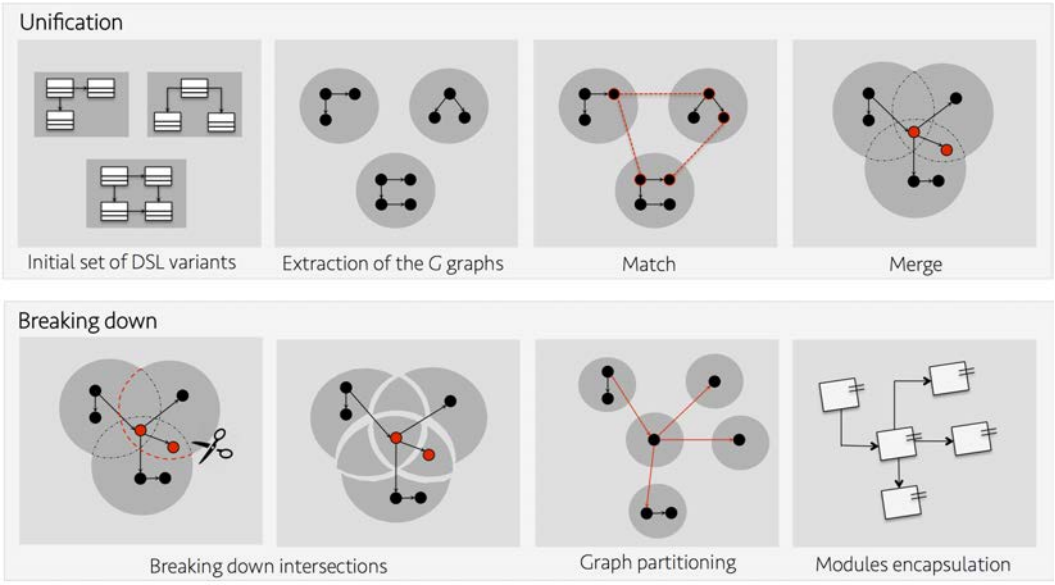


Figure 6.5: Unifying and breaking down for recovering a language modular design

Unification: ***match*** *and* ***merge****.* The objective of this step is to unify all the DSL variants in a unique specification. To this end, we first produce a graph G for the metamodel of each DSL variant according to the principle 4. Second, we use the comparison operators defined in the principle 1 to match the vertices representing the metaclasses repeated in two or more DSL variants. Third, we create the syntactic intersections defined in principle 2 by merging the matched vertices. In doing so, we remove cloned metaclasses. After this process, we have a unified graph (which is not necessarily a connected graph) including all the metaclasses provided in the DSL variants.

To identify semantic intersections, we check whether the domain specific actions of the matched metaclasses are equal. If so, they can be considered as semantic specification clones, and they are also merged. If not all the domain specific actions associated to the matched metaclasses are equal, different clusters of domain specific actions are created, thus establishing semantic variation points.

Breaking down: ***cut*** *and* ***encapsulate****.* Once intersections among the DSL variants have been identified, we factorize the specification clones. To this end, we break down the unified graph using a graph partitioning algorithm. Our algorithm returns a set of clusters of vertices: one cluster for each intersection of the Venn diagram. Arcs defined between vertices in different clusters can be considered as cross-cutting dependencies between clusters. Finally, we encapsulate each vertex cluster in the form of a language module.

Note that the dependencies between language modules can be viewed through the interfaces introduced in chapter 5. Those interfaces are reverse-engineering from each module.

Required interfaces are generating by creating a virtual element for those elements that are required by the module but that are not part of its definition. The provided interface is generated by defining all the specification elements of the language module as public. If there are specification elements that should be hidden, then the language designer should modify the generated definition.

6.3 Synthesizing Language Variability Models

we define an algorithm to synthesize variability models from a given language modular design. This algorithm produces not only a feature model with the abstract syntax variability, but also an orthogonal variability model representing the semantic variability. An overview of the approach is presented in Fig. 6.6.

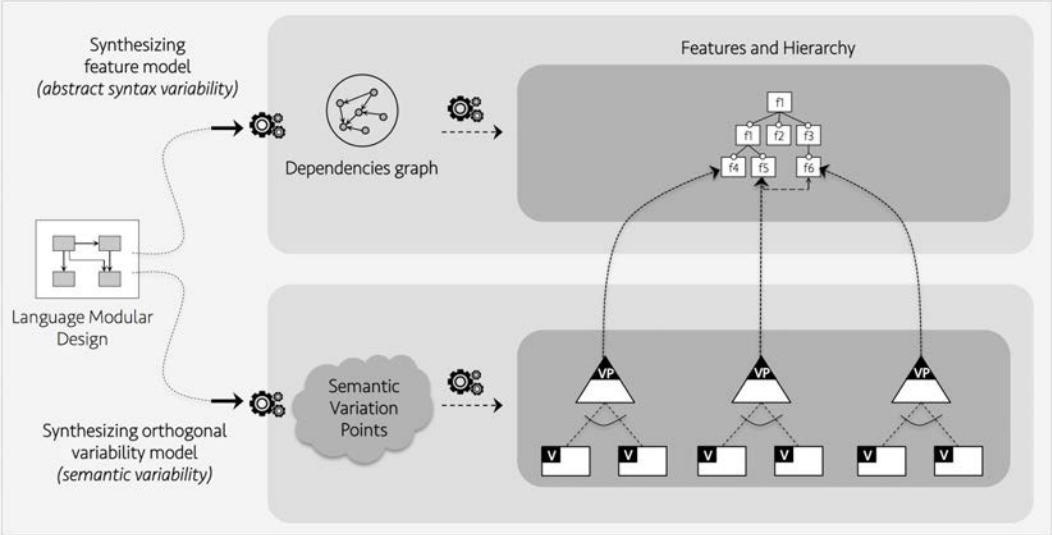


Figure 6.6: Reverse-engineering variability models for language product lines

Synthesizing abstract syntax variability. The first step to represent the variability of a language product line is to extract the feature model that represents the abstract syntax variability. To this end, we need an algorithm that receives the dependencies graph between the language modules, and produces a feature model which includes a set of features representing the given language modules as well as a set of constraints representing the dependencies among those modules. The produced feature model must guarantee that all the valid configurations (i.e., those that respect the constraints) produce correct DSLs.

In the literature, there are several approaches for reverse engineering feature modules from dependencies graphs (consider for example the approach presented by Assunçao et al. [143], or the one presented by She et al., [144]). In our case, we opt for an algorithm

that produces a simple feature model where each language module is represented in a concrete feature, and where the dependencies between language modules are encoded either by parent-child relationships or by the classical *implies* relationship. Our algorithm was inspired from the approach presented by Vacchi et al. [91] which fulfills the aforementioned requirements. Besides, it has been applied for the particular case of languages variability.

The tooling that supports our algorithms is flexible enough to permit the use of other approaches for synthesis of feature model. To this end, we provide an extension point that language designers can use to add new synthesis algorithms. In addition to the one proposed by Vacchi et al. [91], we have integrated our approach with the one provided by Assunçao et al., [143].

Synthesizing semantic variability. Once the feature model encoding abstract syntax variability is produced, we proceed to do the proper with the orthogonal variability model encoding semantic variability. To this end, we need to analyze the results of the process for extracting the language modules. As explained in Section 6.2, according to the result of the comparison of the semantics, a language module might have more than one cluster of domain specific actions. This occurs when the two DSLs share constructs that are equal in terms of the abstract syntax, but differ in their semantics. Since this is the definition of semantic variation point, we materialize those clusters in semantic variation points of an orthogonal variability model.

To do this, we scan all the language modules. For each one, we verify if it has more than one cluster of domain specific actions. If so, we create a semantic variation point where each variation references one cluster. Finally, the semantic variation point is associated with the feature that represents the language module owning the clusters.

6.4 Summary

In this section we presented a reverse-engineering process that allows to automatically produce a language product line from a set of existing DSLs. Our reverse-engineering process starts by breaking down the DSLs into a set of reusable language modules. Then, the process goes through the synthesis of the models that capture the variability existing among the extracted language modules. Since we use the meta-languages introduced in chapter 5, the generated artifacts can be used to configure and assembly new DSLs.

Part III

Implementation and Validation

Chapter 7

Implementation: The Puzzle Toolkit

In this chapter, we present Puzzle: a toolkit that implements the ideas introduced in the contributions of this thesis. It is implemented on top of the Eclipse Modeling Framework (EMF)[1], and it is based on K3[2] and Mélange[3]. Used together, EMF, K3, and Mélange constitute a language workbench to support the definition of executable DSLs where the abstract syntax is specified through metamodels, and the semantics is specified operationally through domain-specific actions. From a high-level perspective, Puzzle can be viewed as a set of extensions to support the construction of language product lines.

7.1 Using EMF, K3, and Mélange to Specify DSLs

EMF is a framework provided by Eclipse that offers a set of capabilities focused on facilitate modeling intensive tasks. Probably, the most visible feature provided by EMF –and the one in which we are more interested for the context of this thesis– is the Ecore meta-language. Such meta-language is intended to facilitate the definition of the abstract syntax of the DSL by means of metamodels.

Once we have a mechanism to express metamodels, we need one for the expression of operational semantics. To this end we use K3, which introduces the notion of *aspect* [69] to facilitate the injection of domain-specific actions to a given metamodel.

The role of Mélange in this context is to facilitate the integration of metamodels with domain specific actions. To do so, Mélange provides meta-language to enable the description in-the-large of the specification artifacts of a given DSL. Hence, in a Mélange script, a language designer can reference a metamodel, inject it some aspects, and perform the weaving process to obtain a complete and executable DSL. In the remainder of this section, we explain how to use those elements though a set of simple examples.

[1] EMF website: https://eclipse.org/modeling/emf/
[2] K3 website: https://github.com/diverse-project/k3
[3] Mélange website: http://melange-lang.org/

Defining aspects in K3. The very first consideration to keep in mind about K3 is that it is the evolution of Kermeta. Kermeta is a language workbench that provide the notion of aspect for specifying operational semantics, and an action language for implementing those semantics. The difference of K3 with respect to the last versions of Kermeta is that K3 does not provide an action language. Rather, it uses Xtend[4] for the definition of semantics, and provides the functionality needed to encapsulate Xtend methods in aspects that can be later weaved in metamodels.

Consider the metamodel introduced at the top of Figure 7.1. It contains two classes `X`, and `Y`; The class `X` contains elements of type `Y` by means of the containment relation `yes`. In turn, the code snippet at the bottom of Figure 7.1 introduces some operational semantics to this metamodel by using K3. Note that the main feature of K3 is the notion of aspect that permits to weave the operational semantics defined in a Xtend class to a metamodel defined in Ecore.

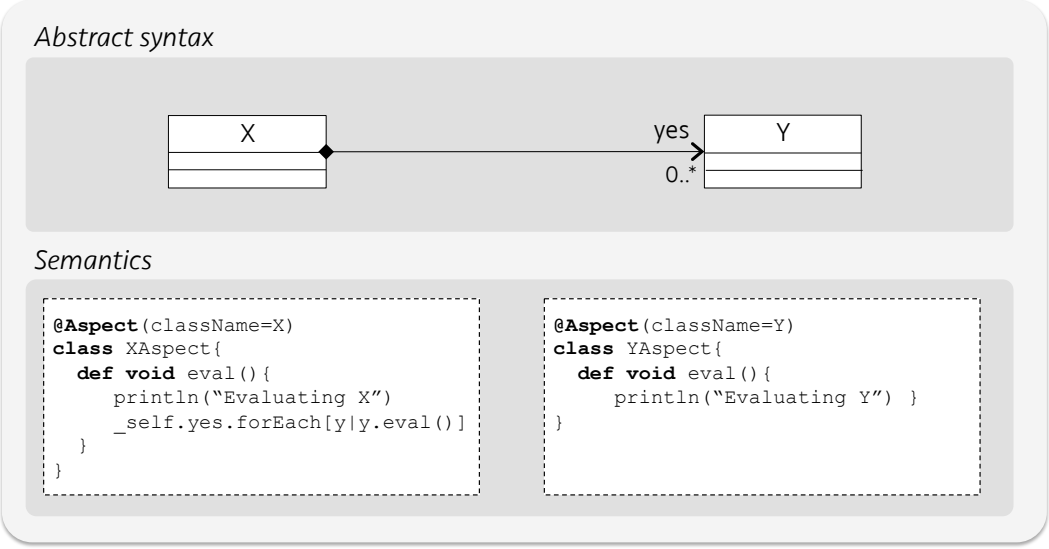


Figure 7.1: Simple example of aspects in K3

In our example, the metaclass `X` is enriched with the operation `eval()` that contains a loop that sequentially invokes the operation defined for the class `Y`. This operation is also defined by using one aspect; in this case extending the metaclass `Y`. Note that the language designer is free to name the aspects as he/she considers correct. In this example, we use the convention `<<metaclass.name>>+Aspect`.

Because K3 is implemented on the top of the EMF, it is possible to use it in junction with tools such as xText [145] or Sirius [146] for generating textual or graphical editors.

[4]Xtend website: https://eclipse.org/xtend/

As a result, we can completely construct DSLs by using this framework: abstract syntax with metamodels in Ecore, semantics with K3, and concrete syntax with editors generators. However, as we mentioned in the definition of the scientific scope of this thesis, our approach for modularization and variability management does not cover this dimension of the specification of a DSL.

Integrating abstract syntax and semantics with Mélange. Let us now to illustrate the integration of a metamodel written in Ecore with the aspects specified in K3. We use Mélange with that purpose as illustrated in Listing. 7.1. To do so, we first define the language itself through the keyword `language`. In the example, we define a language called "Example" for which the abstract syntax is defined in a metamodel called *example.ecore* through the keyword `syntax`, line 4. Then, we enhance that metamodel with the two aspects defined in K3 in Fig. 7.1 through the keyword `with`, lines 6 and 7.

Note also that Mélange provides the expressiveness to execute a model conforming to the defined language. To this end, Mélange provides the keyword `transformation`; lines 10-14 exemplify its use on our simple example. The transformation loads a model located in the file *toy.example*, locates the head element (that should by an instance of `X` since it is the root of the containment tree of the metamodel) and calls the method `eval()` defined in the `XAspect`.

Listing 7.1: Example for the use of Melange to integrate metamodels and aspects

```
1  package SimpleExample
2
3  language Example{
4      syntax "platform:/resource/example/models/example.ecore"
5
6      with fr.inria.diverse.examples.XAspect
7      with fr.inria.diverse.examples.YAspect
8  }
9
10 transformation executeExample{
11     var m = SimpleExample.load("toy.example")
12     var rootX = m.head as X
13     rootX.eval()
14 }
```

7.2 Capabilities of Puzzle

In this section, we describe the capabilities provided by Puzzle to support language product lines engineering, and show how those capabilities can be used in concrete scenarios. As the same as our contributions, those capabilities can be classified into two different types. First, those supporting top-down language product lines engineering; second, those supporting bottom-up language product lines engineering.

7.2.1 Capabilities to Support Top-Down Language Product Lines

Let us start our description of the capabilities of Puzzle by introducing the support that it provides for the construction of top-down language product lines. Puzzle supports the meta-language extensions introduced in Section 5.1, Chapter 5 for the definition of modular DSLs and the expression of variability models for both abstract syntax and semantics.

Modular Specification of DSLs

The first aim of Puzzle is to provide a language modularization approach that supports all the language modularization scenarios while addressing three well known software modularization principles i.e., independent development, information hiding, and substitutability. To do so, we need to enable the definition of language modules with required and provided interfaces. Besides, we need to provide a mechanism to enable the binding between two language modules in order to perform compatibility checking and composition.

Support for required interfaces in Puzzle. As deeply explained in Section 5.1.1, the definition of a required interface corresponds to the definition of virtual elements within a language module. To support such definitions, Puzzle introduces some modularization annotations on Ecore metamodels. For the concrete case of required interface, we introduce the annotation `@Required`. By default, a specification is non-virtual –the flag "virtual" is set in false–; a specification element annotated with `@Required` becomes virtual –the flag "virtual" is set to true– and the element is included to the required interface.

Fig. 7.2 illustrates the use of the `@Required` annotation through the example on state machines (see Fig. 5.2). The example is composed of an Ecore file (at the top of the figure) and a semantics specification (at the bottom of the figure). In the metamodel, the metaclass `Constraint` is virtual so it is annotated with `@Required` as well as the `eval()` operation. It means that the required interface includes not only abstract syntax elements, but also semantic ones. The definition of the `eval()` operation in the required interface permits its use in the definition of the semantics of the state machines language.

Support for provided interfaces in Puzzle. The definition of provided interfaces corresponds to explicitly indicate if the specification elements of a language module are either public or private. To support such a capability, we introduce the `@Private` annotation. By default, all the specification elements are defined as public so they belong the provided interface. If a specification element is annotated with `@Private`, then its visibility is set to private and it is removed from the provided interface.

Fig. 7.3 shows the definition in Puzzle for the language module to express constraints. In that case, we decided to hide all the specification elements of the module except `Program` and `Constraint` because, as explained before, one might consider that they represent the functionality of the language module. All the other specification elements are annotated with `@Private` and they will not appear in the provided interface of the language module.

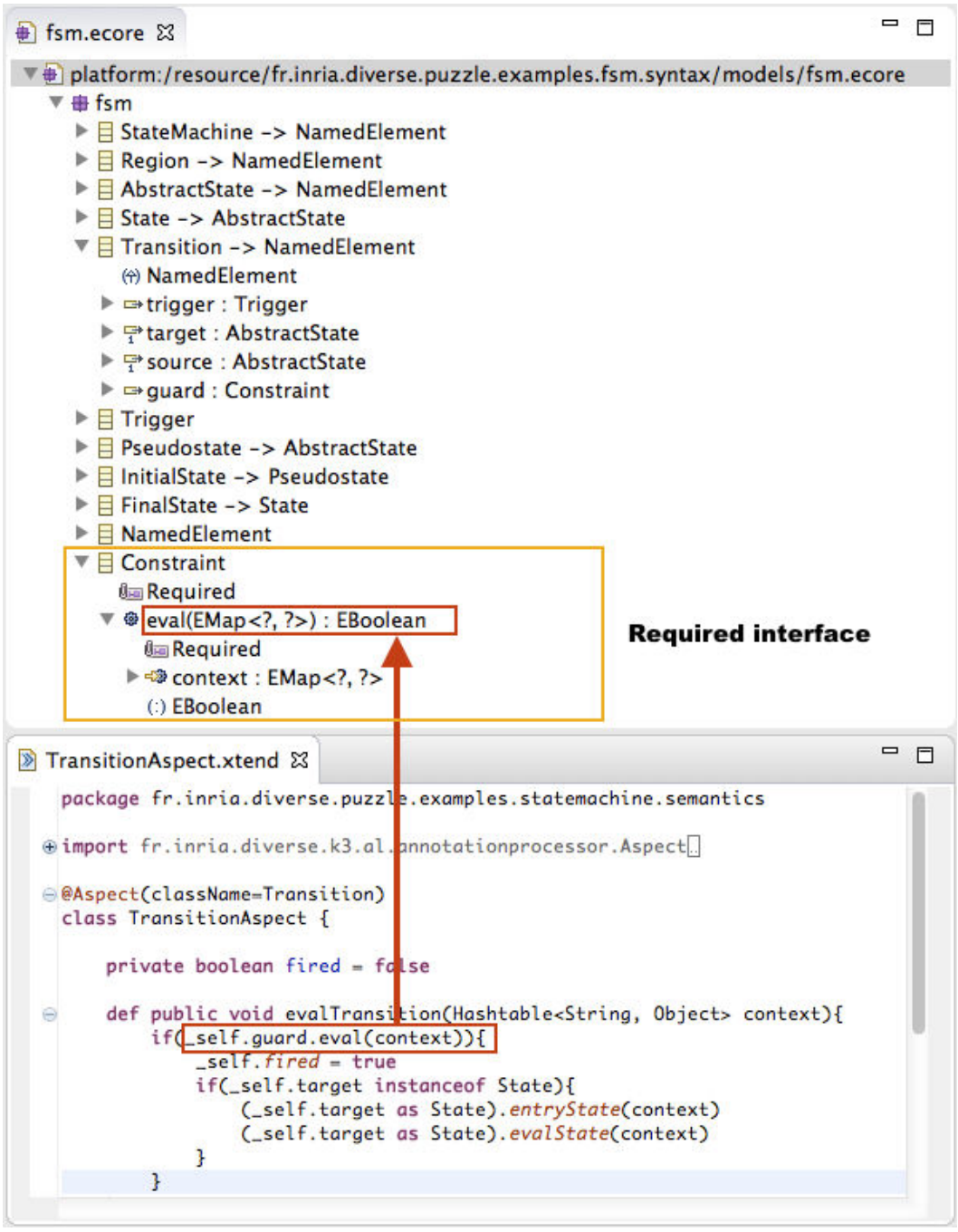


Figure 7.2: Example of required interfaces in Puzzle

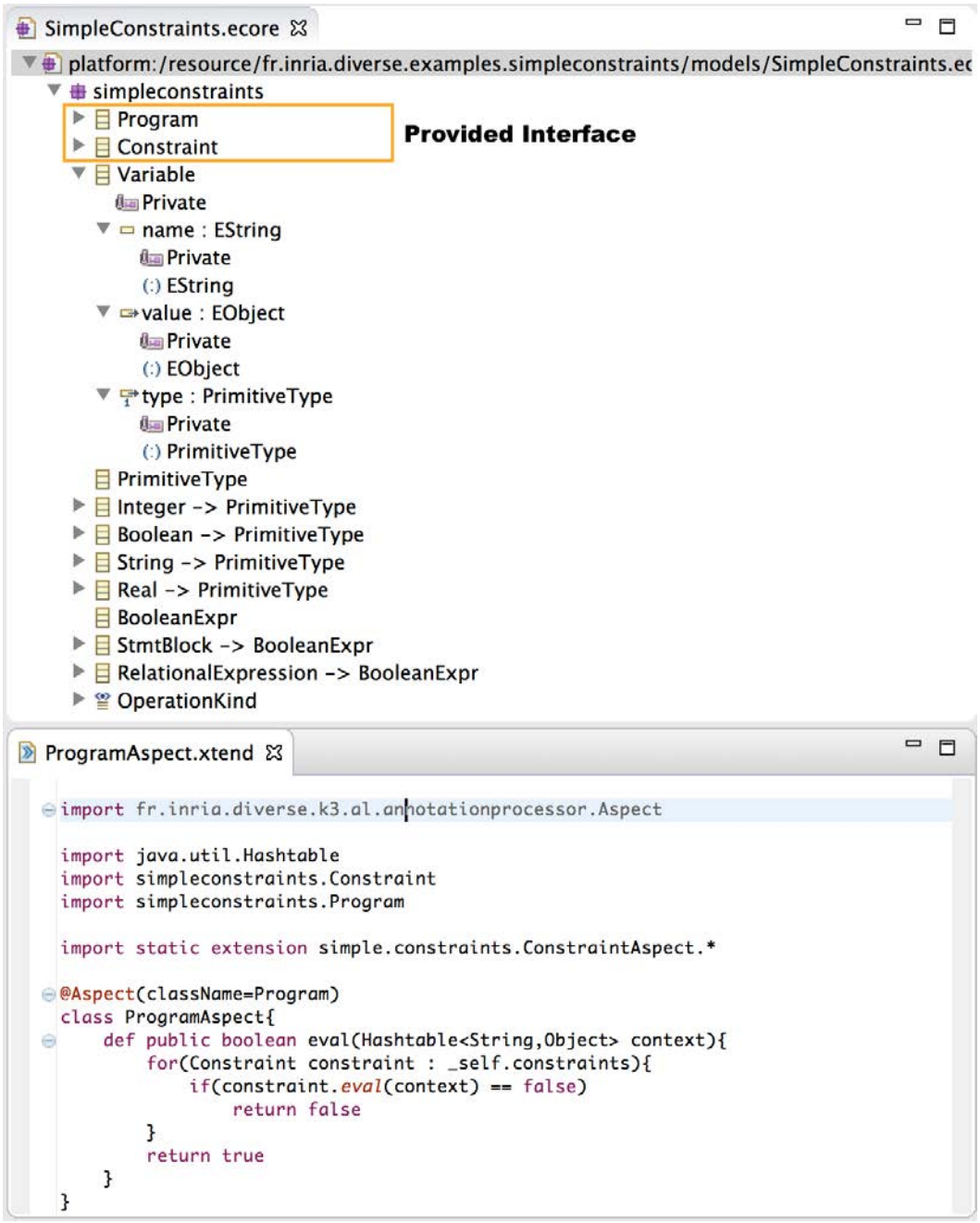


Figure 7.3: Example of provided interfaces in Puzzle

Expressing bindings between language modules. So far, we used required and provided interfaces in the definition of two language modules i.e., one for state machines and another one for constraints. Those modules have a dependency between them: the module for state machines uses some constraints to express guards in the transitions. But... how to define the binding between those modules and perform the corresponding composition?

Note that the binding between two language modules is not defined at the level the modules themselves but at the level of their interfaces. In other words, we do not bind a requiring language module with a providing one, but a required interface to with provided one. We also need to consider that those interfaces should be expressed in terms of model types; currently they are just expressed as annotations in the metamodel. Hence, as a first step we extract the required/provided interfaces from the meta-models by analyzing the annotations; Puzzle provides a service that is able to do this automatically, thus creating the corresponding model types by extracting the elements annotated with `@Required` and `@Private` respectively. Once we have the model types corresponding to each interface of language modules, we can proceed to use Mélange to index all these specification artifacts as shown in Listing 7.2.

Listing 7.2: Example for the use of Melange to integrate metamodels and aspects

```
1  package SimpleFSM
2
3  language CoreFSM requires RequiredFSM{
4      syntax "platform:/resource/fsm/models/fsm.ecore"
5
6      with fr.inria.diverse.fsm.StateMachineAspect
7      with fr.inria.diverse.fsm.StateAspect
8      with fr.inria.diverse.fsm.TransitionAspect
9      ...
10 }
11
12 modeltype RequiredFSM{
13     syntax "platform:/resource/fsm/models/fsm-req.ecore"
14 }
15
16 language Constraints implements ProvidedConstraints{
17     syntax "platform:/resource/constraints/models/const.ecore"
18
19     with fr.inria.diverse.constraints.ConstraintAspect
20     with fr.inria.diverse.constraints.StmtBlockAspect
21     with fr.inria.diverse.constraints.BooleanExpressionAspect
22     ...
23 }
24
25 modeltype ProvidedConstraints{
26     syntax "platform:/resource/constraints/models/const-prov.ecore"
27 }
```

In this Mélange script, each interface is defined by the keyword `modeltype` that references an Ecore file that contains its definition. Besides, the requiring module –in this case, the CoreFSM module defined in lines 3-10– references its required interface through the keyword `requires`. In turn, the providing module –in this case the Constraints module defined in lines 16-23– references its provided interface through the keyword `implements`. The binding between those interfaces is the verification of the subtyping relationship among them. If such a relationship is respected, then the modules are compatible i.e., the services of the provided interface fulfill (at least partially) the needs of the required one, and the composition can be performed.

Expression of Language Variability Models and Configuration

The expression of the variability existing in a language product line is performed via a dedicated metamodel that offers the expressiveness enough to define not only the feature models representing the abstract syntax variability, but also the orthogonal variability model to represent semantic variability and the mapping among these dimensions.

Fig. 7.4 shows how the editor looks like for the example of the state machines. One of the most important characteristics of this editor is that it allows the configuration of a particular DSL. Each feature and variability point can be selected or deselected in the model. Once the configuration is finished, then language designers can use the a menu option to launch a generation process that will produce a Melange script describing the involved module and the corresponding aspects. Then, this script can be used to perform the composition and obtain the DSL.

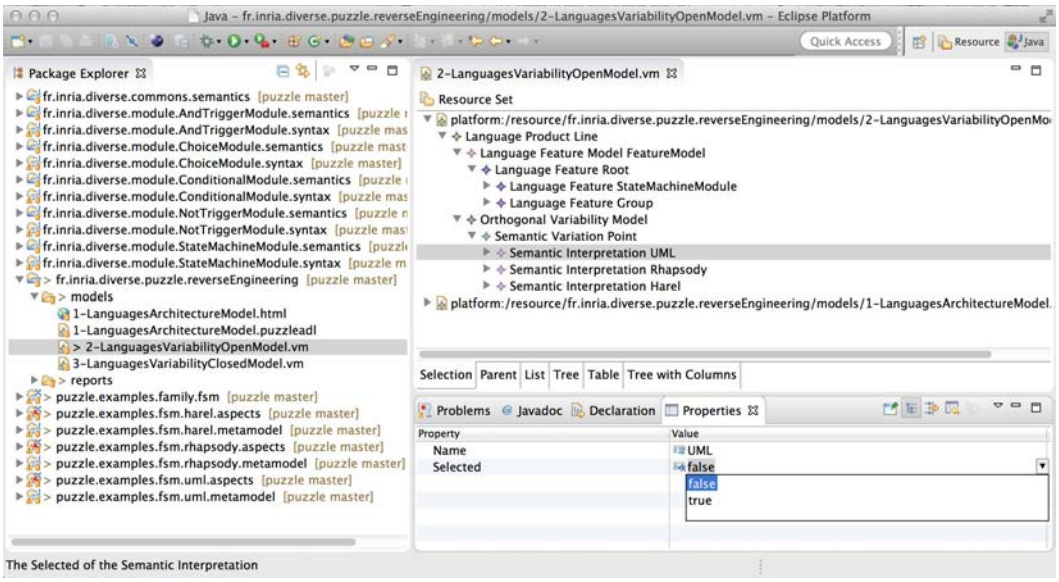


Figure 7.4: Variability management in Puzzle

7.2.2 Capabilities to Support Bottom-Up Language Product Lines

In this section, we present the capabilities provided by Puzzle to support bottom-up language product lines. Concretely, Puzzle offers a set of facilities for the analysis of a set of existing DSL variants and the subsequent reverse-engineering process that permit to extract a language product line from the commonalities and particularities existing among them.

The very first activity we need to perform in the analysis of a given set of DSLs is the identification of their commonalities. As explained in Chapter **??**, our definition of commonalities is based on the notion of specification clones. Then, we need to start by detecting the specification clones in a set of DSL variants. Once those clones are detected, Puzzle provides a mechanism to visualize and quantify those clones. The idea is to provide a set of metrics that allow language designers deciding if there is potential reuse enough to justify a reverse-engineering process. If so, Puzzle offers an automatic process to extract the specification clones in the form of reusable language modules. Besides, both commonalities and particularities are captured by synthesizing variability models. The reminder of this section is dedicated to illustrate each of the aforementioned capabilities.

Detection of Specification Clones

In order to enable detection of specification clones, Puzzle provides a set of comparison operators. These operators take into account not only the names of the constructs, but also the inter-constructs relationships and their semantics. The objective of those operators, is to find out all these segments in the specification that have been introduced through the copy&paste practice. The definition of the composition operators is deeply explained in Section 6.2. However, at implementation level, Puzzle is flexible enough to permit the definition of new comparison operators. Hence, the detection strategy can be easily improved or adapted to particular contexts.

Once those clones are detected, Puzzle permits to visualize them in the form of a Venn diagram. Using these diagrams, language designers can have an idea about the existing commonalities among the involved DSLs. Fig. 7.5 shows the Venn diagrams generated by Puzzle for the case of the example of state machines. It detects specification clones between UML state machines, Rhapsody, and Harel's state machines and at the level of abstract syntax and semantics. The reader can also see a video[5] that shows the process.

Quantification of Potential Reuse

Visualizing specification clones in terms of Venn diagrams can be useful to get a preliminary idea on the potential reuse existing among a set of DSL variants. However, language designers might need further information to decide if such a potential reuse is enough to justify a reverse-engineering process. To address this issue, Puzzle comes out with a set of

[5] Video: Detection of specification clones: https://www.youtube.com/watch?v=uN0tb9TYuyQ

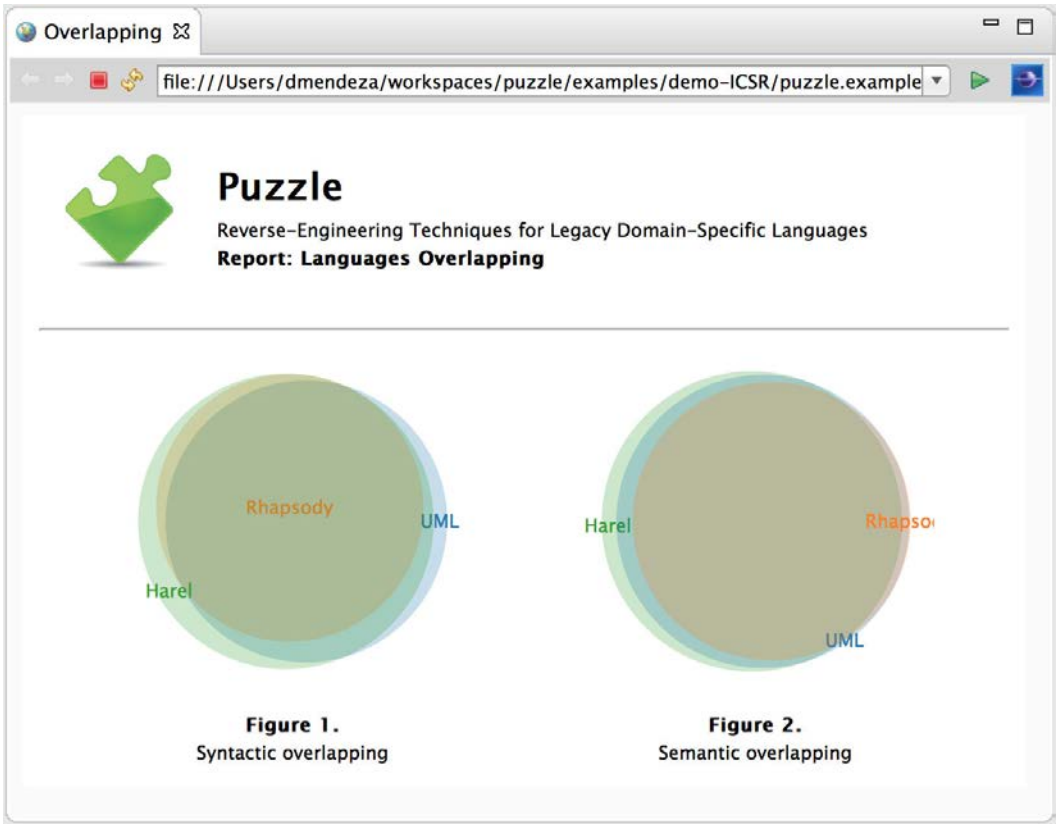


Figure 7.5: Venn diagrams provided by Puzzle for visualizing specification clones

metrics (illustrated in Fig. 7.6 and inspired in [147]) to quantify the potential reuse emerging from the existing specification clones. Those metrics are briefly described below. The objective is to provide a mechanism that allows language designers to estimate (in an objective fashion) the benefit of a reverse engineering process intended to remove specification clones in a given set of DSL variants.

Note that all the metrics are presented in the form of charts implemented as HTML reports that can be easily shared and published.

(1) **Size of Commonality** (SoC)**:** The SoC metric measures the amount of specification elements that are common to all the given DSLs. These specification elements are also known as the *core* of the involved DSLs in the sense that, if we synthesize a language product line, these concepts will be present in all the configurations.

(2) **Product-Related Reusability** (PRR_i)**:** The PRR_i metric measures the percentage of common elements of each DSL with respect the core. It provides an idea on how similar is the DSL with respect to the core of the DSLs.

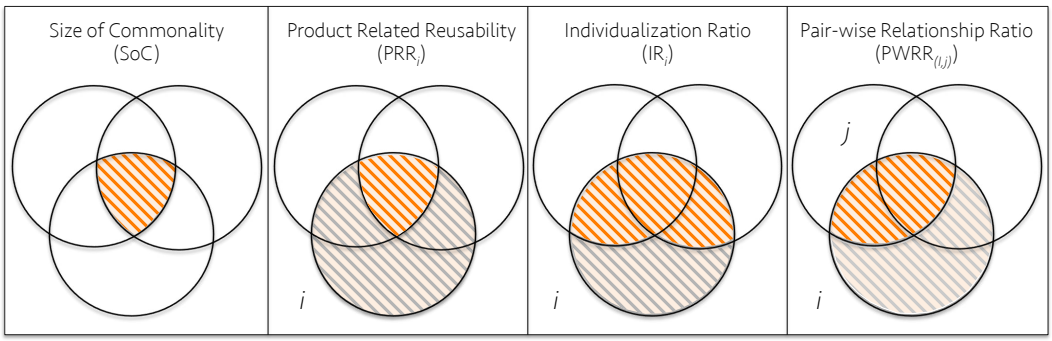


Figure 7.6: Metrics for quantification of potential reuse

(3) **Individualization Ratio (IR_i):** The IR_i metric measures the percentage of specification clones of each DSL with respect the rest of the DSLs. In other words, it shows for each product the amount of constructs/methods that are included in at least another DSL.

(4) **Pairwise Relationship Ratio ($PWRR_{(i,j)}$):** The $PWRR_{(i,j)}$ metric measures the percentage of common elements of each DSL with each of the other DSLs. Concretely, it shows, for each DSL, the amount of constructs/methods that are included each of the other DSL.

We provide a video[6] that shows the manner in which Puzzle computes and deploys the results of the quantification process described above.

The Reverse-Engineering Process

Once the specification clones existing in a set of DSLs are detected and quantified, then we can proceed to the reverse engineering process that ends up in a language product line. As deeply explained in Section 6.1, this process starts with the recovering of a language modular design, and finishes with the synthesis of the corresponding variability models.

The support that Puzzle provides to this reverse engineering process is illustrated in video[7]. At the end of this process, the language designer will have a result as the one presented in the screen-shot introduced in Fig. 7.7. On one hand, the Puzzle creates a visual diagram representing the extracted language modules. Besides, the languages are actually broken down in several projects that can be viewed in the Project Explorer view. The corresponding variability models are generated as part of the process as well.

[6]Video: Quantification of spec. clones: https://www.youtube.com/watch?v=yfkwrxk_uF8
[7]Video: Extraction of language modules: https://www.youtube.com/watch?v=sqF2NQMbGxY

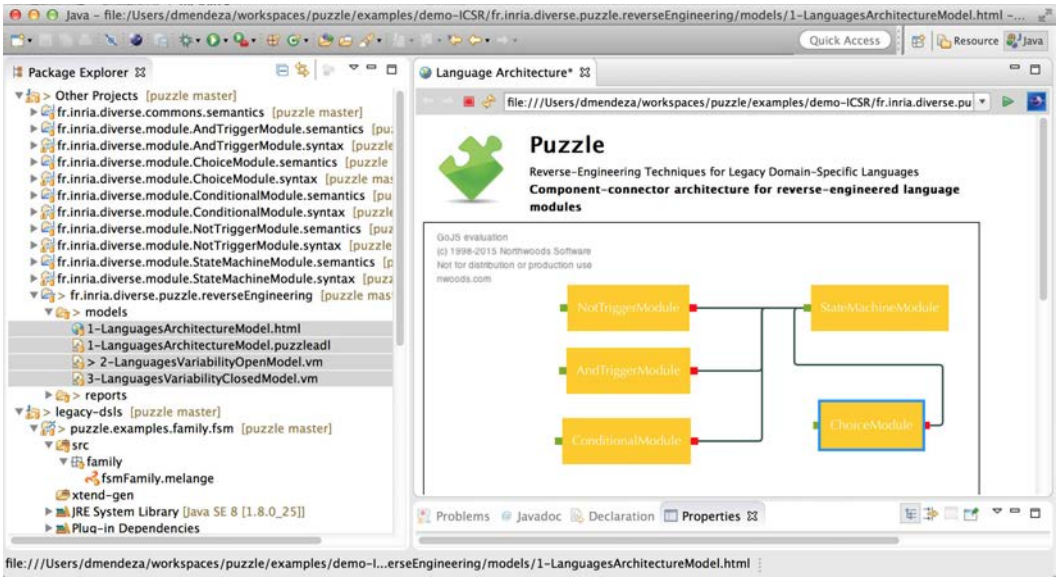


Figure 7.7: Metrics for quantification of potential reuse

7.3 Architecture

Let us close this implementation chapter with a brief description of the architecture we designed for Puzzle. It is composed of two parts illustrated in Fig. 7.8: the infrastructure and the superstructure. The *infrastructure* is a set of plug-ins that enable the specification of DSLs according to the technological space that we target in this thesis. Notably, the Puzzle's infrastructure is composed of EMF, Xtend, K3, and Melange. In turn, the *superstructure* is a set of plug-ins that provides analysis and reverse-engineering techniques on the DSLs specified on top of the infrastructure.

The plug-ins of the Puzzle's superstructure can be divided into seven categories according to their functionalities: core, composition, language architecture, language variability, comparison, metrics, and reverse-engineering. Each of those categories are briefly explained below:

(1) **Ui.** The plug-ins in this category implement the menu options and editor facilities provided by Puzzle to facilitate the interaction with language designers.

(2) **Composition.** These plug-ins implement the composition functionality for language modules. This functionality includes the different strategies for matching and merging of both abstract syntax and semantics.

(3) **Language architecture.** These plug-ins implement the metamodel and corresponding editors for the description of a set of language modules and the dependencies among them.

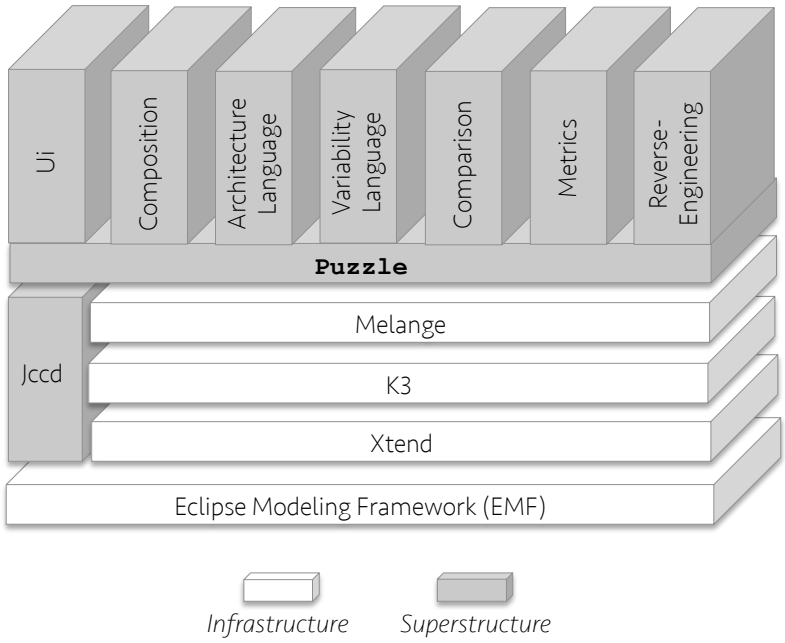


Figure 7.8: Software architecture of Puzzle

(4) **Language variability.** These plug-ins implement the metamodel and corresponding editors for the description of language variability models i.e., feature models for abstract syntax variability and orthogonal variability models for semantic variability.

(5) **Comparison.** Comparison plug-ins implement the comparison operators needed to detect specification clones at the level of abstract syntax and semantics (for the case of comparison of semantics, Puzzle uses JCCD [141].

(6) **Metrics.** The metrics plug-ins compute a set of metrics for the detected specification clones and present the results as a set of HTML reports that display those metrics in the form of charts.

(7) **Reverse-engineering.** The reverse-engineering plug-ins implement the algorithms that extract reusable language modules from the detected specification clones.

7.4 Summary

This chapter presented Puzzle, a tool that implements all the ideas introduced in the contribution of this thesis. Along the chapter, we introduced K3 and Mélange, which used together constitute a language workbench for specification of executable DSLs. Then, we show how we extended this language workbench to support language product lines engineering.

Chapter 8

Validation: Case Studies

In this chapter, we present the validation of this thesis. To this end, we introduce three case studies that show the applicability of our ideas in concrete scenarios. Fig. 8.1 shows an overview of these case studies with respect to the contributions of the thesis; there are two case studies intended to validate our contributions on top-down language product lines, and one case study doing the proper for our contributions on bottom-up language product lines.

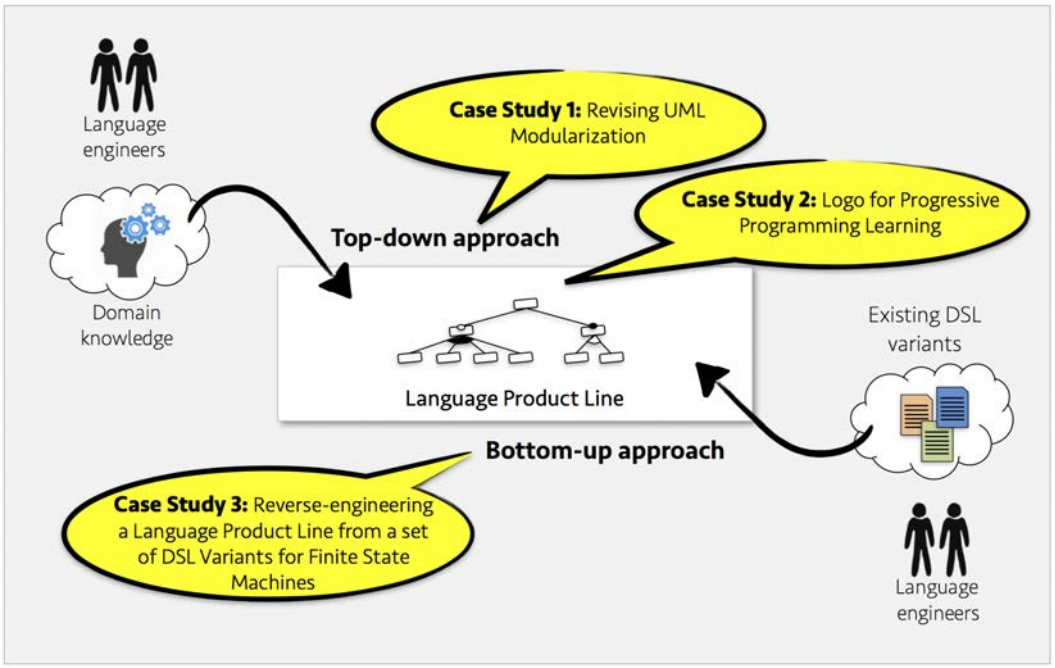


Figure 8.1: Modules for the UML case study

To validate our contributions on top-down language product lines, we start by validating our languages modularization approach since we consider that it is the backbone of the contribution. This validation is based on revisiting the modularization of UML. In other words, we will show that our language modularization approach is expressive enough to support the modular design used in the specification of UML. Afterwards, we present a case study that focus not only in modularization but also in variability management. This case study corresponds to a language product line for different versions of a basic programming language that can be used for teaching purposes.

To validate our contributions on bottom-up language product lines, we use the case study of the state machines that we have used all along this thesis as illustrative example. As a matter of fact, this case study comes from an industrial context. It is one of the motivations of the VaryMDE project, a bilateral collaboration between Thales and INRIA.

8.1 Revisiting the Modular Design of UML

8.1.1 Problem: UML is a Composition of Several "Language Units"

The objective of this first case study is to show the applicability of our language modularization approach in a complex scenario. To this end, we revisit the case of UML, which specification is a divided into different language modules (they use the term *language units*). Each module contains the specification of a type of diagram (e.g., use cases, classes, and deployments). Indeed, the UML specification starts with the following statement:

> *"The modeling concepts of UML are grouped into language units. A language unit consists of a collection of tightly- coupled modeling concepts that provide users with the power to represent aspects of the system under study according to a particular paradigm or formalism. For example, the State Machines language unit enables modelers to specify discrete event-driven behavior using a variant of the well-known statecharts formalism, while the Activities language unit provides for modeling behavior based on a workflow-like paradigm."*

To validate our approach, we implement the UML specification while respecting its modular design. We want to check whether the complex dependencies among the different types of UML diagrams can be expressed through our approach.

8.1.2 Solution: Language Interfaces

Fig. 8.2 introduces a diagram that shows the language modules of UML and the dependencies among them. To implement the case study, the each module has been specified in an Ecore model and the dependencies among them are expressed through the required and provided interfaces defined by our approach. The remainder of this section is dedicated to report on the results that we obtained. However, explaining the entire case study be too long.

Hence, we chose two language modules i.e., *Classes* and *Deplyments* that we consider as representative and we explain them in detail.

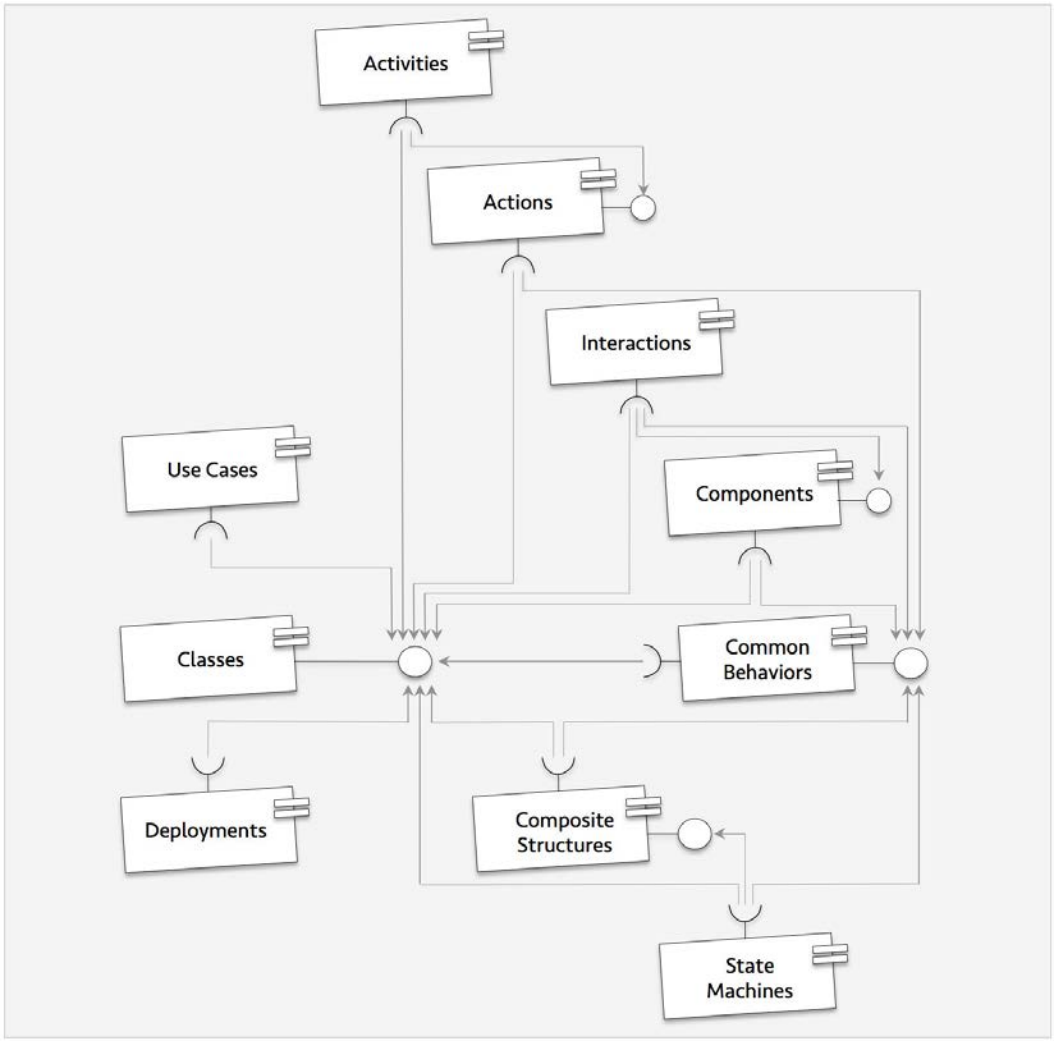


Figure 8.2: Modules for the UML case study

Classes vs. Deployments. Fig. 8.3 shows the abstract syntax specification for two language modules UML: *Classes* and *Deployments*. The first one offers the functionality to express class diagrams whereas the second one does the proper for deployment components. Note that the *Classes* module provides a number of abstractions that result useful for other modules including the *Deployments* one that uses and extends several of those abstractions. For example, The meta-class called *Artifact* uses the meta-classes *Operation* and *Property*

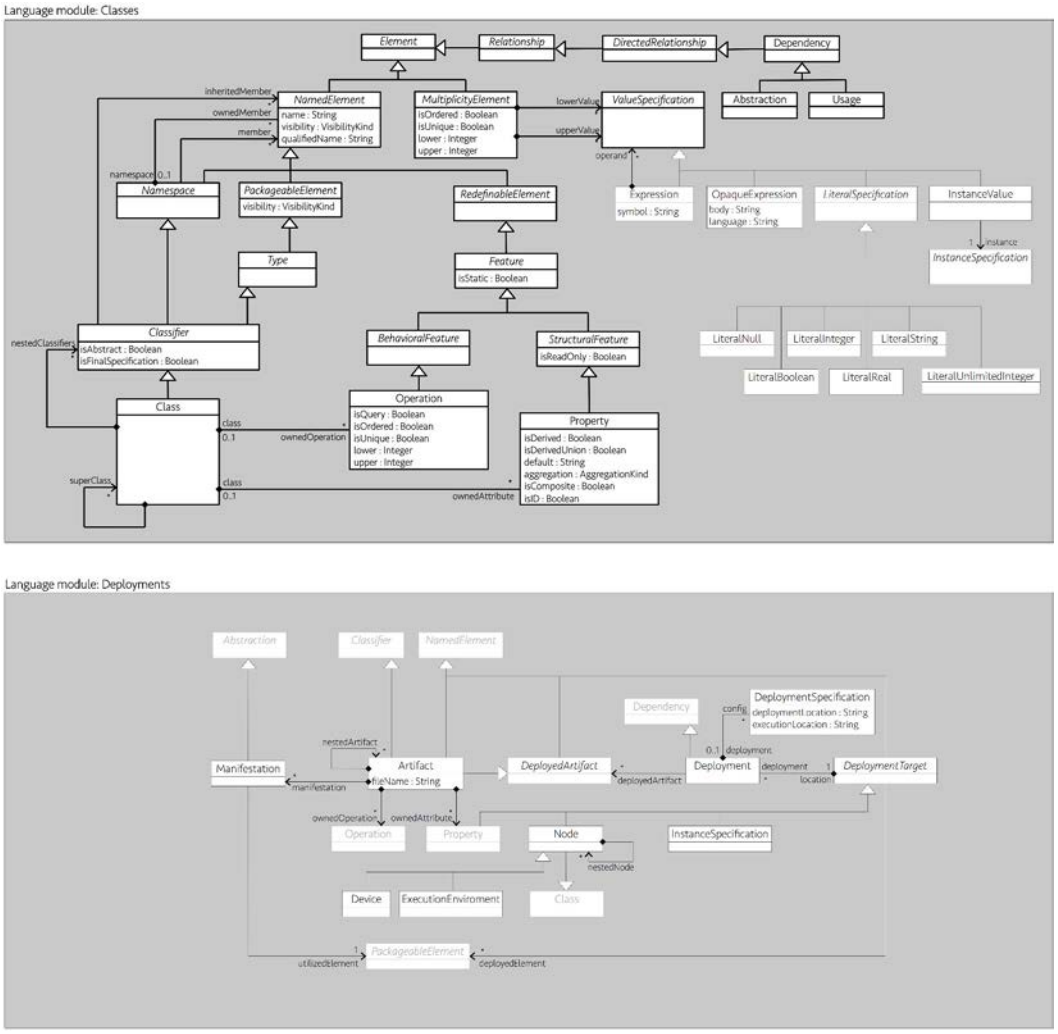


Figure 8.3: Modules for the UML case study

as part of its specification. Besides, the *Artifact* meta-class is a sub-class of the *Classifier* and *NamedElement* meta-classes.

This is probably one of the most important learned lessons that we got from this case study. We can find more than one modularization scenario in the interaction of two language modules: the meta-class *Artifact* uses (by aggregation) the meta-classes *Operation* and *Property* and, at the same time, extends the meta-classes *Classifier* and *NamedElement*.

Another interesting situation is the case of the meta-class called *Property* which is extended through a new super-class. That means that in languages modularization, the open-class approach should permit not only the injection of new properties and operations but also new super-types. Thanks to the composition algorithm we use and the notion of re-

quired interface that permits to have non virtual specification elements within virtual meta-classes, we can support such requirement.

The most challenging issue that we addressed during the implementation of the case study corresponds to the definition of the provided interfaces for the language modules. It is difficult to realize what are the specification elements that represent the core functionality of a language module versus the ones that correspond to implementation details. Indeed, one of the problems we faced to this definition is that the requirements of the language modules are already defined. Hence, we are afraid that the definition of the provided interface is biased. More than the elements that represent core functionality of each language module, we tended to include those that are used by other modules.

We claim that, as the same as in the general case of software architecture, there is still room to the definition of correct methodologies that provide insights to guide the design of both provided and required interfaces. For the case of provided interfaces, we need to establish what is the meaning of ”representing functionality” and evaluating the risk of evolution of each of the specification elements. For the case of the required interfaces, we need to have mechanisms that allow us to identify the more general abstractions required by a certain language module in order to increase the number of language modules that can fulfill those abstractions.

The source code of this case study can be found on GitHub[1].

8.2 Logo for Progressive Programming Learning

The objective of this case study is to show the applicability of our contributions on top-down language product lines in a specific domain with well-defined needs. To this end, we chose a case study on education in computer science, which has been inspired by previous works in the literature [95, 10]. It focuses on the definition of several DSL variants to facilitate progressive learning of programming skills.

The main principle is to define a set of incremental learning levels for a group of students, and support each level with a dedicated language variant, which contains only the constructs that the students will learn during the level. Once the students master all the constructs in a given level, they can upgrade to the next one. It will be supported by a new language variant including the new constructs that constitute the next learning challenge.

Note that the notion of progressive learning is not necessary restricted to the existence of different DSL variants. Contrariwise, it can be achieved by using only one version of the programming language containing all the constructs that the students will learn along the entire learning process. In such a case, each learning level is dedicated to explain a subset of those constructs and students are asked to omit the other ones.

The problem with this approach is that students are exposed to many constructs at the

[1] https://github.com/damende/puzzle/tree/master/examples/uml

same time in the programming tooling (i.e., editors) even if they are learning just a subset of those constructs. This increases the cognitive load for the students making more difficult the learning process [95]. Ideally, each level should be supported with a dedicated tool containing only the constructs for each level [95].

8.2.1 Problem's Description: Learning Sequences and DSL Variants

A learning sequence is an *"ordering of the learning space's concepts into linear sequences in which those concepts could be learned"* [148]. In the context of programming learning, a learning sequence can be concertized as an ordered set of learning levels, each of which contains a well-defined set of programming skills. Once the student achieve the programming skills defined in a given level, then he/she is permitted to advance to the next one. The design of a learning sequence is responsibility of the teacher and it might vary and/or evolve depending of different factors such as the profile of the students [149].

As aforementioned, the notion of DSL variants can be used to support the design and implementation of learning sequences of programming skills [10]. Each learning level introduces a subset of language constructs that the student should learn, so each level can be supported by a specific DSL variant including those concepts. However, this idea might restrict the flexibility of the teachers to modify their learning sequences. The evolution of the learning sequence might imply the construction of new DSL variants supporting the changes in the levels. For example, if a teacher decides to move a topic from one level to another, then the DSL variant should be modified to remove the constructs that should be added to the DSL variant of the next level.

There work presented by Cazzola et al. [95] proposes a definition to this problem. In particular, the authors provide a language product line that supports the construction of the tool support for the learning sequences. The language variant supporting each learning level is produced by configuring and assembling a product in the language product line. Fig. 8.4 illustrates this idea. In the figure, there are three incremental learning levels; each level is supported by the DSL variant resulting of a given configuration. Note that the configurations are incremental; level 2 includes more language features than level 1, and level 3 introduces more language features than level 2.

When using language product lines for supporting the design and implementation of learning sequences, teachers have more flexibility to define and modify the learning levels since the tool support can be easily prototyped.

8.2.2 Solution: A Top-Down Language Product Line

Once we have explained the motivation of the case study, we proceed to explain how to apply our approach on top-down language product line to solve the problem. In particular, we will use the meta-language facilitates introduced in Section 5.1 in junction with the methodology proposed in Section 5.2 to build a language product line that supports incremental

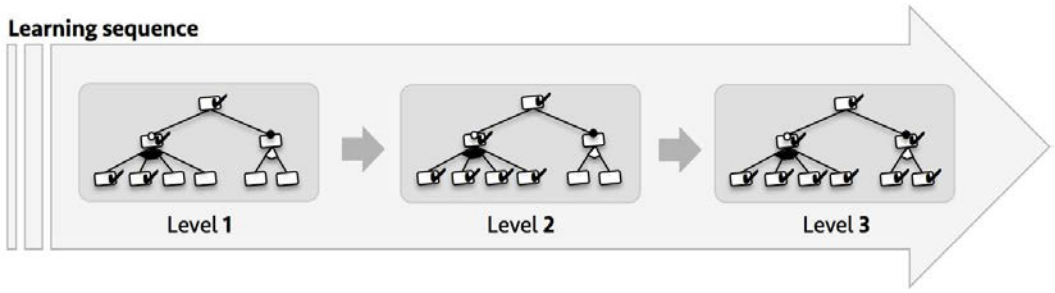


Figure 8.4: Learning Sequences and DSL Variants

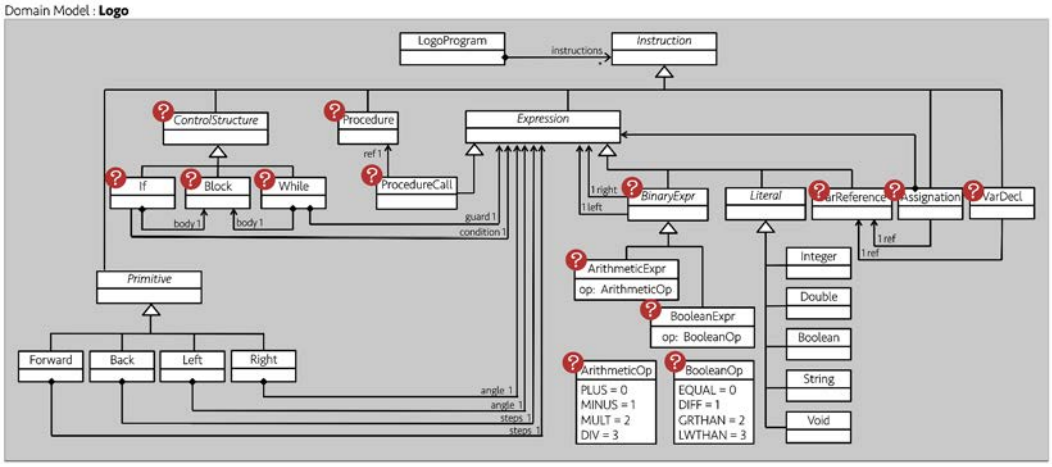


Figure 8.5: Domain model for the Logo case study

programming learning. In this case study, we will use Logo as programming language. This language has been largely used during the last decades to teach programming skills to high-school students.

Structuring the domain knowledge. According to our methodology for the construction of top-down language product lines, the first step is to structure the domain knowledge. In this case, we want to represent all the language constructs that the students will have to learn along the entire learning process. To this end, we use a class diagram which is introduced in Fig. 8.5. This class diagram corresponds to a complete Logo language including not only the typical movement primitives (i.e., forward, back, left), but also imperative programming concepts such as variables, procedures, control structures, and binary expressions. At the end of the learning process, students are intended to master not only the basic primitives, but also these imperative programming constructs that will facilitate the comprehension of more sophisticated programming languages such as Java or C++.

Identifying syntax variability. Once we have structure the domain knowledge in a class diagram, we need to understand what are the optional and mandatory concepts of the

language. This allows us to identify the minimal set of constructs that a DSL variant for Logo must own in order to be fully executable. In the particular case of this case study, this minimal set of constructs can be considered as the constructs introduced in the very first level of the learning sequence.

In Logo, the minimal set of constructs corresponds to the basic movement primitives and literals. Using those constructs, students will be able to draw simple geometric forms. All the other constructs (marked with (?) in the Fig. 8.5) are optional and the order in which they will be learned by the students will depend on the definition of the learning sequence proposed by the teacher.

Identifying language modules. Let us now to find out a good modular design for the language product line. That means, to find an accurate way of grouping the language constructs of Logo that facilitates the definition of learning sequences.

In order to achieve this modular design, we have the cluster analysis provided in Section 5.2. That means that we can use a hierarchical domain analysis based on the cohesion of the constructs to identify clusters of language constructs that can go well together in the same module. In doing so, our hypothesis is that the hierarchical domain analysis based on cohesion will suggest clusters of language constructs that we can easily teach to the students in the same level.

Fig. 8.6 presents the results of applying hierarchical domain analysis on the domain model for the Logo language introduced in Fig. 8.5. These results are quite promising; we can identify certain subsets that, intuitively, can be learned together in a programming course. For example, we have the a first cluster grouping all the basic movement primitives. Similarly, we can see that literals appear to be located in another module as well as control structures and procedures. Indeed, we realize that this grouping fits many of the classical learning sequences that we can find in programming courses. The course starts with simple concepts such as console printing instructions, then the introduction of procedures, and then some control structures.

Build variability models. The resulting variability model after applying our algorithm is presented in Fig. 8.7. Such a model contains one feature by each module identified in the last step. There is a mandatory feature including the basic movement primitives. Similarly, there is a mandatory feature representing literals. This is the minimum set of constructs that should be included to obtain a fully executable logo language. The other features are optional. The selection of those features will respond to a configuration process associated to the design of the learning sequence. For example, a possible learning sequence might include the notion of procedures before including control structures. Because this case study does not include semantic variability, there is no orthogonal variability model.

The source code of this case study can be found on GitHub[2].

[2] https://github.com/damende/puzzle/tree/master/examples/logo

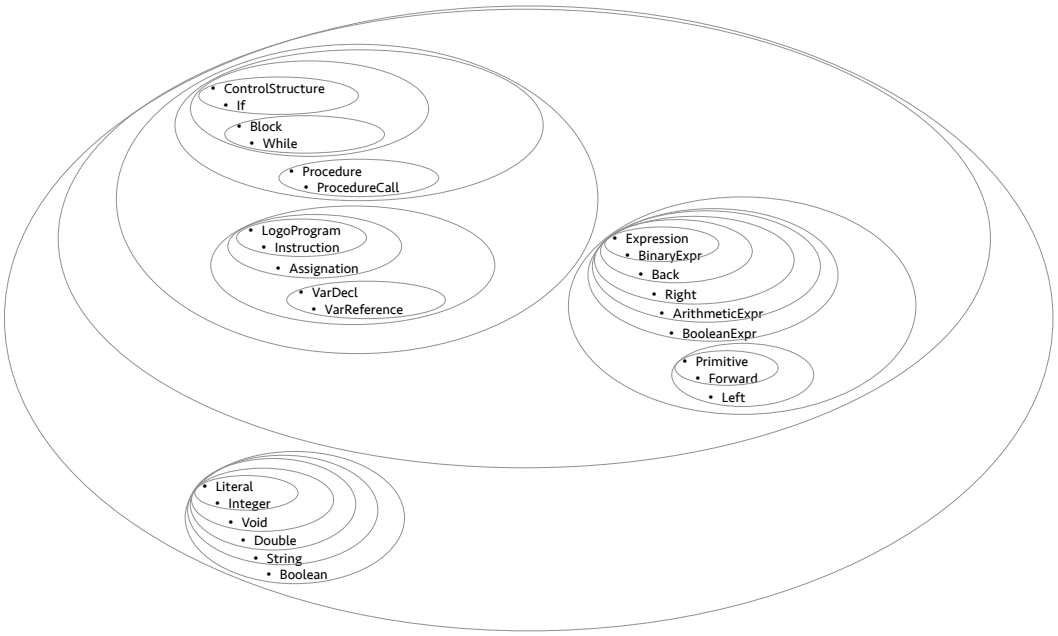


Figure 8.6: Venn diagram for the hierarchical domain analysis for logo

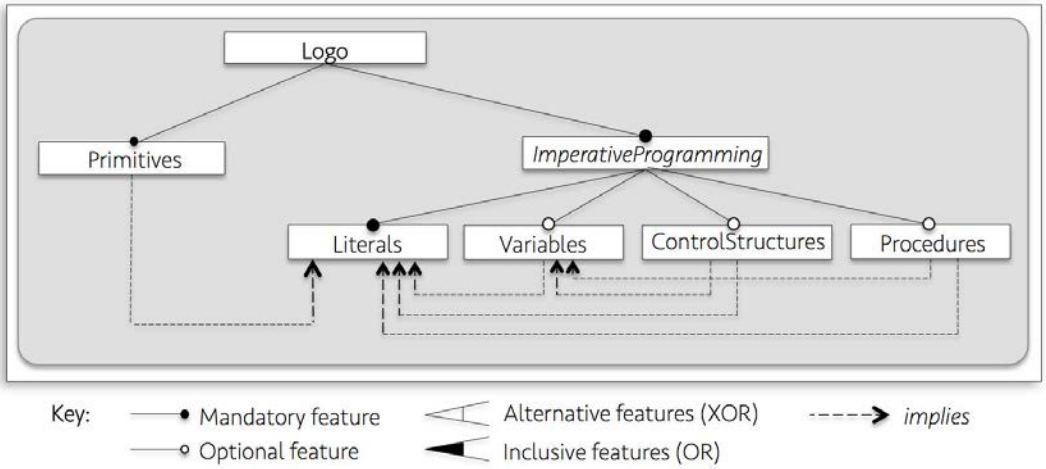


Figure 8.7: Features model for the variability of the Logo case study

8.3 Reverse-Engineering a Language Product Line for FSMs

The objective of our third case study is to show the applicability of our contributions on bottom-up language product lines in a set of DSL variants. To this end, we use as case study the set of DSL variants for finite state machines. Although we already use this case study as running example to illustrate the ideas of this thesis. It is quite complex and rich so we decided to re-use it in the validation of this thesis. In addition, it is inspired from a real industrial problem that we obtain from ou collaboration to Thales Research & Technology along the VaryMDE project.

This case study is inspired from the analysis of variability on languages for finite state machines provided by Crane et al. [59], and it is composed of three different DSLs: UML state diagrams, Rhapsody, and Harel's state charts. As aforementioned, these DSLs have some commonalities since they are intended to express the same formalism. According to the development scenario we address in this thesis –i.e., bottom-up language product lines where the DSL variants were built through the clone-and-own approach–, these commonalities will be materialized as clones in the DSL specifications. In this section, we summarize both commonalities and differences existing in the case study. Then, we apply our approach and we present the obtained results.

8.3.1 Problem Description: Several Formalisms for FSMs

Generally speaking, state machines are graphs where nodes represent states and arcs represent transitions between the states [150]. The execution of a state machine is performed in a sequence of *steps* each of which receives a set of events that the state machine should react to. The reaction of a machine to set of events can be understood as a passage from an initial configuration (t_i) to a final configuration (t_f). A configuration is the set of active states in the machine.

The relationship between the state machine and the arriving events is materialized at the level of the transitions. Each transition is associated to one or more events (also called triggers). When an event arrives, the state machine fires the transitions outgoing from the states in the current configuration whose trigger matches with the event. As a result, the source state of each fired transition is deactivated whereas the corresponding target state is activated. Optionally, guards might be defined on the transitions. A transition is fired if and only if the evaluation of the guard returns true at the moment of the trigger arrival.

The initial configuration of the state machine is given by a set of initial pseudostates. Transitions outgoing from initial pseudosates are fired automatically when the state machine is initialized. In turn, the execution of a state machine continues until the current configuration is composed only by final states (an special type of states without outgoing transitions).

Figure 8.8 illustrates the aforementioned behavior with a simple state machine. It is composed of the states S_1, S_2, and S_3. The state S_1 is the entry point of the state machine.

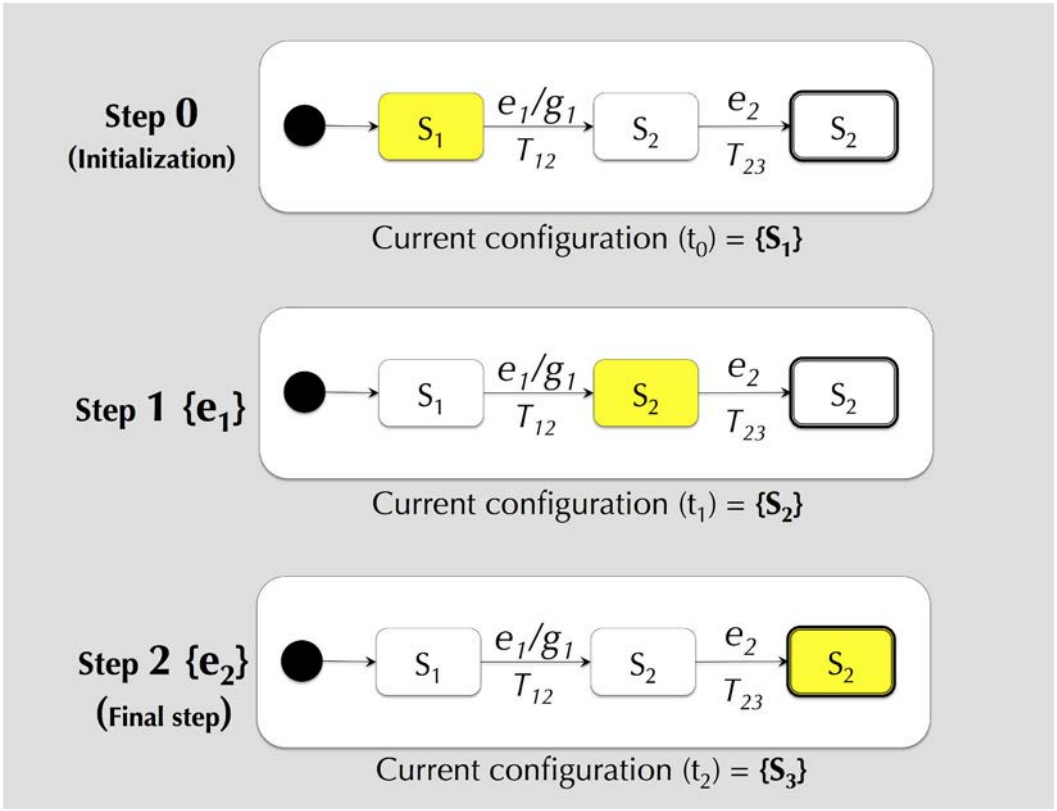


Figure 8.8: Example of a simple state machine and its execution

This is expressed by means of the initial pseudosate (the filled circle) containing a transition towards S_1.When the event e_1 arrives, the state machine evaluates the guard g_1 and, if the evaluation returns true, the transition T_{12} is fired. Then, the state S_1 is deactivated and the state S_2 is activated. In the second step, event e_2 arrives and the final state S_3 is activated after firing the transition T_{23}.

Regions. All of the DSLs for expressing state machines that conform our family support the notion of region. A state machine might be divided in several regions that are executed concurrently. Each region might have its own initial and final (pseudo)states. Figure 8.9 illustrates a state machine with two regions. Note that each region of the state machine contains its proper initial pseudostate.

Actions. Note that a state machine has some additional capabilities with respect to the ones introduced so far. States can define entry/do/exit actions. Similarly, transitions can have some effects. States' actions and transitions' effects are intended to interact with the environment of the state machine. The environment is a mapping that associates variables with values.

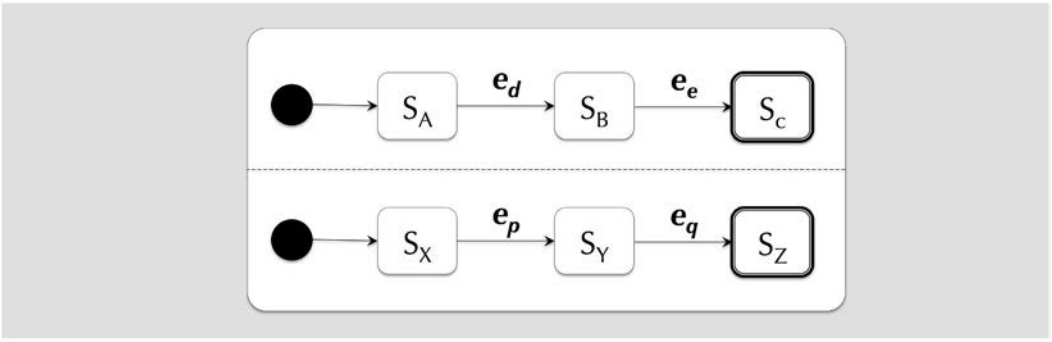


Figure 8.9: Example of a simple state machine with two regions

Abstract Syntax Variability

Let us introduce the syntactic variation points present in the family of DSLs for state machines. These semantic variation points are presented by showing the constructs available in the language and their availability in each of the involved DSLs

Triggers: disjunction, conjunction, and negation. The first syntactic variation point of this case study refers to the relationship between triggers and transitions. As we said earlier, a transition T is associated to certain event e. T is fired when e is dispatched. This relation can be more complex, however. The transition T can be associated to the negation of an event i.e., not e. As a result, it will be fired in every step where its target state belongs to the current configuration and the associated event *does not arrive*. Moreover, DSLs for state machine languages often offer the capability of associating multiple events to a transition combined through classical logical operators such as `OR` and `AND`.

In terms of triggers combination, Rhapsody is the more restrictive DSL where transition can be only associated to one event. Conversely, in UML a transition can be triggered by several events associated to the `AND` logical operator. In Harel's state charts are even more expressive and allow conjunction and negation of events.

Pseudostates: Pseudo-states are special types of states that allow the expression of compound transitions. For example, as illustrated in Fig. **??** the pseudo-state `Fork` enables the bifurcation of a transition so different states can be executed in parallel. Similarly, the pseudostate `Join` enables the unification of two transitions outgoing from concurrent states. There are other types of pseudostates such as the history ones that stores a reference to the last sub-state executed in a composite state. Conditional pseudostates are also available enabling different execution paths for the state machine according to the value of the variables in the environment.

Formalisms configuration: There are certain differences in the DSLs regarding the pseudostates they support. All the tree DSLs offer the initial pseudo-states, as well as fork, join, junction and deep history. Shallow history is only supported by UML and Harel's state

Language vs. Construct	StateMachine	Region	AbstractState	State	Transition	Trigger	NotTrigger	AndTrigger	OrTrigger	Pseudostate	InitialState	Fork	Join	DeepHistory	ShallowHistory	Junction	Conditional	Choice	FinalState	Constraint	Statement	Program	NamedElement	Total
UML	●	●	●	●	●	●	-	●	-	●	●	●	●	●	●	●	-	●	●	●	●	●	●	20
Rhapsody	●	●	●	●	●	●	-	-	-	●	●	●	●	-	●	●	●	-	●	●	●	●	●	18
Harel	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	-	●	●	●	●	●	22

Figure 8.10: Diversity of constructs provided by the DSLs for state machines

machines. A particular pseudo-state for representing dynamic choice is offered by UML. Static choice is supported by Rhapsody and Harel's state charts.

Summay. Figure 8.10 summarizes the discussion presented above by indicating the language constructs provided by each DSL.

Semantic Variability

The reminder of this section presents the semantic variation points existing in our family of DSLs for state machines.

Events dispatching policy: The first semantic difference in the semantics of state machines refers to the way in which events are consumed by the state machine. In a first interpretation, simultaneous events are supported whereas in a second interpretation the state machine follows the principle of run to completion. Let us discuss each scenario.

- **Simultaneous events:** There are DSLs for state machines that support this capability so a step consumes a subset of the events where the size of the subset can be greater than one and attends all of them at the same time. For example, consider the state machine presented in Fig. 8.11. In that case, the events e_1 e_3 arrive at the same time in the step 1 and they are attended simultaneously. As a result, the configuration of the machine after that step is S_2, S_5.
- **Run-to-completion:** A different interpretation of this variation point is to comply to the run to completion policy. That means that the state machine is able only of supporting one event by time and it cannot consume another one until de execution of the current event is completed. For example, consider the state machine presented in Fig. 8.11. Differently from the later case, in this one the events e_1 e_3 that arrive at the same time in the step 1 are attended one by one. As a result, the configuration of the machine after that step is S_2, S_4 and an automatic additional step is dispatched then. After that second step, the configuration of the machine is S_2, S_5.

Note that in the example we presented for the sequential events interpretation, we show that the state machine executes the events in a given order that corresponds to the order in which the events are listed. However, it is important to clarify that it is not always like that. In fact, there is no notion of order in the consumption of events that arrive in the same

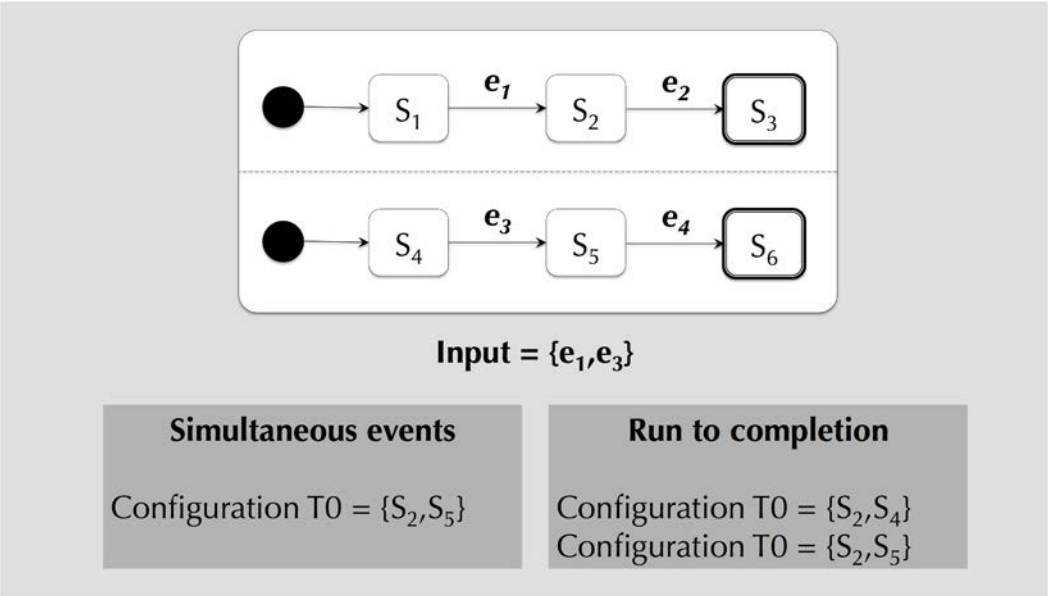


Figure 8.11: Difference between simultaneous events and run to completion

step. This fact has been identified as a potential semantic variation point for state machines. However, it is out of the scope of this document.

DSLs configuration: The semantics of UML and Rhapsody fit the run to completion policy for events dispatching whereas Harel's statecharts support simultaneous events.

Execution order of transitions' effects: It is possible to define actions on the transitions that will affect the execution environment where transitions are fired. These actions are usually known as transitions effects. All the DSLs for state machines support the expression of such effects. However, there are certain differences regarding their execution.

- **Effects executed sequentially:** The first and most common way of executing the effects of a transition is by following the order in which they are defined. This is due to the fact that transitions effects are usually defined by means of imperative action script languages where the order of the instructions is intrinsic. Under this interpretation, once the event e_0 is dispatched, the state machine in Fig. 8.12 moves from state S_1 to the state S_3. This is because the value of the variable y depends on the value of x that has been previously modified in a precedent instruction.
- **Effects executed in parallel:** The second interpretation to the execution of transitions' effect is to execute them in parallel. In other words, the effects are defined as a set of instructions that will be executed at the same time so no assumptions should be made with respect to the execution order. Under this interpretation, once the event e_0 is dispatched, the state machine in Fig. 8.12 moves from the state S_1 to the state S_2 Although the value of y depends on the value of x, the instructions are executed

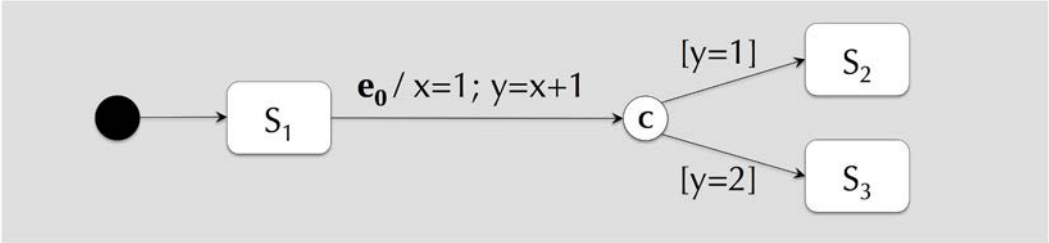


Figure 8.12: Example of a state machine with effects in the transitions

simultaneously.

DSLs configuration: UML, Rhapsody, and Stateflow execute the transition effects in parallel. Harel's statecharts execute transition effects simultaneously.

Priorities in the transitions: Because several transitions can be associated to the same event, there are cases in which more than one transitions are intended to be fired in the same step. In general, all the DSLs for state machines agree in the fact that all the activated transitions should be fired. However, this is not always possible because conflicts might appear. Consider for example the state machine presented in Fig 8.13. The transitions T_D and T_E are conflictive because they are activated by the same event i.e., e_2, they exit the same state, and they go to different target states. Then, the final configuration of the state machine will be different according to the selected transition.

In order to tackle such situations, it is necessary to establish policies that permit to solve such conflicts. Specifically, we need to define a mechanism for prioritizing conflicting transitions so the interpreter is able to easily select a transition from a group of conflicting transitions. One of the best known semantic differences among DSLs for state machines is related with these policies. In particular, there are two different mechanisms for solving this kind of conflicts:

- **Priority for deepest transitions:** A first mechanism for solving conflicting transition is to select the transition with the lower scope. That means, the transition that is deeper in the hierarchy of the state machine. In the example presented in Fig 8.13 the dispatched transition according to this policy would be the transition T_E so the state machine would move to the state S_5.

- **Priority for higher transitions:** The second mechanism for solving conflicts in the transition is to select the transition with the higher scope. That is, the transition in the higher level of the hierarchy of the state machine. In the example presented in Fig 8.13 the dispatched transition according to this policy is the transition T_D so the state machine will move to the state S_4.

DSLs configuration: The semantics of UML and Rhapsody fits on the first interpretation i.e., deepest transition priority whereas the semantics of Harel's statecharts fits on the second interpretation i.e., highest transitions priority.

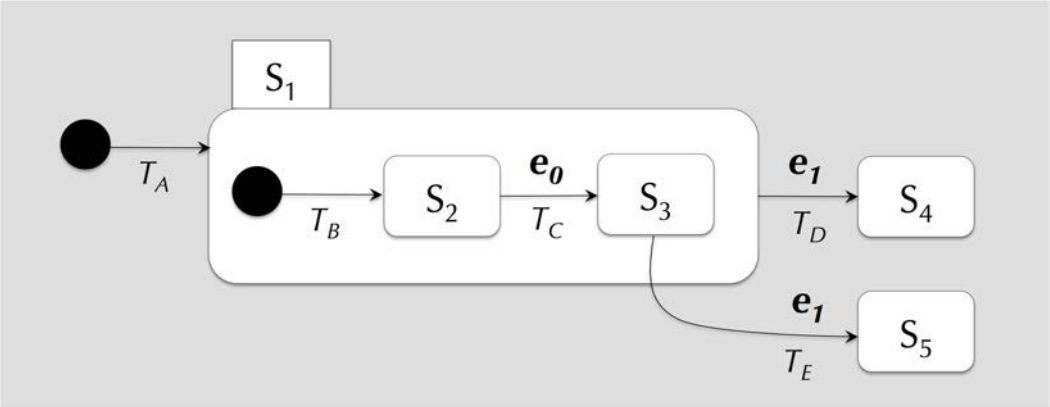


Figure 8.13: Example of a state machine with conflicting priorities

8.3.2 Solution: Reverse-Engineering a Language Product Line

The starting point of the applicability of our approach in the case study is a set of DSLs implementing each of the specifications explained above. Hence, at the beginning we have three different DSLs for state machines that can be accessed in a `GitHub` repository[3]. Using these specifications as input, we proceed to apply our approach.

The results are summarized in Fig. 8.14. At the left of the figure we present the set of language modules we obtained as well as the language interfaces existing among them. Those modules group the language constructs according to the heuristic introduced in Section 6.2 on breaking down intersections. At the right of the figure we show the corresponding variability models. Each feature of the feature models is associated to a given language module. In turn, the semantic variability points in the orthogonal model are associated to clusters of domain specific actions.

Analysis of the results. Let us now discuss the results of the case study. As expected, we obtained a language product product line from a set of DSL variants for finite state machines. But... Does this product line identify all the variation points and commonalities existing in the DSL variants? Are those variation points properly specified in the language modular design and variability models? Since we know these variation points and commonalities, we can check whether they are appear in the produced language product line. The results of this verification are presented in Table 8.1.

The results are promising in the case of abstract syntax variability. According to the Table 8.10, the DSL variants share 17 constructs in common. Those constructs are properly factorized in a language module that we named StateMachine. This module is correctly identified during the recovering of the language modular design, and it is properly specified

[3]GitHub repository for the case study: https://github.com/damende/puzzle/tree/master/examples/state-machines

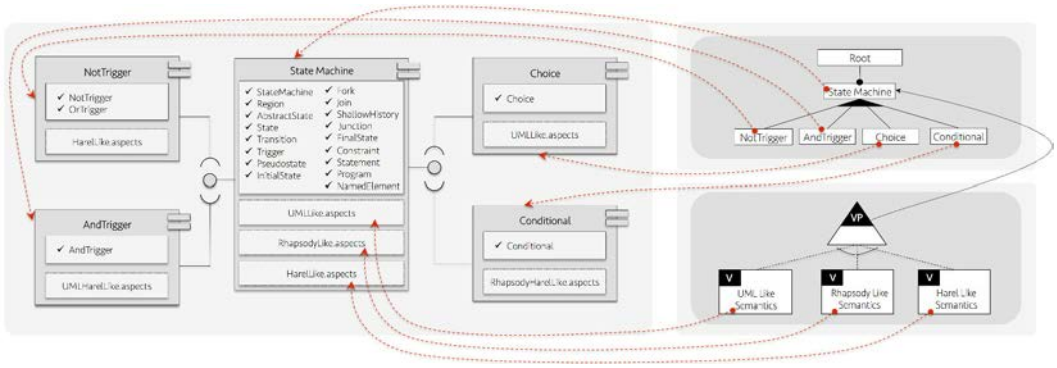


Figure 8.14: Language product line produced for the case study of the finite state machines.

Oracle	Result	
	Properly identified?	Properly specified?
Abstract Syntax Variability		
Module: *[StateMachine, Region, AbstractState, State, Transition, Trigger, Pseudostate, InitialState, Fork, Join, ShallowHistory, Junction, FinalState, Constraint, Statement, Program, NamedElement]*	✓	✓
Module: *[NotTrigger, OrTrigger]*	✓	✓
Module: *[AndTrigger]*	✓	✓
Module: *[Choice]*	✓	✓
Module: *[Conditional]*	✓	✓
Semantic Variability		
Events dispatching policy	✓	✗
Execution order of transitions' effects	✓	✗
Priorities of conflictive transitions	✓	✗

Table 8.1: Analysis of the results of the case study

as a language module in terms of a metamodel enhance with domain specific actions and offering a provided interface. Besides, the particularities of the DSL variants are also well factorized. There is a module that contains the constructs NotTrigger and OrTrigger that belong only to the variant complying the Harel' statecharts specification. Besides, there are three additional modules that contain the constructs AndTrigger, Choice, and Conditional respectively. Using this modular design, we can re-compose any of the three initial DSL variants.

The situation is different for the case of semantic variability. Although our reverse-engineering strategy is able to identify that the domain specific actions are different in the three DSL variants, the level of granularity at which those variation points are detected is coarser than one might expect. At the beginning of this section, we described three semantic variation points and their possible interpretations i.e., events dispatching policy,

execution order of transitions' effects, and priorities of conflicting transitions. Using the proposed technique, we can identify just one semantic variation point indicating that the language module called StateMachines contains three different clusters of domain specific actions, which is reflected in the orthogonal variability model.

This threat to validity of our technique can be explained by the fact that the analysis of commonalities and variability is conducted by means of static analysis. We can analyze the structure of the metamodels and the domain specific actions, but not their behavior at runtime. Hence, we cannot see how these differences impact the execution of the models. For example, we cannot infer that the differences among the domain specific actions in the StateMachine module impact the way in which conflicting priorities are managed. A next step in this research could be to use also dynamic analysis in the domain specific actions to better specify semantic variation points.

The source code of this case study can be found on GitHub[4].

8.4 Summary

This chapter was dedicated to the validation of our contributions. To this end, we presented three case studies. In the first one, we show that our approach for languages modularization is complete enough to support the modular design existing behind the specification of UML. The second case study is intended to validate our contributions on top-down language product lines engineering. This case study focuses on the construction of a language product line for facilitating different variants of a DSL used for gradual teaching of programming skills. Finally, the third case study is intended to validate our contributions in bottom-up language product lines. This case study corresponds to the implementation of the example on the different variants for state machines that we have used along this thesis to illustrate our ideas. We implemented each of those variants using the clone-and-own approach; then we use our approach to reverse-engineering the corresponding language product line.

[4] https://github.com/damende/puzzle/tree/master/examples/state-machines

Part IV

Closure

Chapter 9

Conclusion and Perspectives

9.1 Conclusion

A domain-specific language provides a set of abstractions that belong to specific area of endeavor (i.e., the domain) and that serve to a concrete purpose. When the same abstractions are used in different domains, or when the complexity of a certain domain demands various DSLs addressing different purposes, we obtain DSL dialects. As the same as in the case of natural languages, DSL dialects are variants of a given DSL that introduce some changes in terms of syntax and semantics.

The phenomenon described above supposes an important challenge for language designers. This is specially true in large technology companies that use the language-oriented development approach, and which business is based on providing numerous systems through diverse domains. The construction of a DSL is a complex activity, and it is even more challenging when language designers have to deal not only with DSLs, but also with different variants of the same DSL.

The community in software language engineering is currently seeking manners to ease-off the construction of DSL variants. The purpose is to increase **reuse** during the construction of such variants DSL by exploiting the abstractions and definitions they share among them. In that context, the notion of language product lines have recently appeared. Several state-of-the art woks have shown how to use the ideas on software product lines engineering in the construction of DSL variants.

In this thesis, we provided a deep analysis of current approaches supporting language product lines engineering. Our study was conducted in the form of a systematic literature review where we included a large spectrum of approaches published during the last 16 years. The main result of this study was a description of the life-cycle of language product lines based on the discussions provided in the research works included in the literature review. We used such a life-cycle to evaluate current approaches and to identify a set of open issues in the field of language product lines engineering.

These open issues can be divided into two categories according to the flow in which the activities of the life-cycle of a language product lines are addressed: top-down and bottom-up. We deeply explained each of those categories and we listed a specific set of limitations of current approaches. Afterwards, we prioritized those limitations and we selected a subset of them to address along these years of research. The contribution of this thesis can be summarized in two points:

First, we provided facilities for the construction of top-down language product lines. The facilities are a modularization approach which favors three classical software modularization principles i.e., independent development, substitutability, and information hiding. This approach is accompanied with an strategy to model the variability of a language product line, as well as a set of methodological insights that facilitate the daily basis of language designers.

As a second contribution of this thesis, we presented a reverse-engineering algorithm to support the construction of bottom-up language product lines. Our approach assumes the existence of a set of DSL variants that have been built through the clone-and-own approach. Based on that, we used static analysis to identify commonalities that we extract as a set of reusable language modules. Then, we provide an algorithm to synthesize the variability models of the language product line.

The validation of our ideas was conducted through a set of three case studies. In the first one, we revisit the design of UML as modeling language and we replicate its modular design using our modularization approach. This exercise allowed us showing that our approach is complete enough to support the definition of complex and large modeling languages. In the second case study, we provided a case study composing of several DSL variants for incrementally teaching programming skills. This case study allowed us illustrating the construction of language product line using a top-down approach. Finally, in the third case study, we use three different variants of a DSL to express finite state machines to reverse-engineer a language product line by using the bottom-up perspective.

9.2 Perspectives

In this section, we present different directions in which the work presented in this thesis can be pursued. We start by providing a discussion on how to broaden the spectrum of our contributions to more diverse development scenarios and in other technological spaces. Then, we discuss the open issues that we find during the conduction of the state of the art, and which were not covered in the scope of this thesis i.e., testing and evolution of language product lines.

9.2.1 Broadening the Spectrum of our Contributions

Thinking outside executable metamodeling. All along this thesis, we have insisted in the fact that all our ideas are applicable when the DSLs under study are developed by means

of metamodels enhanced with operational semantics. We have described the particularities of such a technological space, and we provided the tooling enough to support our approach in that context. However, what if during language development the DSL variants are implemented through a different technological space? At which point our approach can be re-used?

We need to start by admitting that all the implementation part of our approach –i.e., the tooling encapsulated in the Puzzle toolkit–, can be hardly extended to other technological spaces. This is due to the fact, that Puzzle was built on top of a language workbench which is exclusively dedicated to metamodel-based DSLs which semantics is expressed operationally through domain-specific actions. However, many of the ideas that we have developed during this thesis can be reused in other language workbenches dedicated to other technological spaces. We consider that the answer of this question represents a relevant direction for future work.

Thinking outside the copy-and-own approach. An important constraint of our approach for bottom-up language product lines is that it is scoped to DSLs that have been built through the clone-and-own approach. This fact permit to assume the existence of specification clones which is the backbone of our strategy for reverse engineering language modules. But... what if we have DSLs that are not necessarily built in those conditions? Suppose for example that we have as input a set of DSLs that share certain commonalities but that have been developed in different development teams. In that case, the probability of finding specification scenarios is quite reduced, and our approach will not be useful. How our strategies can be extended to deal with such a scenario?

According to our experience during the development of this thesis, we think that the answer to that question relies on the definition of more complex comparison operators. As we deeply explain in Section 6.1, the very first step of our reverse engineering strategy is to perform a static analysis of the given DSLs and apply two comparison in order to specify specification clones. If what we want is to find commonalities that are not necessarily materialized in specification clones but in ”equivalent functionality”, then we need to enhance the comparison operators in order to detect such as equivalences.

Note the complexity behind the notion of ”equivalent functionality”. In the case of abstract syntax, two meta-classes might provide equivalent functionality by defining different language constructs e.g., using different names for the specification elements and even different relationships among them. In the case of the semantics, two different domain specific actions might provide equivalent functionality through different programs. We claim that further research is needed to establish this notion of equivalence thus supporting more diverse development scenarios.

9.2.2 Testing and Evolution of Language Product Lines

Two of the open issues that we identified during our study of the state of the art are testing and evolution of language product lines. In Section 3.4, we explained the importance of

this phases of the life-cycle of a language product line. However, those issues are out of the scope of this thesis. Their complexity demands a deep analysis, which can be the subject of study of future work.

In addressing these issues, researchers will have to consider some of the aspects we included in this thesis. For example, a testing approach for DSLs might be also quite dependent of the technological space in which the DSLs are specified. For example, the challenges towards testing the operational semantics of a DSL can be different from the ones behind testing denotational semantics. Whereas in the first case, testing corresponds to test imperative programs that describes the execution steps of a given model, in the second case the testing phase corresponds to test declarative rules. Testing declarative programs have different challenges w.r.t testing imperative ones [151].

Part V

Appendixes

Appendix A

Extending EMOF to Support Language Interfaces

In this appendix, we present a mechanism to support the notion of languages interfaces, which is the backbone of approach for modularization of DSLs. This mechanism is based on an extension to EMOF; it introduces the notion of *virtualization* to support the definition of required interface, as well as the notion of *module visibility* to support the definition of provided interface.

A.1 Introducing Virtualization in EMOF

Fig. A.1 shows our extension to EMOF to introduce the notion of virtualization. We create a new meta-class called `VirtualizableElement` which is aimed to identify the specification elements that can be defined as virtual in a language module. We consider the following specification elements as virtualizable: (meta-)classes, properties, operations, parameters, enumerations, and enumeration literals.

Note that this extension requires some well-formedness rules riles to avoid inconsistent metamodels. For example, if an operation is defined as virtual, then all their parameters should be defined as virtual as well. Otherwise, we will face a situation that is conceptually wrong. These well-formedness rules are specified in the form of OCL constraints at the bottom of Fig. A.1, and they are explained in the following.

(1) **All meta-classes containing virtual properties and/or operations must be virtual as well.** The existence of a virtual property and/or operation implies that there is an actual property/operation that implements the desired functionality. However, properties and operations are always defined within a certain meta-class. Hence, there is also a virtual vs. actual relationship between the involved meta-classes. As a result, the meta-class containing a virtual property/operation must be virtual so this relationship can be established.

(2) **Non-primitive types of virtual property must be virtual as well.** Defining a virtual property in a language module `A` implies that there is a language module `B` which implements the desired functionality; thus introducing a dependency from the module `A` to the module `B`. If the type of the virtual property is not primitive (so it is either a meta-class or an enumeration) and it is defined as non-virtual in the module `A`, then this type should be defined as virtual in the module `B`. This implies that there is a second dependency from the module `B` to the module **A** thus creating a dependencies loop. As we will discuss later in this chapter, dependencies loops are not supported in our approach. Then, to be well formed a required interface must guarantee that all the types of virtual references are virtual as well.

(3) **All parameters of a virtual operations must be virtual as well.** Defining a virtual operation in a module `A` implies that there is a language module `B` that implements defines an actual operation with the desired functionality. The only way to implement the same functionality is to respect the signature of the virtual operation. If one of the parameters of the a virtual operation are defined as non-virtual then that the definition of the virtual operation will be incomplete in the required interface and the signatures will not match.

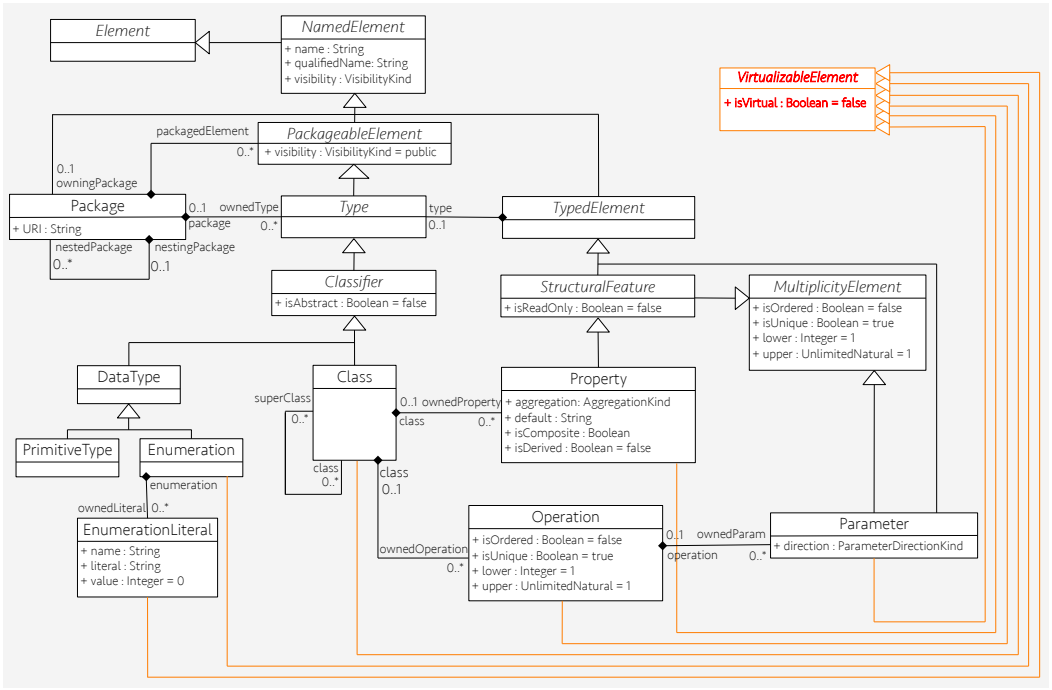
(4) **The owning operation of a virtual parameter must be virtual as well.** The existence of a virtual parameter implies that there is an actual parameter that implements the desired functionality. However, parameters are always defined within a certain operation. Hence, there is also a virtual vs. actual relationship between the involved operations. As a result, the operation containing a virtual parameter must be virtual so this relationship can be established.

(5) **The owning enumeration of a virtual enumeration literal must be virtual as well.** Similarly than in the case of classes and properties, enumeration literals defined as virtual must be always defined in virtual enumerations so the needs they represent can be fulfilled by an actual enumeration in an external language module.

A.2 Introducing Module Visibility in EMOF

Fig. A.2 shows our extension to EMOF to support module visibility. We create a new meta-class called `ModuleVisibilityElement` that allows two levels of visibility (i.e., `public` and `private`) for the specification elements of a language module. We consider the following specification elements as module visibility elements: (meta-)classes, properties, operations, parameters, enumerations, and enumeration literals.

Similarly than in the case of virtualization, we need to establish certain well-formedness rules to guarantee the consistency of a provided interface that is extracted from the public elements within a language module. Those rules are formalized at the bottom of Fig. A.2 and deeply explained in the following.



context Class **inv** : (**self**.ownedProperties.*exists*(p | p.virtual) or **self**.ownedOperation.*exists*(o | o.virtual) **implies self**.virtual
context Property **inv** : **self**.virtual **implies self**.type.virtual
context Operation **inv** : **self**.virtual **implies self**.ownedParam.forAll(p | p.virtual)
context Param **inv** : **self**.virtual **implies self**.operation.virtual
context EnumerationLiteral **inv** : **self**.virtual **implies self**.enumeration.virtual

Figure A.1: Extension to the EMOF to support virtualization

(1) **All meta-classes containing public properties and/or operations must be public as well.** As aforementioned, properties and operations are defined within specific meta-classes. Moreover, the only way to access or call a given property/operation is through its containing meta-class. A public property/operation which is defined in a private meta-class cannot be actually accessed. Hence, a first consistency rule for providing interface is that all the meta-classes containing public properties or operations must be public as well.

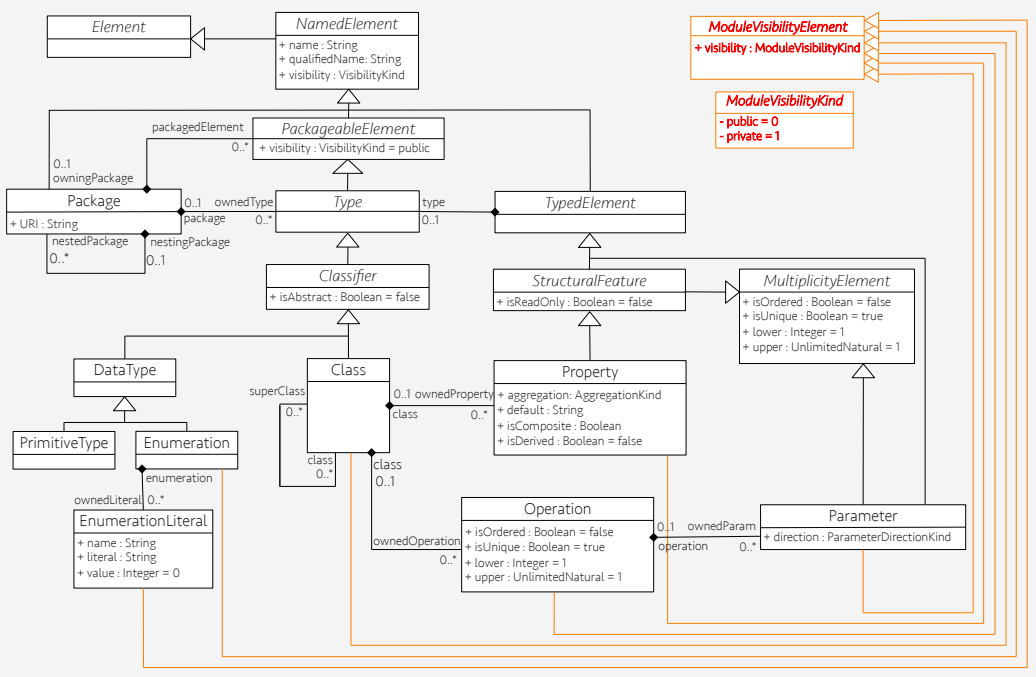
(2) **Non-primitive types of public properties must be public as well.** Defining a property as public implies that the language module under construction provides navigability towards the instances of a given meta-class. If such meta-class is not public, then the navigability cannot be actually performed and we will face an inconsistency.

(3) **All the parameters of public operations must be public as well.** When defining an operation as public, a language designers is exposing its functionality –notably its signature– to external modules. Naturally, to be consistent all the parameters of a

public operation should be public as well so they can be also accessed.

(4) **Public parameters always belong to public operations.** Because parameters can only be accessed through its owning operation, it makes no sense to specify public parameters in private operations. They will never be actually accessed.

(5) **Every enumeration containing at least one public enumeration literal must be public as well.** Because enumeration literals can only be accessed through its owning enumeration, it makes no sense to specify public enumeration literals in private enumerations. They will never be actually accessed. The inverse situation is different. A public enumeration might contain private enumeration literals.



context Class **inv** : (**self**.ownedProperties.*exists*(p | p.public) or **self**.ownedOperation.*exists*(o | o.public) **implies self**.public
context Property **inv** : **self**.public **implies** **self**.type.public
context Operation **inv** : **self**.public **implies self**.ownedParam.forAll(p | p.public)
context Param **inv** : **self**.public **implies** **self**.operation.public
context EnumerationLiteral **inv** : **self**.public **implies** **self**.enumeration.public

Figure A.2: Extension to the EMOF to support module visibility

Appendix B

Hierarchical Domain Analysis

In this appendix, we explain how to use hierarchical domain analysis to identify concept clusters during the design phase of the construction of language product lines. The input of this process is a domain model representing relevant concepts and relationships among them. In turn, the output of this process is a Venn diagram (such as the one introduced in Fig. 5.11) that guides language designers in the definition of concept clusters that will be later encapsulated as language modules.

What is hierarchical domain analysis? Hierarchical domain analysis is a method that allows the extraction of a hierarchy from a given set of elements. Such a hierarchy represents a degree of relatedness between the given elements. The higher level of relatedness the closer relation in the hierarchy. Consider the example introduced in Fig. B.1. In that figure, the fact that A and B are siblings means that they are strongly related. In turn, A and B are also related with C but this relationship is weaker. Similarly, the set composed of A, B, and C is related to D but this relation is even weaker.

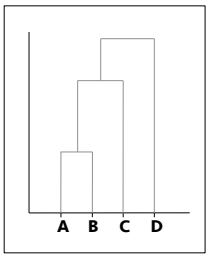
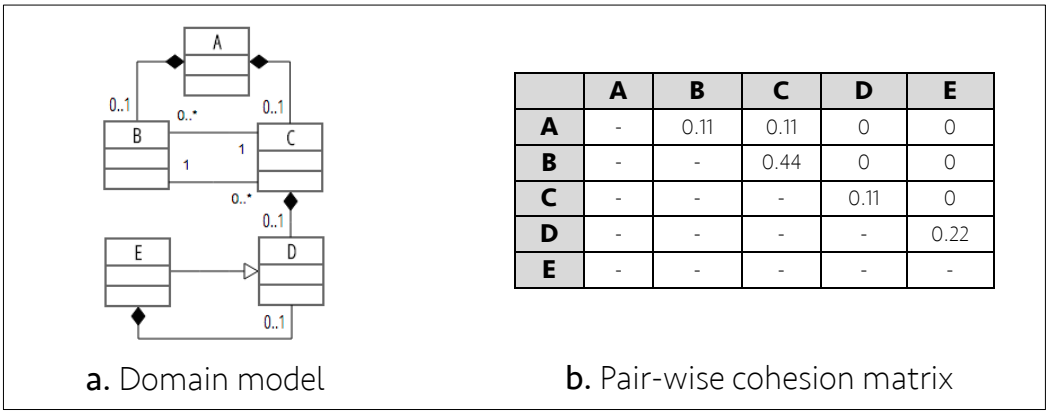


Figure B.1: Example

Using cohesion as a measure of relatedness. Since in our case the main objective of the hierarchical domain analysis is to organize the language concepts of a domain model in a hierarchy that shows the existing sub-domains, we propose to base the analysis in terms of the degree of cohesion among language concepts. The more degree of cohesion between two language concepts, the bigger probability that they are part of the same sub-domain.

But, how to measure cohesion between language concepts? Inspired on the notion of *intra-connectivity* introduced by Mancoridis et al. [132], we propose to measure pair-wise cohesion as the quotient between the amount relationships existing between the language constructs and the total of relationships existing in the domain model. Figure B.2 illustrates this idea through an example. Particularly, Figure B.2a shows a domain model including five concepts: `A`, `B`, `C`, `D`, and `E`. In turn, Figure B.2b shows the corresponding pair-wise cohesion values. Note that the total number of relationships in the domain model is 9 (in-

cluding not only references between classes but also inheritances relationships). Hence, the cohesion between the concepts `B`, and `C` (which have 4 relationships among them corresponding to 2 bidirectional references) is $4/9 = 0.44$. Note that this metric for cohesion provides a number between 0 and 1 being 0 less cohesive than 1.



	A	B	C	D	E
A	-	0.11	0.11	0	0
B	-	-	0.44	0	0
C	-	-	-	0.11	0
D	-	-	-	-	0.22
E	-	-	-	-	-

Figure B.2: An example of pair-wise cohesion in domain models

Once we have defined the cohesion as relatedness metric, we can perform the hierarchical domain analysis. To this end, we use the algorithm introduced by Li et al. [152] on the cohesion matrix. Figure B.3 shows an example of the results for the domain model introduced above including the concepts `A`, `B`, `C`, `D`, and `E`. Note that the result of the analysis is a cohesion tree that shows the hierarchies among language concepts shown in Figure B.3a that, in turn, can be viewed as a Venn diagram as shown in Figure B.3b. In this case, the concepts `B` and `C` are quite cohesive among them so they form a sub-domain. A similar case occurs for the concepts `D` and `E`. Moreover, we can see that `A`, `B`, and `C` form another sub-domain since they are more related each other than with the concepts `D` and `E`.

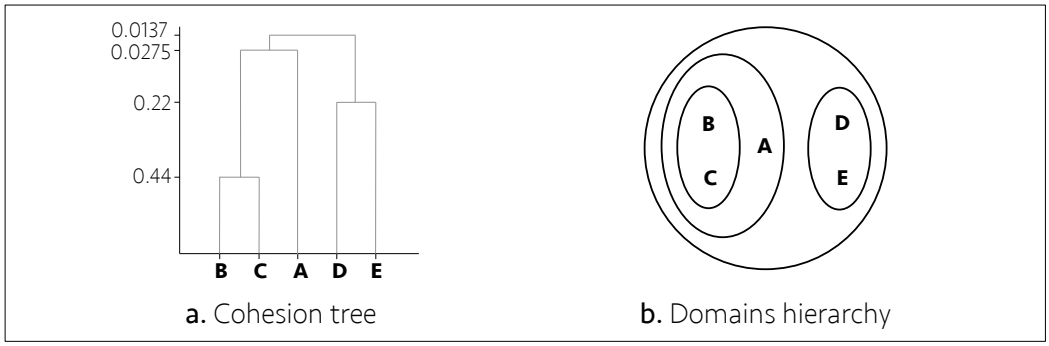


Figure B.3: Example of hierarchical domains

Appendix C

Empirical Data on Specification Cloning in DSLs

Although our experience indicates that the clone-and-own approach is a real practice in language development processes so it is normal to find specification clones, we still need to verify that it is a phenomenon that appears in other development teams, and industrial contexts. To answer that question, we explored public `GitHub` repositories in search of DSLs that are built on the same technological space that we used in our approach. The intention is to confirm the existence of specification clones among those DSLs. The results are presented in this section, and all the data and tooling needed to replicate these experiments are available on-line [1].

Data. We conducted an automatic search on GitHub repositories to find Ecore metamodels enriched with operational semantics written as Kermeta aspects in Xtend. As a result of this search, we obtained a data set composed **2423** metamodels. Nevertheless, because Kermeta 3 and its implementation in Xtend is a quite recent idea, we found very few data for the semantics part. Besides, all of them have been developed in our research team. We decided to conduct the analysis only in the metamodels since we consider that detection of specification clones at the level of the abstract syntax can give us a good insight about the existence of clone-and-own in DSLs development processes.

Experiment. To identify specification clones in the metamodels from our data set, we performed a pair-wise comparison among all the metamodels (w.r.t. the $\div$ operator introduced in section 6.2). Then, we compute the matrix $O(i,j)$ where each cell (i,j) contains the number of cloned metaclasses between the metamodels i and j. $O(i,j) = 0$ means that there is no cloned metaclasses between the metamodels i and j. We are interested in the cells (i,j) such that $O(i,j) \neq 0$ and $i \neq j$. Those cells correspond to a pair of metamodels with some specification clones. Then, we analyze the matrix with two questions in mind: (1) how many metamodels have some specification clones among them?; and (2) how many

[1] Website for experiment 2: http://empiricalpuzzle.weebly.com/

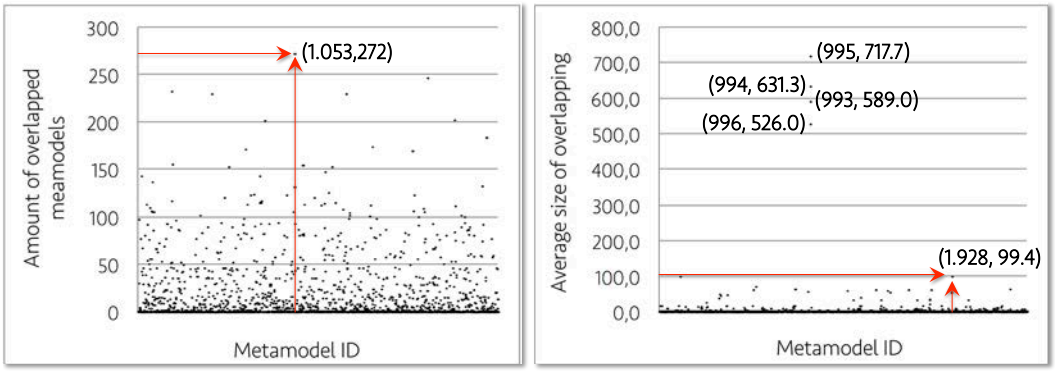


Figure C.1: Results for the evaluation of overlapping in GitHub metamodels

classes are cloned from one metamodel to the other?.

Results. Figure C.1 shows two charts with the results to the experiment. The chart at the left is intended to answer the first question. In this chart, each entry x of the horizontal axis represents one metamodel of the data set. In turn, the vertical axis i.e., $y(x)$ shows the amount of metamodels with some specification clones for x. Formally, $y(x) = (+k|\ 0 \geq k \geq 2423 \wedge O(x,k) > 0 : 1)$. For example, the metamodel with ID **1.053** has some specification clones with **272** metamodels. Note that each point located up the zero line of the vertical axis represents a metamodel with some specification clones with one or more metamodels, thus suggesting that specification clones is a real phenomenon.

The chart at the right of the Figure C.1 is intended to answer the second question. In this chart, each entry x of the vertical axis represents one metamodel of the data set. The vertical axis i.e., $z(x)$ shows the average amount of cloned classes for x. Formally, $z(x) = 1/y(x) * (+k|\ 0 \geq k \geq 2423 : O(x,k))$ For example, the metamodel **1.928** shares, in average, **99.4** metaclasses with other metamodels. Note that there is an important amount of metamodels whose average overlapping size is between **0** and **100** metaclasses. Note also that there are four metamodels that share about **600** metaclasses. This case corresponds to a set of different versions of a metamodel for UML.

List of Tables

List of Figures

List of Publications

Accepted Publications

(1) (**Journal Article**) David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoît Baudry. *Leveraging Software Product Lines Engineering in the Construction of Domain-Specific Languages: A Systematic Literature Review.* In Computer Languages, Systems & Structures. 2016.

(2) (**Conference Paper**) David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, Benoît Baudry, and Gurvan Le Guernic. *Reverse Engineering Reusable Language Modules from Legacy DSLs.* In Proceedings of the 15th International Conference on Software Reuse. Limassol, Cyprus. 2016.

(3) (**Tool Demo Paper**) David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, and Benoît Baudry. *Puzzle: A tool for analyzing and extracting specification clones in DSLs.* In Proceedings of the 15th International Conference on Software Reuse. Limassol, Cyprus. 2016.

(4) (**Keynote Paper**) Jean-Marc Jézéquel, David Méndez-Acuña, Thomas Degueule, Benoît Combemale, and Olivier Barais. *When Systems Engineering Meets Language Engineering.* In Complex Systems Design & Management (CSD&M'14). Paris, France. 2014.

(5) (**Doctoral Symposium Paper**) David Méndez-Acuña. *Variability Management in Domain-Specific Languages*. In Proceedings of the Doctoral Symposium of the 17th International Conference on Model-Driven Engineering Languages and Systems. Valencia, Spain. 2014.

(6) (**Poster**) David Méndez-Acuña, Benoît Combemale, and Benoît Baudry. *Variability Management in Domain Specific Languages.* Presented during the Inter-

national School on Model-Driven Development for Distributed Realtime Embedded Systems (MDD4DRES'14). Aber Whrac'h, France. 2014.

Publications Under Review

(1) (**Journal Article**) David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, and Benoît Baudry. *Reverse Engineering Language Product Lines from Existing DSL Variants.* Submitted to the Journal of Systems and Software. Submission date: 3rd October 2016.

Bibliography

[1] Marsha Chechik, Arie Gurfinkel, Sebastian Uchitel, and Shoham Ben-David. Raising level of abstraction with partial models: A vision. In *Proceedings of Internaional Workshop on Usable Verification*, NSF/MSR 2010. Redmond, Washington, 2010.

[2] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 2005.

[3] Jean-Marc Jézéquel, David Méndez-Acuña, Thomas Degueule, Benoit Combemale, and Olivier Barais. When systems engineering meets software language engineering. In *Proceedings of International Conference on Complex Systems Design & Management*, CSD&M 2014. Springer, Paris, France, 2014.

[4] Stephen Oney, Brad Myers, and Joel Brandt. Constraintjs: Programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*, UIST 2012, pages 229–238, New York, NY, USA, 2012. ACM.

[5] Torsten Lodderstedt, David Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *Proceedings of the International Conference on the Unified Modeling Language*, UML 2002. Springer, London, UK, 2002.

[6] André Ribeiro and Alberto Rodrigues da Silva. Xis-mobile: A dsl for mobile applications. In *Proceedings of the Annual ACM Symposium on Applied Computing*, SAC 2014, New York, NY, USA, 2014. ACM.

[7] M. P. Ward. Language oriented programming. *Software—Concepts and Tools*, 15(1), 1995.

[8] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim R.H. Steel, and Didier Vojtisek. *Engineering Modeling Languages*. Chapman and Hall/CRC, 2016.

[9] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. Dsls: The good, the bad, and the ugly. In *Companion to the ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA Companion 2008, Nashville, TN, USA, 2008. ACM.

[10] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. *Graceful Dialects*. ECOOP 2014. Springer Berlin Heidelberg, Uppsala, Sweden, 2014.

[11] Mathias Funk and Matthias Rauterberg. *PULP Scription: A DSL for Mobile HTML5 Game Applications*. ICEC 2012. Springer, Bremen, Germany, 2012.

[12] Phillip James and Markus Roggenbach. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *Mathematics in Computer Science*, 8(1), 2014.

[13] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. *The SafeCap Platform for Modelling Railway Safety and Capacity*. SAFECOMP 2013. Springer, Toulouse, France, 2013.

[14] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. VML* – a family of languages for variability management in software product lines. In *Proceedings of the International Conference on Software Language Engineering*, SLE 2010. Springer, 2010.

[15] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and picky: Configuration of language product lines. In *Proceedings of the International Conference on Software Product Lines*, SPLC '15, Nashville, Tennessee, 2015. ACM.

[16] João Bosco Ferreira Filho. *Leveraging model-based product lines for systems engineering*. Theses, Université Rennes 1, 2014.

[17] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE Software*, 26(4):47–53, 2009.

[18] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented language families: A case study. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS 2013, Pisa, Italy, 2013. ACM.

[19] Thomas Kühn and Walter Cazzola. Apples and oranges: Comparing top-down and bottom-up language product lines. In *Proceedings of the International Conference on Software Product Lines*, SPLC 2016, Beijing, China, 2016. ACM.

[20] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10), 2004.

[21] Bran Selic. The theory and practice of modeling language design for model-based software engineering—a personal perspective. In *Proceedings of the International Summers School on Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *GTTSE 2011*. Springer, Braga, Portugal, 2011.

[22] Markus Vöelter. Language and ide modularization and composition with MPS. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering IV*, GTTSE 2011. Springer, Braga, Portugal, 2011.

[23] Peter D. Mosses. The varieties of programming language semantics and their uses. In *International Conference on Perspectives of System Informatics*, PSI 2001. Springer, Novosibirsk, Russia, 2001.

[24] Philipp W. Kutter, Daniel Schweizer, and Lothar Thiele. Integrating domain specific language design in the software life cycle. In *Internaional Conference on Applied Formal Methods*, FM 1998. Springer, Berlin, Germany, 1998.

[25] Raphael Mannadiar and Hans Vangheluwe. Debugging in domain-specific modelling. In *International Conference on Software Language Engineering*, SLE 2010. Springer, Eindhoven, Netherlands, 2010.

[26] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[27] Gopal Gupta and Enrico Pontelli. Specification, implementation, and verification of domain specific languages: A logic programming-based approach. In *Computational: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002.

[28] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456 – 479, 2007.

[29] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[30] Benoit Langlois, Consuela Elena Jitia, and Eric Jouenne. Dsl classification. In *Proceedings of the International Workshop on Domain-Specific Modeling*, DSM 2007, 2007.

[31] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[32] Lukas Renggli and Tudor Gîrba. Why smalltalk wins the host languages shootout. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST 2009, Brest, France, 2009. ACM.

[33] Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

[34] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, New York, NY, USA, 2014. ACM.

[35] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag, Secaucus, NJ, USA, 2007.

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

[37] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[38] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[39] R.R. Macala, Jr. Stuckey, L.D., and D.C. Gross. Managing domain-specific, product-line development. *Software, IEEE*, 13(3), 1996.

[40] Walter Cazzola and Davide Poletti. DSL evolution through composition. In *Proceedings of the Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE 2010, Maribor, Slovenia, 2010. ACM.

[41] Tomaz Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71, 2016.

[42] Leandro Marques do Nascimento, Daniel Leite Viana, Paulo Silveira Neto, Dhiego Martins, Vinicius Cardoso Garcia, and Silvio Meira. A systematic mapping study on domain-specific languages. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA 2012. Lisbon, Portugal, 2012.

[43] Ileana Ober, Louis Féraud, and Christian Percebois. Dealing with variability within a family of domain-specific languages: comparative analysis of different techniques. *Innovations in Systems and Software Engineering*, 6(1), 2010.

[44] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software & Systems Modeling*, 14(2), 2013.

[45] Barbara Kitchenham, Rialette Pretorius, David Budgen, O. Pearl Brereton, Mark Turner, Mahmood Niazi, and Stephen Linkman. Systematic literature reviews in software engineering - a tertiary study. *Inf. Softw. Technol.*, 52(8), 2010.

[46] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the International Workshop on Language Descriptions, Tools, and Applications*, LDTA 2012, Tallinn, Estonia, 2012. ACM.

[47] Arne Haber, Markus Look, Antonio Navarro Perez, Pedram Mir Seyed Nazari, Bernhard Rumpe, Steven Volkel, and Andreas Wortmann. Integration of heterogeneous modeling languages via extensible and composable language components. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2015, Angers, France, 2015. Scitepress.

[48] Mads Torgersen. The expression problem revisited. In *Proceedings of the European Conference in Object-Oriented Programming*, ECOOP 2004. Springer, Oslo, Norway, 2004.

[49] Michael Keating. Hierarchical state machines. In *The Simple Art of SoC Design*, pages 47–54. Springer, New York, NY, USA, 2011.

[50] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9), 2013.

[51] Thomas Gschwind. *Automated Adaptation of Component Interfaces with Type Based Adaptation*. Springer, London, UK, 2012.

[52] Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of the European Conference on Object-oriented Programming*, ECOOP 2011, Lancaster, UK, 2011. Springer.

[53] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), 1972.

[54] Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, and Marjan Mernik. Component-based LR parsing. *Computer Languages, Systems & Structures*, 36(1), 2010.

[55] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within modeling language definitions. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS 2009. Springer, Denver, CO, USA, 2009.

[56] Hans Grönniger and Bernhard Rumpe. Modeling language variability. In *Proceedings of the Workshop on Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *Monterey Workshops 2010*. Springer, Redmond, WA, USA, 2010.

[57] Beatriz Mora, Félix García, Francisco Ruiz, and Mario Piattini. Graphical versus textual software measurement modelling: an empirical study. *Software Quality Journal*, 19(1), 2011.

[58] Holger Eichelberger and Klaus Schmid. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 12–21, New York, NY, USA, 2013. ACM.

[59] Michelle L. Crane and Juergen Dingel. UML vs. Classical vs. Rhapsody statecharts: Not all models are created equal. *Software & Systems Modeling*, 6(4), 2007.

[60] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-dimensional variability modeling. In *Proceedings of the International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS 2011, Namur, Belgium, 2011. ACM.

[61] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *International Conference in Software Product Lines*, SPLC 2004. Springer Berlin Heidelberg, Boston, MA, USA, 2004.

[62] Tom Dinkelaker, Martin Monperrus, and Mira Mezini. Supporting variability with late semantic adaptations of domain-specific modeling languages. In *Proceedings of the International Workshop on Composition and Variability*, Composition & Variability 2010, 2010.

[63] Marjan Mernik, Viljem Žumer, Mitja Lenič, and Enis Avdičaušević. Implementation of Multiple Attribute Grammar Inheritance in the Tool LISA. *SIGPLAN*, 34(6), 1999.

[64] Marjan Mernik and Viljem Žumer. Incremental programming language development. *Computer Languages, Systems & Structures*, 31(1), 2005.

[65] Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques, and Maria Jo ao Varanda Pereira. AspectLISA: An Aspect-oriented Compiler Construction System Based on Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 164(2), 2006.

[66] Jaroslav Porubän, Miroslav Sabo, Ján Kollár, and Marjan Mernik. Abstract syntax driven language development: Defining language semantics through aspects. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, FML 2010, Maribor, Slovenia, 2010. ACM.

[67] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A meta-language for modular and reusable development of DSLs. In *Proceedings of the International Conference on Software Language Engineering*, SLE 2015, Pittsburgh, PA, USA, 2015. ACM.

[68] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS 2005. Springer, 2005.

[69] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, 14(2), 2015.

[70] Thomas Cleenewerck. Component-based DSL development. In *International Conference on Generative Programming and Component Engineering*, GPCE 2003. Springer, 2003.

[71] Daniel Ratiu, Markus Völter, Zaur Molotnikov, and Bernhard Schaetz. Implementing modular domain specific languages and analyses. In *Proceedings of the International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa 2012, Innsbruck, Austria, 2012. ACM.

[72] Thomas Cleenewerck and Ivan Kurtev. Separation of Concerns in Translational Semantics for DSLs in Model Engineering. In *Proceedings of the ACM Symposium on Applied Computing*, SAC 2007, Seoul, Korea, 2007. ACM.

[73] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*. Springer, 2010.

[74] Bart Meyers, Antonio Cicchetti, Esther Guerra, and Juan de Lara. Composing textual modelling languages in practice. In *Proceedings of the International Workshop on Multi-Paradigm Modeling*, MPM 2012, Innsbruck, Austria, 2012. ACM.

[75] Iván Melo, Mario Sánchez, and Jorge Villalobos. Composing graphical languages. In *Proceedings of the Workshop on the Globalization of Domain Specific Languages*, GlobalDSL 2013, Montpellier, France, 2013. ACM.

[76] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. In *Human-Computer Interaction. Novel Interaction Methods and Techniques*, volume 5611 of *Lecture Notes in Computer Science*. Springer, 2009.

[77] Luis Pedro, Matteo Risoldi, Didier Buchs, and Vasco Amaral. Developing domain-specific modeling languages by metamodel semantic enrichment and composition: A case study. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2010, Reno, Nevada, 2010. ACM.

[78] Srdan Ẑivković and Dimitris Karagiannis. Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development. In *Enterprise, Business-Process and Information Systems Modeling*, volume 214 of *Lecture Notes in Business Information Processing*. Springer, 2015.

[79] Christian Wende, Nils Thieme, and Steffen Zschaler. A role-based approach towards modular language engineering. In *Proceedings of the International Conference on Software Language Engineering*, SLE 2010. Springer, 2010.

[80] Tom Dinkelaker, Christian Wende, and Henrik Lochmann. Implementing and Composing MDSD-Typical DSLs. Technical Report TUD-CS-2009-0156, Technische Universität Darmstadt, 2009.

[81] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated definition of abstract and concrete syntax for textual languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS 2007. Springer, 2007.

[82] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of the International Conference TOOLS EUROPE*, TOOLS 2008. Springer, 2008.

[83] Walter Cazzola and Ivan Speziale. Sectional domain specific languages. In *Proceedings of the Domain Specific Aspect-Oriented Languages*, DSAL 2009. ACM, 2009.

[84] Walter Cazzola. Domain-specific languages in few steps: The neverlang approach. In *Proceedings of International Conference on Software Composition*. Springer, 2012.

[85] Walter Cazzola and Edoardo Vacchi. Neverlang 2 − componentised language development for the JVM. In *Software Composition*, volume 8088 of *Lecture Notes in Computer Science*. Springer, 2013.

[86] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43, 2015.

[87] Paul Klint. A meta-environment for generating programming environments. *ACM Transations of Software Engineering Methodologies*, 2(2), 1993.

[88] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions Programming Languages Systems*, 24(4), 2002.

[89] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1), 2008.

[90] Jonathan Y. Marchand, Benoit Combemale, and Benoit Baudry. A categorical model of model merging and weaving. In *Proceedings of the International Workshop on Modeling in Software Engineering*, MiSE 2012, Piscataway, NJ, USA, 2012. IEEE Press.

[91] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoit Combemale. Variability support in domain-specific language development. In *International Conference on Software Language Engineering*, SLE 2013. Springer, Indianapolis, IN, USA, 2013.

[92] Edoardo Vacchi, Walter Cazzola, Benoit Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the International Conference on Software Product Lines*, SPLC 2014, Florence, Italie, 2014. ACM.

[93] Brian Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3), 1981.

[94] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*. Springer, 2006.

[95] W. Cazzola and D.M. Olivares. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, PP(99), 2015.

[96] Dimitrios Kolovos, Richard F. Paige, Tim Kelly, and Fiona A. C. Polack. Requirements for domain-specific languages. In *Proceedings of 1st ECOOP Workshop on Domain-Specific Program Development*, DSPD 2006. July 2006.

[97] Leonardo P. Tizzei, Marcelo Dias, Cecília M.F. Rubira, Alessandro Garcia, and Jaejoon Lee. Components meet aspects: Assessing design stability of a software product line. *Information and Software Technology*, 53(2):121 – 136, 2011.

[98] Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado, Márcio Eduardo Delamaro, Edmundo Sérgio Spoto, and W. Eric Wong. *Component-Based Software Quality: Methods and Techniques*, chapter Component-Based Software: An Overview of Testing, pages 99–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[99] José A. Galindo, Hamilton Turner, David Benavides, and Jules White. Testing variability-intensive systems using automated analysis: an application to android. *Software Quality Journal*, pages 1–41, 2014.

[100] Hui Wu, Jeff Gray, and Marjan Mernik. Unit testing for domain-specific languages. In *Proceedings on the 2nd Working Conference on Domain-Specific Languages*, DSL 2009, pages 125–147, Oxford, UK, 2009. Springer Berlin Heidelberg.

[101] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software & Systems Modeling*, pages 1–36, 2015.

[102] Mari Inoki and Yoshiaki Fukazawa. Software product line evolution method based on kaizen approach. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 1207–1214, New York, NY, USA, 2007. ACM.

[103] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6):391–422, November 1999.

[104] Paul Laird and Stephen Barrett. Towards dynamic evolution of domain specific languages. In *Proceedings of 2nd International Conference on Software Language Engineering: Second International Conference*, SLE 2009, pages 144–153. Springer Berlin Heidelberg, Denver, CO, USA, October 2010.

[105] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Planning*, pages 89–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[106] Vander Alves, Pedro Matos, Leonardo Cole, Paulo Borba, and Geber Ramalho. *Extracting and Evolving Mobile Games Product Lines*, pages 70–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[107] Haeng-Kon Kim. *Applying Product Line to the Embedded Systems*, pages 163–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[108] Benoît Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the chasm between executable metamodeling and models of computation. In *Proceedings of the International Conference on Software Language Engineering*, SLE 2012, pages 184–203, Dresden, Germany, 2013. Springer.

[109] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.

[110] Clément Guy, Benoît Combemale, Steven Derrien, Jim R. H. Steel, and Jean-Marc Jézéquel. On model subtyping. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ECMFA 2012, pages 400–415, Lyngby, Denmark, 2012. Springer Berlin Heidelberg.

[111] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.

[112] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA, 1996. IEEE Computer Society.

[113] Fabricia Roos-Frantz, David Benavides, Antonio Ruiz-Cortés, André Heuer, and Kim Lauenroth. Quality-aware analysis in product line engineering with the orthogonal variability model. *Software Quality Journal*, 20(3):519–565, 2012.

[114] Eduardo Santana de Almeida, Jorge Cláudio Cordeiro Pires Mascena, Ana Paula Carvalho Cavalcanti, Alexandre Alvaro, Vinicius Cardoso Garcia, Silvio Romero de Lemos Meira, and Daniel Lucrédio. The domain analysis concept revisited: A practical approach. In *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components*, ICSR'06, pages 43–57, Turin, Italy, 2006. Springer-Verlag.

[115] Cesar Gonzalez-Perez, Tom McBride, and Brian Henderson-Sellers. A metamodel for assessable software development methodologies. *Software Quality Journal*, 13(2):195–214, 2005.

[116] Jerzy Nawrocki, Lukasz Olek, Michal Jasinski, Bartosz Paliświat, Bartosz Walter, Błażej Pietrzak, and Piotr Godek. Balancing agility and discipline with xprince. In *Second International Workshop on Rapid Integration of Software Engineering Techniques*, RISE 2005, pages 266–277, Heraklion, Crete, Greece, 2006. Springer Berlin Heidelberg.

[117] Robert Tairas, Marjan Mernik, and Jeff Gray. Models in software engineering. chapter Using Ontologies in the Domain Analysis of Domain-Specific Languages, pages 332–342. Springer-Verlag, Berlin, Heidelberg, 2009.

[118] Ednaldo Dilorenzo Souza Filho, Ricardo Oliveira Cavalcanti, Danuza F. Neiva, Thiago H. Oliveira, Liana Barachisio Lisboa, Eduardo Santana Almeida, and Silvio Romero Lemos Meira. Evaluating domain design approaches using systematic review. In *Proceedings of the 2Nd European Conference on Software Architecture*, ECSA '08, pages 50–65, Berlin, Heidelberg, 2008. Springer-Verlag.

[119] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[120] Inaya Lahoud, Davy Monticolo, Vincent Hilaire, and Samuel Gomes. A metamodeling and transformation approach for knowledge extraction. In *Proceedings of 4th International Conference on Networked Digital Technologies*, NDT 2012, pages 54–68, Dubai, UAE, 2012. Springer Berlin Heidelberg.

[121] Object Management Group (OMG). Uml 2.4.1 superstructure specification, 2011.

[122] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.

[123] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[124] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[125] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, January 1999.

[126] Daniel Lucrédio, Renata P. M. Fortes, Eduardo S. Almeida, and Silvio Lemos Meira. Performing domain analysis for model-driven software reuse. In *Proceedings of the 10th International Conference on Software Reuse: High Confidence Software Reuse in Large Systems*, ICSR '08, pages 200–211, Berlin, Heidelberg, 2008. Springer-Verlag.

[127] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, Nov 2007.

[128] Jun Jang Jeng and Betty H. C. Cheng. Using formal methods to construct a software component library. In *Proceedings of 4th European Software Engineering Conference*, ESEC '93, pages 397–417, Garmisch-Partenkirchen, Germany, 1993. Springer Berlin Heidelberg.

[129] Ebrahim Bagheri and Dragan Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612, 2011.

[130] Lennart C.L. Kats, Rob Vermaas, and Eelco Visser. Testing domain-specific languages. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 25–26, New York, NY, USA, 2011. ACM.

[131] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting efficient and advanced omniscient debugging for xdsmls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 137–148, New York, NY, USA, 2015. ACM.

[132] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 45–, Washington, DC, USA, 1998. IEEE Computer Society.

[133] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253, Nov 1996.

[134] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Cloned product variants: from ad-hoc to managed software product lines. *International Journal on Software Tools for Technology Transfer*, 17(5):627–646, 2015.

[135] David Méndez-Acuña, José A. Galindo, Benoit Combemale, Arnaud Blouin, and Benoit Baudry. Reverse-engineering reusable language modules from legacy

domain-specific languages. In *Proceedings of the International Conference on Software Reuse*, ICSR 2016. Springer, Limassol, Cyprus, 2016.

[136] Nadia Martaj and Mohand Mokhtari. Stateflow. In *MATLAB R2009, SIMULINK et STATEFLOW pour Ingénieurs, Chercheurs et Étudiants*, pages 513–586. Springer Berlin Heidelberg, 2010.

[137] Roberto E. Lopez-Herrejon, Lukas Linsbauer, José A. Galindo, José A. Parejo, David Benavides, Sergio Segura, and Alexander Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353 – 369, 2015.

[138] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110, 2015.

[139] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon. Automating the extraction of model-based software product lines from model variants (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 396–406, Nov 2015.

[140] Dorel Lucanu and Vlad Rusu. Program equivalence by circular reasoning. In *Proceedings of the International Conference on Integrated Formal Methods*, IFM 2013, pages 362–377, Turku, Finland, 2013. Springer.

[141] Benjamin Biegel and Stephan Diehl. Jccd: A flexible and extensible api for implementing custom code clone detectors. In *Proceedings of the International Conference on Automated Software Engineering*, ASE 2010, pages 167–168, Antwerp, Belgium, 2010. ACM.

[142] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[143] Wesley K.G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. Extracting variability-safe feature models from source code dependencies in system variants. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1303–1310, New York, NY, USA, 2015. ACM.

[144] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wasowski, and Krzysztof Czarnecki. Efficient synthesis of feature models. *Information and Software Technology*,

56(9):1122 – 1143, 2014. Special Sections from “Asia-Pacific Software Engineering Conference (APSEC), 2012” and “ Software Product Line conference (SPLC), 2012”.

[145] Eclipse, xText. Accessed: 2016-08-16. http://www.eclipse.org/Xtext/.

[146] Eclipse, Sirius. Accessed: 2016-08-16. https://eclipse.org/sirius/overview.html.

[147] Christian Berger, Holger Rendel, Bernhard Rumpe, Carsten Busse, Thorsten Jablonski, and Fabian Wolf. Product line metrics for legacy software in practice. In *Workshop Proceedings of the International Software Product Lines Conference*, SPLC 2010, pages 247–250, Jeju Island, South Korea, 2010. Springer.

[148] David Eppstein. *Learning Sequences: An Efficient Data Structure for Learning Spaces*, pages 287–304. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[149] Carla Limongelli, Filippo Sciarrone, Marco Temperini, and Giulia Vaste. *Lecomps5: A Framework for the Automatic Building of Personalized Learning Sequences*, pages 296–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[150] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[151] Sebastian Fischer and Herbert Kuchen. Data-flow testing of declarative programs. *SIGPLAN Not.*, 43(9):201–212, September 2008.

[152] Simon Li. A matrix-based clustering approach for the decomposition of design problems. *Research in Engineering Design*, 22(4):263–278, 2011.

Abstract

The use of domain-specific languages (DSLs) has become a successful technique in the development of complex systems because it furnishes benefits such as abstraction, separation of concerns, and improvement of productivity. Nowadays, we can find a large variety of DSLs providing support in various domains. However, the construction of these languages is an expensive task. Language designers are intended to invest an important amount of time and effort in the definition of formal specifications and tooling for the DSLs that tackle the requirements of their companies.

The construction of DSLs becomes even more challenging in multi-domain companies that provide several products. In this context, DSLs should be often adapted to diverse application scenarios, so language development projects address the construction of several variants of the same DSL. At this point, language designers face the challenge of building all the required variants by reusing, as much as possible, the commonalities existing among them. The objective is to leverage previous engineering efforts to minimize implementation from scratch.

As an alternative to deal with such a challenge, recent research in software language engineering has proposed the use of product line engineering techniques to facilitate the construction of DSL variants. This led the notion of language product lines i.e., software product lines where the products are languages. Similarly to software product lines, language product lines can be built through two different approaches: top-down and bottom-up. In the top-down approach, a language product line is designed and implemented through a domain analysis process. In the bottom-up approach, the language product line is built up from a set of existing DSL variants through reverse-engineering techniques.

In this thesis, we provide support for the construction of language product lines according to the two approaches mentioned before. On one hand, we propose facilities in terms of language modularization and variability management to support the top-down approach. Those facilities are accompanied with methodological insights intended to guide the domain analysis process. On the other hand, we introduce a reverse-engineering technique

to support the bottom-up approach. This technique includes a mechanism to automatically recover a language modular design for the language product line as we as a strategy to synthesize a variability model that can be later used to configure concrete DSL variants.

The ideas presented in this thesis are implemented in a well-engineered language workbench. This implementation facilitates the validation of our contributions in three case studies. The first case study is dedicated to validate our languages modularization approach that, as we will explain later in this document, is the backbone of any approach supporting language product lines. The second and third case studies are intended to validate our contributions on top-down and bottom-up language product lines respectively.