

Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review

David Méndez-Acuña, José Angel Galindo Duarte, Thomas Degueule, Benoit Combemale, Benoit Baudry

▶ **To cite this version:**

David Méndez-Acuña, José Angel Galindo Duarte, Thomas Degueule, Benoit Combemale, Benoit Baudry. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. Computer Languages, Systems and Structures, 2016, 10.1016/j.cl.2016.09.004 . hal-01372702

HAL Id: hal-01372702
https://hal.science/hal-01372702

Submitted on 7 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review

David Méndez-Acuña (✉), José A. Galindo, Thomas Degueule, Benoît Combemale, Benoît Baudry

INRIA/IRISA and University of Rennes 1. France

Abstract

The use of domain-specific languages (DSLs) has become a successful technique in the development of complex systems. Consequently, nowadays we can find a large variety of DSLs for diverse purposes. However, not all these DSLs are completely different; many of them share certain commonalities coming from similar modeling patterns –such as state machines or petri nets– used for several purposes. In this scenario, the challenge for language designers is to take advantage of the commonalities existing among similar DSLs by reusing, as much as possible, formerly defined language constructs. The objective is to leverage previous engineering efforts to minimize implementation from scratch. To this end, recent research in software language engineering proposes the use of product line engineering, thus introducing the notion of *language product lines*. Nowadays, there are several approaches that result useful in the construction of language product lines. In this article, we report on an effort for organizing the literature on language product line engineering. More precisely, we propose a definition for the life-cycle of language product lines, and we use it to analyze the capabilities of current approaches. In addition, we provide a mapping between each approach and the technological space it supports.

Keywords: Software Language Engineering, Domain-Specific Languages, Variability Management, Software Product Lines Engineering

1. Introduction

The increasing complexity of modern software 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 hiding fine-grained implementation details and favoring the participation of domain experts in the software development process [2].

DSLs are software languages whose expressiveness is localized in a well-defined domain, and which provide the abstractions (a.k.a., *language constructs*) intended to describe certain aspect of a system under construction [3]. Naturally, the adoption of such a language-oriented vision relies on the availability of the DSLs needed to describe the diverse aspects of the system [4]; consequently, nowadays there is a large variety of DSLs conceived for diverse purposes [5]. We can find, for example, DSLs to build graphical user interfaces [6], or to specify security policies [7].

Although each of the existing DSLs is unique and has been developed for a precise purpose, not all the existing DSLs are completely different among them. Recent research has shown the existence of DSLs providing similar language constructs [8, 9]. A possible explanation to such phenomenon is the recurrent use of certain modeling patterns that, with proper adaptations, are suitable for several purposes. Consider, for

Email addresses: david.mendez-acuna@inria.fr (David Méndez-Acuña (✉)), jagalindo@inria.fr (José A. Galindo), thomas.degueule@inria.fr (Thomas Degueule), benoit.combemale@inria.fr (Benoît Combemale), benoit.baudry@inria.fr (Benoît Baudry)

instance, the case of finite state machines which have inspired many DSLs dealing with diverse problems such as the design of integrated circuits [10], performing software components composition [11], or the alignment of business processes with legislation [12].

In this context, the challenge for language designers is to take advantage of the commonalities existing among similar DSLs by reusing, as much as possible, formerly defined language constructs [13]. The objective is to leverage previous engineering efforts to minimize implementation from scratch. Ideally, the reuse process should be systematic. This requires, on one hand, to define reusable segments of language specifications that can be included in the definition of several DSLs, and on other hand, an appropriate management of the variability introduced by the particularities of each DSL [2].

To overcome this challenge, the research community in software language engineering has proposed the use of Software Product Lines Engineering (SPLE) in the construction of DSLs [14]. Indeed, the notion of *Language Product Lines Engineering (LPLE)* –i.e., construction of software product lines where the products are languages –has been recently introduced [13, 15]. The main principle behind language product lines is to implement DSLs through *language features*. A language feature encapsulates a set of language constructs that represents certain functionality offered by a DSL [16].

Language features can be combined in different manners to produce tailor-made DSLs targeting the needs of well-defined audiences. This feature-oriented approach for DSLs engineering requires the definition of DSLs in a modularized fashion where language features are implemented as interdependent and composable *language modules*. Additionally, language designers should model the existing variability [17, 18], and provide a configuration mechanism that enable the selection and composition of the language features required in a concrete scenario [19]. The aforementioned challenges constitute the life-cycle of a language product line.

Nowadays, there are several approaches that result useful in the construction of language product lines. Yet, it is difficult to realize what are the most appropriated approach to build a language product line in a particular language engineering project. This difficulty is due to two reasons. Firstly, approaches rarely address the whole life-cycle of language product lines. Rather, many of them focus on a specific issue, and integral solutions are rarely provided. Secondly, approaches are quite dependent on the technological space where DSLs are implemented. For instance, an approach conceived for grammars-based DSLs might be not applicable for metamodels-based DSLs.

This article reports on an effort for organizing the literature on language product line engineering through a systematic literature review. We consider two different perspectives. On one hand, we propose a definition to the life-cycle of language product lines. We use such a life-cycle to analyze the current approaches available in the literature. On the other hand, we establish a mapping between each approach and the technological space it supports. In this sense, this article 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 article a practical guide that they can use to find out the most convenient approach for a particular project according to the technological space used in the implementation of the involved DSLs.

The reminder of this article is structured as follows. Section 2 introduces some background knowledge and states the motivation of this literature review. Section 3 describes the used research method. Section 4 presents the results of this literature review. Section 5 discusses the threats to validity of our study. Finally, Section 6 concludes the article.

2. Background and Motivation

2.1. Basics on Domain-Specific Languages and Software Product Lines Engineering

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

2.1.1. Domain-Specific Languages (DSLs)

In recent years, growing interest in DSLs has led to the proliferation of formalisms, tools, and methods for software language engineering. Hence, numerous techniques for implementing DSLs have emerged. In this section, we shed some light on the most prominent approaches.

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 create 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.

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. 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.

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].

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].

It is worth noting that the different methods for implementing the semantics of software languages are not mutually exclusive. Indeed, some works suggest that one language should own the three types of semantics since each of them provides better support for certain kind of users [27, 26].

Figure 1 sums up the discussed taxonomy in the form of a feature model [28]. Each feature represents a technological space, and the relationships between features represent constraints on the combination of technological spaces. This taxonomy is compatible 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 also complies with the classification of DSLs introduced in [30].

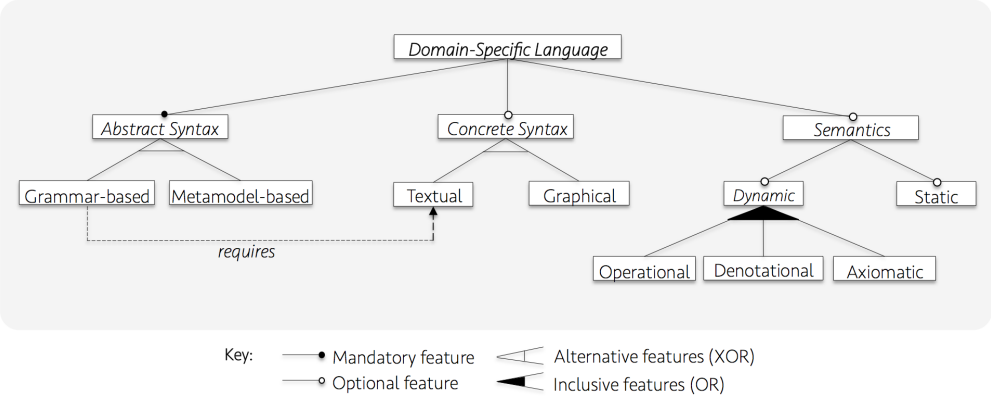


Figure 1: Technological spaces for the implementation of domain-specific languages

External versus internal DSLs. Another important decision when designing a DSL concerns the shape of the resulting language. 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 [5] 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 [5]. 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 are reused, 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. Because of all these reasons, an appropriate selection of the host language is of vital importance [32].

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 and users in, respectively, creating new DSLs and using them. 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.

Most state-of-the-art approaches for software language engineering thus ultimately materialize as features of a language workbench. Therefore, language workbenches occupy a prominent place in this literature review. Similarly, future approaches for language product line engineering should be integrated in language workbenches to promote their adoption by a wide audience. The interested reader can refer to [35] for an in-depth study of the features offered by different popular language workbenches.

2.1.2. 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 [36]. A software product line is an infrastructure that enables to assembly several software products that share some commonalities with well-defined variations [36].

[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].

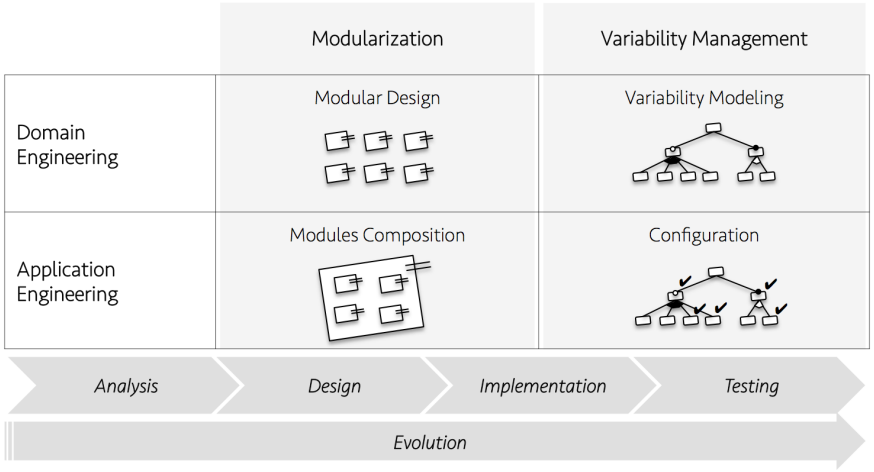


Figure 2: Phases of the SPLE's life cycle

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 shows the life-cycle of a software product line; it is divided into two phases: domain engineering and application engineering [36].

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 [37].

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) [38] became the *"de facto"* standard to express these rules [39].

During the *application engineering* phase, the objective is to derivate software products according to the needs of specific customer segments [36]. 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 phases are intended to be formal software development process. Hence, these phases require the typical steps towards the construction of software: requirements analysis, solution design, implementation, and testing [36]. 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 [40].

2.2. Motivation for a systematic literature review

As aforementioned, there is synergy between the construction of DSLs and software product line engineering [15]. The ideas towards systematic management of software variants provided the product line engineering approach can be used to build similar DSLs while adapting them to specific application contexts [14]. To this end, the life-cycle of software product lines should be adapted to the particularities of DSLs development process. In addition, language workbenches should provide the capabilities that allow

language designers to adapt those ideas in concrete DSLs [2]. Nowadays, we can find several approaches from the software language engineering community that directly or indirectly support this vision. Each approach provides insights and/or tooling 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 DSLs development project. Such 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 the life-cycle. Rather, they are often focused on a particular step (such as modular design) without discussing the other ones. This can be explained by the fact that 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 [41].

The second dimension is the misalignment between the technological space supported by each approach and the technological constraints of a particular DSLs development project. Approaches in software language engineering are quite dependent of 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.

Research questions. The objective of this literature review is to help language designers to find out approaches to leverage software product line engineering in the development of DSLs. To this end, we need to first analyze how the life-cycle of software product line engineering can be adapted to the construction of language product lines. Then, we need to explore the current approaches that result useful in the construction of language product lines, and identify what are the steps of the life-cycle they address as well as the technological space they support. Finally, we need to identify open issues. In summary, this literature review is intended to answer the following research questions:

- **RQ.1:** What is the life-cycle of a language product line?
- **RQ.2:** What are the approaches supporting language product line engineering?
- **RQ.3:** How those approaches support the life-cycle of a language product line?
- **RQ.4:** What are the technological spaces that current approaches support?
- **RQ.5:** What are research open issues in language product line engineering?

Scope. It is worth mentioning that this literature review is restricted to approaches for external DSLs. Conducting a literature review that also includes internal DSLs might be too broad, specially because the development of internal DSLs can resemble to the development of traditional software libraries [42]. Establishing the boundary of the discussion is quite difficult.

Other surveys and literature reviews on software languages engineering. There are other literature studies in the field of software languages engineering. Perhaps the most notable one is presented by Mernik et. al. [5] which provides a comprehensive analysis of the different development phases in the construction of DSLs: analysis, design, and implementation. Besides, the study introduces some 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., [43] 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. [44]. 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

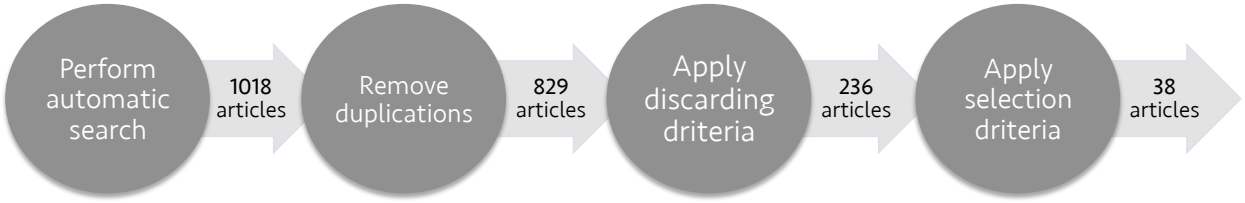


Figure 3: Protocol used to chose the articles included in the discussion.

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 languages engineering is the one presented by Erdweg et al. [45]. 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.

As the reader might notice, all the studies presented so far are intended to provide a general vision on the field of software languages engineering. They analyze a large amount of approaches and 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 article 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 domain-specific languages.

In that sense, our literature review can be compared with other studies addressing localized issues. Some examples of those localized studies are: (1) the work presented by Ober et al. [46] surveys different techniques to deal with interoperability between DSLs; (2) the work presented by Kusel et al. [47] studies approaches to leverage reuse in model-to-model transformations; and (3) the work by Silva et al. [48] focus on describing the elements of model-driven software development which has been used as technological space in the implementation of domain specific languages.

3. Research Method

In this section we provide the details about the research method that we followed during the conduction of this literature review. Concretely, we describe the search protocol that we used to find and select the articles included in the discussion that ends up to the answers for the research questions introduced in section 2.2. The search protocol is illustrated in Fig. 3; it was inspired on the guidelines for systematic literature reviews presented by Kitchenham et al [49].

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 languages engineering. We decided to discard other sources such as *GoogleScholar* that do not guarantee that the indexed documents are validated through peer-reviewing.

That 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 1. Naturally, there might be several variants of these strings. For example, we can consider plurals and acronyms. However, at the end of the searching protocol we performed a validation of the results (that we will explain later in this section) and we concluded that the strings we used are appropriate. This first phase resulted in 1.018 articles.

Table 1: Strings for automatic search

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

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 discussing problems and solutions on software language engineering. 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 language product line engineering i.e., modularity, composition, and variability management.

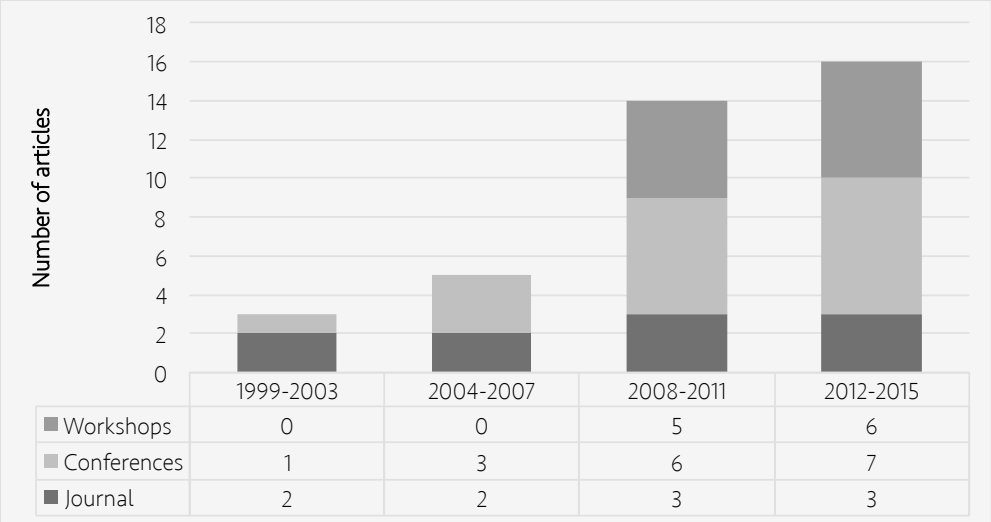
Apply selection criteria. After applying the discarding criteria, we applied a second filter intended to select the articles will be definitely part of the discussion. To this end, we defined a two-fold *selection criteria* that we applied on the article's introductions. This phase resulted in 38 articles.

- Select the articles that have a clear contribution to one or several issues which are relevant on language product line engineering for external DSLs. It is worth highlighting that this filter will exclude works, such as the one presented by Sánchez Cuadrado et al. [50], dealing with issues on language product lines of internal 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. Figure 4 presents the selected articles classified by year and type of publication. Of the 38 articles included in this literature review, 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[2]. It is worth mentioning that a possible threat to validity associated to our research protocol is that all the phases were performed by the same person. This decision favors the uniformity of the results but, at the same time, avoids possible discussions which might enrich the selection process.

Validation of the searching protocol. Despite the rigorous process that we followed in order to identify the articles discussed in this literature review, we wanted to be sure that we considered all the papers that are relevant in the area. In particular, we wanted to be sure that the automatic search does not omit any important article. So, we used three strategies to reduce such a risk. First, before conducting this literature review we established a set of articles that we knew in advance and that are relevant in this study. Afterwards, we checked if those papers were included by the automatic search. The results were positive, all papers in the predefined set were included in the automatic search. Second, we checked the papers cited

[2]Website of the systematic literature review: http://spltosle-survey.weebly.com/



	1999-2003	2004-2007	2008-2011	2012-2015
Workshops	0	0	5	6
Conferences	1	3	6	7
Journal	2	2	3	3

Figure 4: Number of articles per year and type of publication

by the 38 articles finally included in the literature review. We select those 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.

4. Results

In this section, we answer the research questions introduced before. These results were achieved through a systematic process where each paper was read and analyzed according to the vocabulary presented in Section 2.

4.1. ***RQ.1:*** *The life-cycle of a language product line*

The life-cycle of a language product line addresses the same issues addressed by the life-cycle of a software product line (introduced in section 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 many of the articles we selected during the search protocol. We recapitulate these discussions in the rest of this section.

4.1.1. Languages modularization

During the construction of a language product line, language designers should implement DSLs in the form of interdependent *language modules* which materialize language features. Each module provides a set of language constructs, and a DSL is obtained from the composition of two or more language modules.

Just as in components-based software development [51], languages modularization supposes the existence of two properties: separability and composability. *Separability* refers to the capability of designing and verifying language modules independently of other language modules it may require. Separability relies on the definition of interfaces specifying the interactions between language modules. In turn, *composability* refers to the capability of integrate several language modules to produce a complete and functional DSL. Composability relies on the usage of the interfaces between language modules in such a way that they can interchange both control and information.

Table 2: Modularization scenarios in the literature

This article	**Erdweg et al.**	**Vöelter et al.**	Haber et al.
	[45]	[22]	[52]
Extension	Extension	Extension	Inheritance
Restriction	Restriction	Restriction	-
Aggregation	-	Combination	Embedding
Unification	Unification	-	-
Self-extension (-)	Self-extension	Embedding	-
Referencing	-	Referencing	Aggregation
Extension composition	Extension composition	-	-

Modular languages design to reach separability. To achieve a modular language design that effectively reaches separability, language designers must (1) decide how to group language constructs into different language modules, and (2) establish the dependencies among them. The first task corresponds to a design process which, ideally, consider classical good practices such as high-cohesion and low-coupling. The second task corresponds to the definition of the interfaces among those language modules.

These design dimensions have been discussed in the literature in the form of modularization scenarios [45, 22, 52]. A modularization scenario describes a situation where two language modules interact each other according to the nature of the dependencies existing among their constructs. Those scenarios are explained in the following. As the reader will notice, the modularization scenarios have named differently along the literature. A unified vocabulary is presented in Table 2.

The modularization scenario called self-extension will not be discussed in this literature review because it is only applicable to the case of internal DSLs.

- **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 [53].

  For instance, a language for expressing finite state machines can be extended to support hierarchical state machines by introducing the notion of composite state [54]. 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 [55] 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 [45, 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

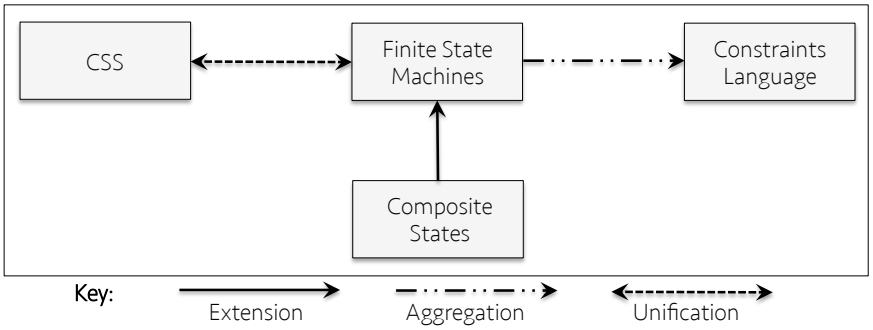


Figure 5: Example of extension composition

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.

It is worth mentioning that, 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 [6] 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 in the literature as **extension composition**. Consider for example the case presented in Fig. 5 where the web styling language CSS is unified with a state machines language that, in turn, uses the functionality of a constraints language (i.e., aggregation) and that is extended by the notion of composite states. It can be viewed as a sort of algebra that allows to structure relationships composition among several language modules in terms of the scenarios presented so far.

Language modules composition to reach composability. 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 [45, 55] or at the level of the tooling [56]. 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".

It is worth mentioning that language modules composition requires a previous phase of *compatibility checking*. In this context, compatibility checking corresponds to verify that the interface between two language modules is consistent according to the modularization scenario. In the case of extension, the interface is consistent if the elements used in the extension module correspond to elements defined in the extension points of the base module. Similarly, in restriction we need to check that the elements that are being restricted in the restriction module corresponds to elements that actually exist in the restriction point of the base module. For the case of aggregation and referencing, compatibility checking is performed by verifying that the requirements of the requiring language module are correctly fulfilled by the constructs offered in the providing language module. Finally, in unification we need to check compatibility between the glue code with respect to the left and right modules.

The substitutability principle. Substitutability is an important property in components-based software development. It states that software modules should be easily interchangeable with other software modules providing the same functionality. Substitutability has been largely discussed in object oriented programming [57], and it is also relevant in modular language design [2]. Indeed, when language designers are facing the problem of variability, substitutability becomes not only relevant issue, but also necessary.

Being transversal to modular languages design, and to language modules composition, substitutability is not easy to address. Language modules should be as less coupled as possible, and properties such as polymorphism are required. We will discuss later how current approaches address this challenge.

4.1.2. Languages variability management

Language variability management is the second component of the language product line's life cycle. The specificities of the application contexts where a DSL can be applied suppose adaptations should be systematically managed. In the case of DSLs, the variability management process should take into account the three dimensional nature of the specification of a DSL. In this section, we discuss the impact that this fact has on the variability management of a language product line.

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 [17, 18]. Let us summarize each of these dimensions.

- **Abstract syntax variability (or functional 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"* [13].

  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* in order to facilitate the selection. Such grouping is motivated by the fact that, usually, selecting constructs can be difficult because a DSL usually has many constructs, so a coarser level of granularity is required.

- **Concrete syntax variability (or representation variability)** Depending on the context and 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 [58] 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 [59]. Representation variability refers to the capability of supporting different representations for the same language construct.

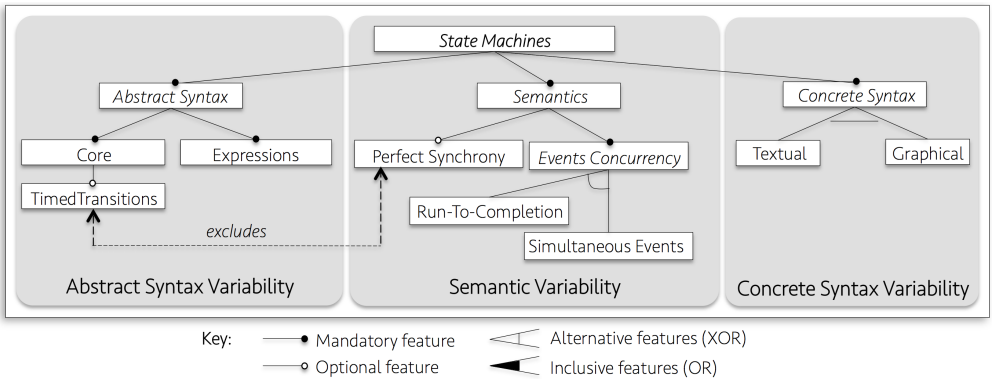


Figure 6: Multi-dimensional variability for a state machine language

- **Semantic variability:** Another problem that has gained attention in the literature of software languages 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 [60]. 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 construct.

These dimensions of variability are not mutually exclusive. 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 [61] is required; it should take into account the fact that decisions taken in the resolution of the functional variability may affect decisions taken in the representation and semantics variability.

Figure 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 functional 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 representation variability dimension presents the choice between graphical or textual DSLs for state machines.

Multi-staged languages 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 functional 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. [62] for the general case of software product lines, and discussed by Dinkelaker et al. [63] 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 [63]. 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 7. In that scenario,

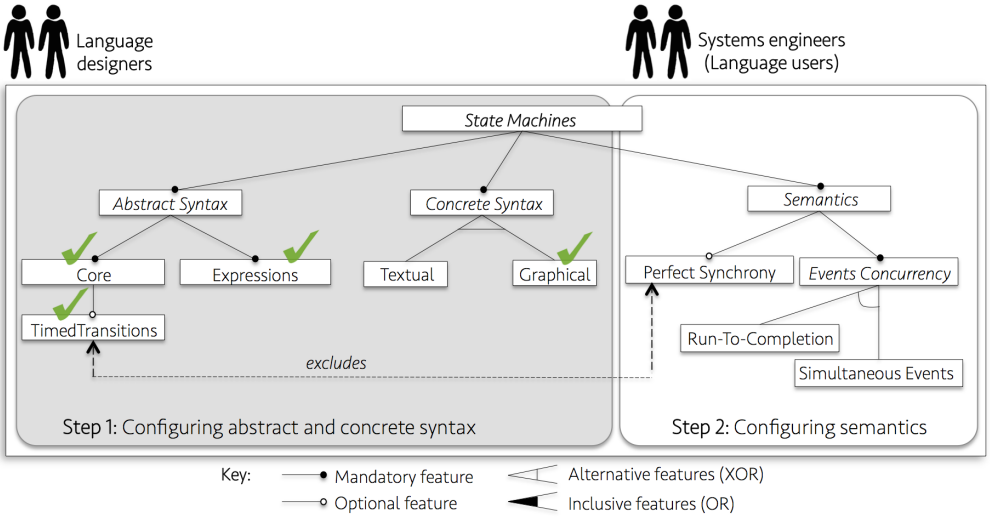


Figure 7: Example for multi-staged configuration of a language product line

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* [63]. 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.

4.1.3. Top-down vs. bottom-up language product lines

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 [64].

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.

*4.2. **RQ.2:** Approaches supporting language product line engineering*

After reading the articles obtained from the search 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.

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.

Table 3: Approaches that support language product line engineering

Name	Articles
LISA	[65, 66, 67, 68, 55]
Melange	[69, 70, 71]
Keywords-based modularization	[72]
Meta Programming System, MPS	[73, 22]
Modularization on top of ATL	[74]
Modularization on top of MetaDepth	[75, 76]
Gromp	[77]
Domain-concepts based modularization	[78, 79]
Interfaces-based modularization	[80]
Components-based LR parsing	[56]
Roles-based modularization	[81, 82]
MontiCore	[83, 84, 31, 17, 18]
Neverlang+AiDE	[85, 41, 86, 87, 88]
ASF+SDF+FeatureHouse	[89, 90, 91, 16]

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

*4.3. **RQ.3:** Current support for the language product lines' life-cycle*

In this section, we analyze how the aforementioned approaches support each step of the language product line's life-cycle.

4.3.1. Support for languages modularization

Languages modularization has been largely discussed in the literature. Indeed, the most part of the approaches that we discuss in this article aim to support languages modularization. In the following we discuss current advances in modular languages design and language modules composition.

Support for modular languages design. 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

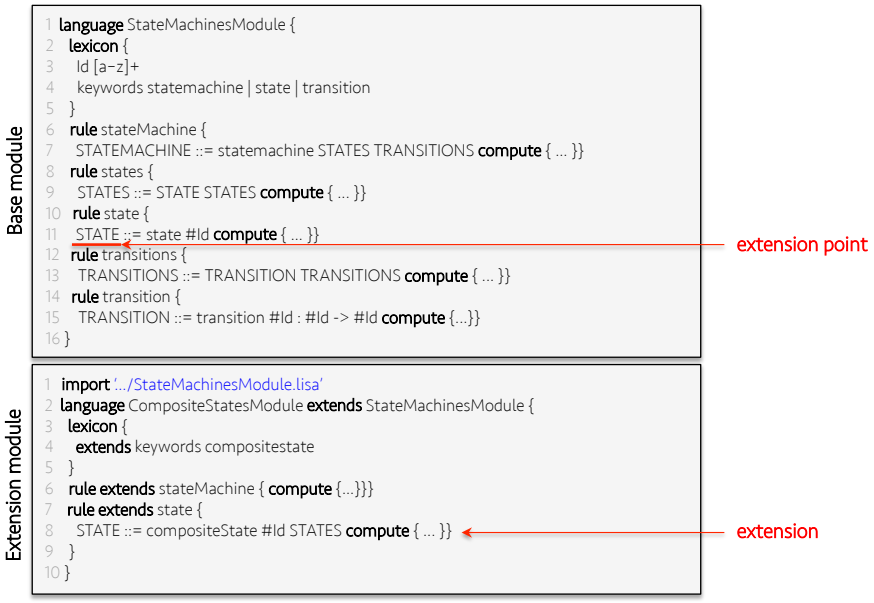


Figure 8: Exogenous extension in LISA

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 interfaces that support the modularization scenarios (introduced in section 4.1.1) are addressed. Second, they constraint the type of composition strategy used for the composition of language modules. In the reminder of this section, we will discuss how the modularization scenarios are addressed through the modularization techniques. Afterwards, we analyze the composition strategies required in each case. We 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. In this case, 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.

  LISA is one of the approaches that use endogenous modularization to support extension/restriction of language modules. In LISA, language designers define language modules thought attribute grammars that can be imported and extended by extension modules. Extension modules can introduce new production rules and/or overriding existing ones.

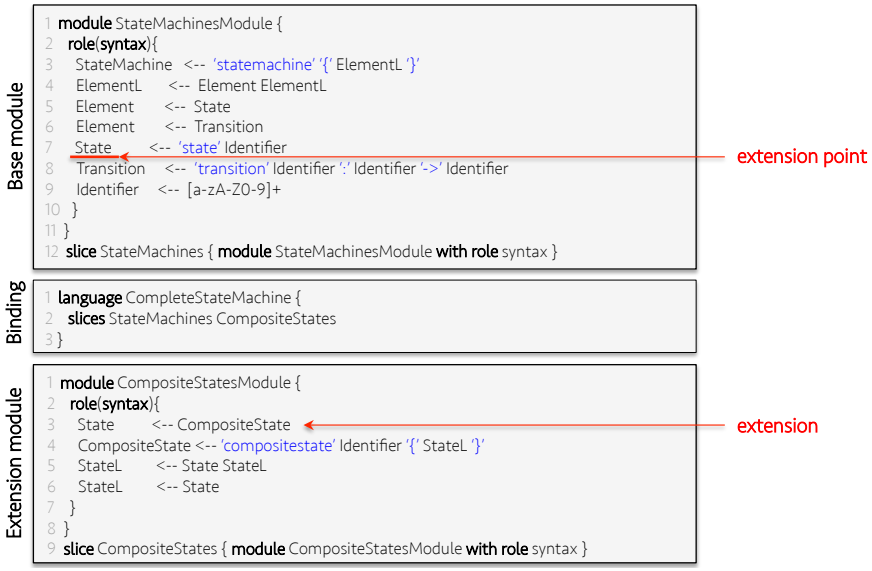


Figure 9: Exogenous extension of language modules in Neverlang

Consider for example the base module defined on top of Figure 8 which defines a simple DSL for state machines. It contains production rules for the concepts StateMachine, State, and Transition. In turn, the extension module presented on the bottom of Figure 8 imports the base module and introduces the concept CompositeState through a new production rule for the non-terminal State. Once the composition of the modules is performed, the non-terminal State can take the form of either a simple state or a composite state.

- **Exogenous modularity to support extension and restriction.** Also, 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.

  Neverlang is one of the approaches that use exogenous modularity to support extension of language modules. In Neverlang, a base module is specified in a BNF-like grammars that can be enhanced by extension modules introducing new production rules. Figure 9 illustrates this capability with the example of state machines and composite states. The base module is a simple DSL for state machines and the extension module introduces a new production rule for the construct State, thus introduction the notion of composite state. Note that there is a composition script (in the middle of the Figure 9) that indicates that a language called CompleteStateMachine is the result of the composition of the language modules called StateMachines (base module) and CompositeStates (extension module).

- **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.

  Figure 10 illustrates endogenous aggregation of language modules in LISA. In the top of the figure, there is a language module that defines a DSL for state machines which requires boolean expressions



Figure 10: Endogenous aggregation of language modules in LISA

to express guards in the transitions. Such boolean expressions are provided by the language module presented at the bottom of the figure, and which is imported by the state machines DSL. In this case, we use the import clause not for extend some language constructs, but to use those language constructs in the definition of new ones. Hence, the language constructs used by the requiring language are used at the left of the production rules.

- **Exogenous modularity to support aggregation.** Aggregation of language modules 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.

  Consider the example introduced in Figure 11 that uses Neverlang in the same example of the state machines language and expressions. In that case, the production rule for the construct transition uses the non-terminal Expression. Since this non-terminal is not implemented in the language module, it represents a declaration that is intended to be fulfilled by another language module. At the bottom of the figure, we introduce an expression language which implements the required language constructs.

  At a first view, 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



Figure 11: Exogenous aggregation of language modules in Neverlang

modularity, the integration module directly imports the left/right modules and uses (or extends) their language constructs to define a unified language.

Consider the example introduced in Figure 12. In that case, the idea is to unify a DSL for state machines with a CSS to provide an approach to define the style of web pages. In particular, the idea is to define state machines with associated style sheets. To this end, there are a glue module that imports both the module containing the state machines language and the module containing the CSS language. Such a glue module extends the construct State to add a new property corresponding to the style sheet. Note that this idea can be generalized to support more complex modularization scenarios. Hence, extension composition can also be supported.

- **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.

  Consider the example presented in the Figure 13 regarding the case of state machines for styling web pages. In this case, the approach for unification is the same that in endogenous modularity with the difference that the binding between language modules is specified externally by a composition artifact. These ideas can be generalized to support extension composition.

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 [56].

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., [56] introduces a



Figure 12: Example of endogenous unification in LISA

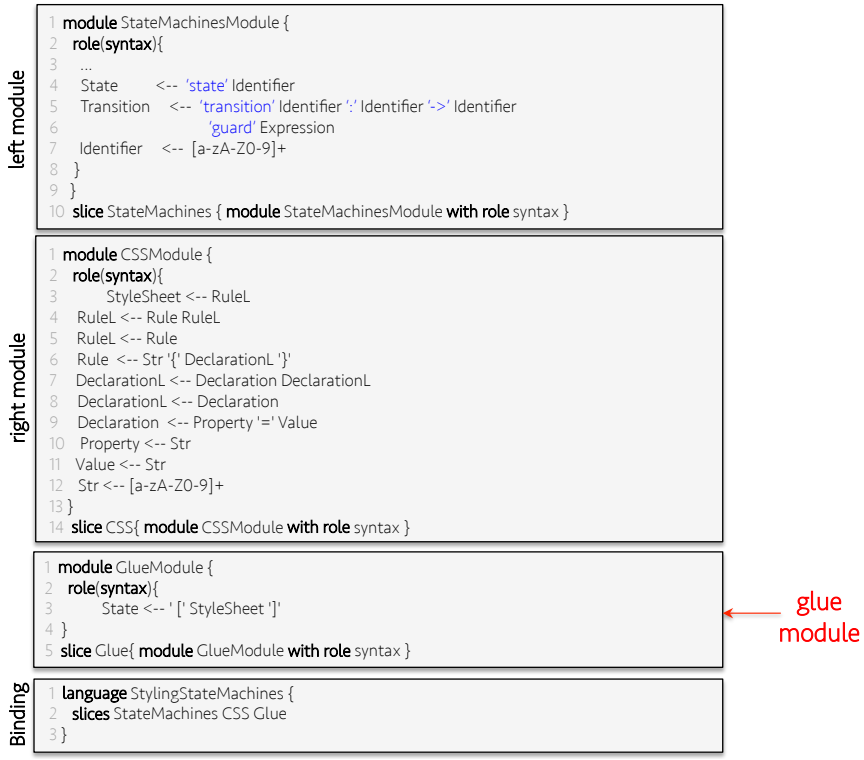


Figure 13: Exogenous unification in Neverlang

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; we briefly explains those techniques in the following.

- **Direct linking to compose endogenous modules.** In endogenous modularization, implementation artifacts are physically related via direct linking usually realized through the `import` clause. In direct linking, all the content of the referenced artifact is included at the beginning of the referencing one. The involved artifacts should be viewed as a unique specification.

- **Artifacts merging to compose exogenous modules.** In exogenous modularization, implementation artifacts are completely independent so there are not 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. 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 4.1.1. This is because the importing language module can access all the language constructs of the imported one, thus enabling 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 literature review.

- **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 [55]. 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 4.1.1. 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"* [92]. In language modules composition, merging is an additive operator that sums the language 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 4.1.1. 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 [92]. 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.

Current approaches dealing with languages modularization. Fig. 4 shows how current approaches support languages modularization. In particular the figure presents table which, for each approach, indicates: (1) the modularization scenarios it supports, (2) the modularization techniques it uses, (3) whether the composition is performed at the level of the specification or tooling, and (4) the composition operator it provides (if any).

The most remarkable conclusion of this figure is that endogenous and exogenous modularization are not mutually exclusive. Indeed, Monticore uses endogenous modularization in junction with the inheritance operator to support language modules extension and restriction. In turn, Monticore it uses exogenous modularization and merge to support language modules aggregation.

4.3.2. Support for languages variability management

In contrast to the large amount of articles on languages modularization, we found 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, and vary with respect to the modeling approach they use to represent the variability. In particular we found three different modeling approaches.

- **Feature models supporting functional variability.** Wende et al. [81] use feature models as a documentation artifact to present a catalog of language features that can be combined each other to produce different variants of a DSL. The modeling approach that they propose is illustrated in Figure 14. In that approach, each feature is associated to a language module that is completely specified in terms of abstract syntax, concrete syntax, and semantics.

  This approach is quite useful in language product lines with functional 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

	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 scenarios														
Extension	●	●	-	●	-	●	●	●	-	●	-	●	●	●
Restriction	●	●	-	●	-	●	●	●	-	●	-	●	●	●
Aggregation	●	●	●	●	●	●	-	●	●	●	●	●	●	●
Unification	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Extension Composition	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Modularization technique														
Endogenous modularization	●	-	-	●	●	●	-	●	-	●	-	●	-	-
Exogenous modularization	-	●	●	-	-	-	●	-	●	-	●	●	●	●
Composition strategy														
Specification composition	●	●	●	●	●	●	●	●	●	-	●	●	●	●
Tooling composition	-	-	-	-	-	-	-	-	-	●	-	-	-	-
Composition operator														
Inheritance	●	-	-	-	-	-	-	-	-	-	-	●	-	-
Merge	-	●	●	-	-	-	●	-	●	-	●	●	●	-
Superimposition	-	-	-	-	-	-	-	-	-	-	-	-	-	●
Weaving	-	●	-	-	-	-	-	-	-	-	-	-	-	-

Table 4: Mapping current approaches with language modularization

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.

- **Multi-dimensional variability with concern-specific features.** The approach presented in [93] 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 15 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 [17]. In other words, this approach supports not only abstract syntax variability, but also concrete syntax and semantic variability. To support multi-dimensional 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 16 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

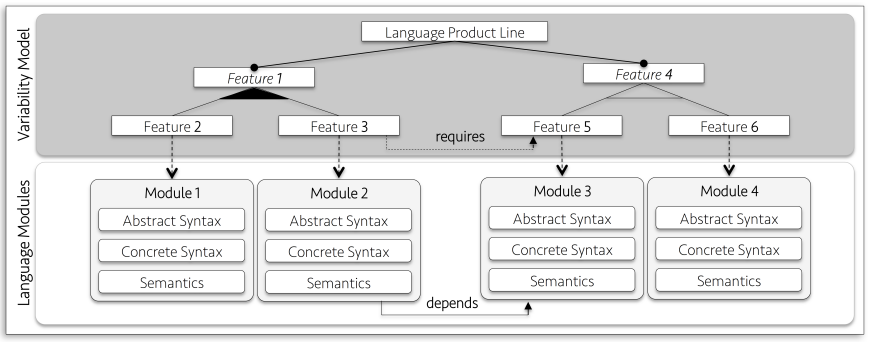


Figure 14: Boolean feature models for representing functional variability

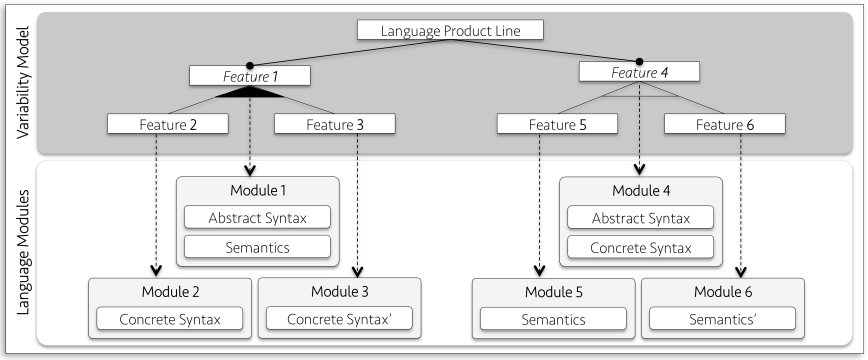


Figure 15: Boolean feature models for representing syntactic and semantic variability

implements the corresponding functionality in the corresponding implementation concern. 16.

The approach presented in [16] proposes the use of FeatureHouse as a languages variability management framework on top of ASD+SDF. It supports not only functional variability but also syntactic and semantic variability. The variability modeling approach is quite similar to the one used in MontiCore and illustrated in Figure 16. This variability management mechanism provides facilities to configure and derivate languages from a given feature model.

Multi-dimensional variability requires a modularization approach that supports the definition of each implementation concern (i.e., abstract syntax, concrete syntax, and semantics) in a different language module. In the cases where such a 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 variability models from a given set of language modules. The first approach (presented in [94]) 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 a domain ontology that describes the main concepts of the domain of the language product line. The second approach (presented in [15]) refines the former by removing the ontology as an input. 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.

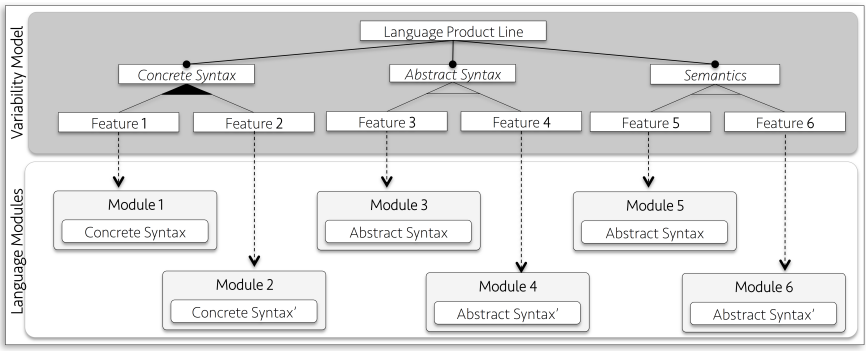


Figure 16: Boolean feature models for representing multi-dimensional variability

Support for multi-staged languages 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. As a result, once the variability of a language product line is modeled in a tool-supported feature model, language designers already 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 [17] 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 particularly relevant when feature models become bigger.

Current approaches dealing with languages variability management. Figure 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 5: Current approaches dealing with languages variability management

4.4. ***RQ.4:*** *Mapping current approaches with technological spaces*

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

- **LISA.** LISA supports the construction of textual DSLs where the abstract syntax, concrete syntax, and semantics are specified through attribute grammars[3]. 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 [70, 71]. 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. This approach supports modular languages design and language modules composition. More concretely, it introduces modularization on top of the ATL transformation language [96].

- **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

[3]Semantics defined through attribute grammars is associated to a form of denotational semantics [95].

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 real world case study that is presented in [79].

- **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 [82]. 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 are defined denotationally in a theorem prover. 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 [17].

- **Neverlang+AiDE.** Neverlang+AiDE 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 [89, 90, 91]. In turn, the approach presented in [16] proposes the use of FeatureHouse as a languages variability management framework on top of ASD+SDF.

As a summary of the discussion presented in this section, we introduce the mapping that relates each approach with the DSLs technological spaces it supports. The results are summarized in Table 6.

4.5. ***RQ.5:*** *Open issues and research roadmap*

During the conduction of this literature review, we found an important amount of approaches to support the implementation of language product lines. We provide evidence enough to state that, with the current tool support, 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 some approaches to automatically generate a first version of those feature models.

	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 6: Mapping between current approaches and technological spaces for the implementation of DSLs

Despite all these advances, there is a long path to follow. In particular, the methodological aspects of the construction of a language product line are rarely mentioned and never studied properly. 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 question that should be addressed to facilitate the construction of language product lines.

4.5.1. ***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 [36]. 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 [97]. They identify common and variability 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. [98] 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.

4.5.2. ***Design*** *in language product line engineering*

In the development of software product lines, the design phase is dedicated to the definition of a *reference architecture* 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 [36]. To address this activity, software engineers often use design patterns and modularization properties (such as low coupling and high cohesion) [99] to design appropriate reference architectures 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 design of a reference architecture 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?

4.5.3. ***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 [36]. 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 [100]. 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 [101].

All the issues mentioned above are still open in language product lines; they are never discussed in the literature discussed in this article. 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 [102], or [103]) 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 [104].

4.5.4. ***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 [105].

Evolution in software product lines supposes several challenges and depending on the nature of the evolution in the requirements the infrastructure might change differently [106]. 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 [107]. 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?

5. Threats to Validity

In this section, we discuss the possible threats to validity of our study. Concretely, we discuss three of the different types of validity proposed by Wohlin et al. [108]: 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). Does our study include all the relevant articles existing in the literature on language product line engineering?

In order 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; 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; 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 articles we 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 2). 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 [109] or *Embedded Systems* Product Lines [110].

The study provided in this article is based on an abstraction of the generalities of Software Product Lines Engineering. In section 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.

6. Conclusion

This article reports on an effort for organizing the literature on the applicability of software product line engineering techniques un the construction of domain-specific languages i.e., language product line engineering (LPLE). Being the intersection of two different bodies of knowledge of software engineering, LPLE is motivated by the need of building DSLs that, sharing some commonalities with existing DSLs, have their proper particularities emerging from their specific application contexts.

The main contribution of this article is a detailed study that shows how the works existing today in the literature adapt the life-cycle of a software product line where the software products resulting of configuration processes are DSLs. To this end, we conducted a systematic literature review which includes a large amount of articles, and then we present the results structured in such a way that they are useful for researchers and practitioners. From the point of view of the researchers, this article provides a conceptual framework to understand the challenges behind the construction of language product lines as well as a roadmap envisioning future research directions. From the point of view of the practitioners, this article presents a catalog of approaches that they can use to select the best approach according to the needs of a particular development project involving language product lines.

Acknowledgments

This work is supported by the ANR INS Project GEMOC (ANR-12-INSE-0011); the bilateral collaboration VaryMDE between Inria and Thales; the bilateral collaboration FPML between Inria and DGA; and the European Union within the FP7 Marie Curie Initial Training Network RELATE" under grant agreement number 264840. We thank the anonymous reviewers for their insightful comments, which helped us improve the manuscript.

References

[1] M. Chechik, A. Gurfinkel, S. Uchitel, S. Ben-David, Raising level of abstraction with partial models: A vision, in: Proceedings of NSF/MSR Workshop on Usable Verification, ICMT 2012, Redmond, Washington, 2010.

[2] J.-M. Jézéquel, D. Méndez-Acuña, T. Degueule, B. Combemale, O. Barais, When systems engineering meets software language engineering, in: F. Boulanger, D. Krob, G. Morel, J.-C. Roussel (Eds.), Complex Systems Design & Management, Springer International Publishing, 2015, pp. 1–13. doi:10.1007/978-3-319-11617-4_1.

[3] B. Combemale, J. De antoni, B. Baudry, R. France, J.-M. Jézéquel, J. Gray, Globalizing modeling languages, Computer 47 (6) (2014) 68–71. doi:10.1109/MC.2014.147.

[4] T. Clark, B. S. Barn, Domain Engineering: Product Lines, Languages, and Conceptual Models, Springer, Berlin, Heidelberg, 2013, Ch. Domain Engineering for Software Tools, pp. 187–209.

[5] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.

[6] S. Oney, B. Myers, J. Brandt, ConstraintJS: Programming interactive behaviors for the web by integrating constraints and states, in: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, ACM, New York, NY, USA, 2012, pp. 229–238. doi:10.1145/2380116.2380146.

[7] T. Lodderstedt, D. Basin, J. Doser, SecureUML: A UML-Based modeling language for model-driven security, in: J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.), UML 2002 - The Unified Modeling Language, Vol. 2460 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 426–441. doi:10.1007/3-540-45800-X_33.

[8] S. Zschaler, D. S. Kolovos, N. Drivalos, R. F. Paige, A. Rashid, Domain-specific metamodelling languages for software language engineering, in: Software Language Engineering, Vol. 5969 of LNCS, Springer Berlin Heidelberg, 2010, pp. 334–353. doi:10.1007/978-3-642-12107-4_23.

[9] J. Lara, E. Guerra, Domain-specific textual meta-modelling languages for model driven engineering, in: 8th European Conference on Modelling Foundations and Applications, ECMFA 2012, Springer Berlin Heidelberg, Lyngby, Denmark, 2012, pp. 259–274.

[10] J. Agron, Domain-specific languages, in: W. M. Taha (Ed.), Domain-Specific Language for HW/SW Co-design for FPGAs, DSL 2009, Springer Berlin Heidelberg, Oxford, United Kingdom, 2009, pp. 262–284. doi:10.1007/978-3-642-03034-5_13.

[11] M. Anlauff, P. W. Kutter, A. Pierantonio, A. Sünbül, Using domain-specific languages for the realization of component composition, in: T. Maibaum (Ed.), Fundamental Approaches to Software Engineering, FASE 2000, Springer Berlin Heidelberg, Berlin, Germany, 2000, pp. 112–126. doi:10.1007/3-540-46428-X_9.

[12] S. Nešković, O. Paunović, S. Babarogić, Using protocols and domain specific languages to achieve compliance of administrative processes with legislation, in: K. N. Andersen, E. Francesconi, Å. Grönlund, T. M. Engers (Eds.), International Conference on Electronic Government and the Information Systems Perspective, EGOVIS 2011, Springer Berlin Heidelberg, Toulouse, France, 2011, pp. 284–298. doi:10.1007/978-3-642-22961-9_23.

[13] S. Zschaler, P. Sánchez, J. a. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. a. Araújo, U. Kulesza, VML* a family of languages for variability management in software product lines, in: M. van den Brand, D. Gasevic, J. Gray (Eds.), Software Language Engineering, Vol. 5969 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 82–102. doi:10.1007/978-3-642-12107-4_7.

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

[15] T. Kühn, W. Cazzola, D. M. Olivares, Choosy and picky: Configuration of language product lines, in: Proceedings of the 19th International Conference on Software Product Line, SPLC '15, ACM, New York, NY, USA, 2015, pp. 71–80. doi:10.1145/2791060.2791092.

[16] J. Liebig, R. Daniel, S. Apel, Feature-oriented language families: A case study, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, ACM, New York, NY, USA, 2013, pp. 11:1–11:8. doi:10.1145/2430502.2430518.

[17] M. V. Cengarle, H. Grönniger, B. Rumpe, Variability within modeling language definitions, in: A. Schürr, B. Selic (Eds.), Model Driven Engineering Languages and Systems, Vol. 5795 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 670–684. doi:10.1007/978-3-642-04425-0_54.

[18] H. Grönniger, B. Rumpe, Modeling language variability, in: R. Calinescu, E. Jackson (Eds.), Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems, Vol. 6662 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 17–32. doi:10.1007/978-3-642-21292-5_2.

[19] D. Méndez-Acuña, Variability management in domain-specific languages, in: B. Baudry (Ed.), Proceedings of the Doctoral Symposium at MODELS'14, Vol. Vol-1321, CEUR, Valence, Spain, 2014.

[20] D. Harel, B. Rumpe, Meaningful modeling: what's the semantics of "semantics"?, Computer 37 (10) (2004) 64–72. doi:10.1109/MC.2004.172.

[21] B. Selic, The theory and practice of modeling language design for model-based software engineering: A personal perspective, in: J. Fernandes, R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering III, Vol. 6491 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 290–321. doi:10.1007/978-3-642-18023-1_7.

[22] M. Vöelter, Language and IDE modularization and composition with MPS, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering IV, Vol. 7680 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 383–430. doi:10.1007/978-3-642-35992-7_11.

[23] P. D. Mosses, The varieties of programming language semantics and their uses, in: D. Bjørner, M. Broy, A. V. Zamulin (Eds.), Perspectives of System Informatics, Vol. 2244 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 165–190. doi:10.1007/3-540-45575-2_18.

[24] P. W. Kutter, D. Schweizer, L. Thiele, Integrating domain specific language design in the software life cycle, in: D. Hutter, W. Stephan, P. Traverso, M. Ullmann (Eds.), Applied Formal Methods FM-Trends 98, Vol. 1641 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 196–212. doi:10.1007/3-540-48257-1_12.

[25] R. Mannadiar, H. Vangheluwe, Debugging in domain-specific modelling, in: B. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering, Vol. 6563 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 276–285. doi:10.1007/978-3-642-19440-5_17.

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

[27] G. Gupta, E. Pontelli, Specification, implementation, and verification of domain specific languages: A logic programming-based approach, in: A. C. Kakas, F. Sadri (Eds.), Computational: Logic Programming and Beyond, Vol. 2407 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 211–239. doi:10.1007/3-540-45628-7_10.

[28] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456 – 479, feature Interaction. doi:http://dx.doi.org/10.1016/j.comnet.2006.08.008.

[29] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, J. van der Woning, The state of the art in language workbenches, in: M. Erwig, R. Paige, E. Van Wyk (Eds.), Software Language Engineering, Vol. 8225 of Lecture Notes in Computer Science, Springer International Publishing, 2013, pp. 197–217. doi:10.1007/978-3-319-02654-1_11.

[30] B. Langlois, C. E. Jitia, E. Jouenne, DSL classification, in: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling, 2007.

[31] H. Krahn, B. Rumpe, S. Völkel, MontiCore: a framework for compositional development of domain specific languages, International Journal on Software Tools for Technology Transfer 12 (5) (2010) 353–372. doi:10.1007/s10009-010-0142-1.
[32] L. Renggli, T. Gîrba, Why smalltalk wins the host languages shootout, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST '09, ACM, New York, NY, USA, 2009, pp. 107–113. doi:10.1145/1735935.1735954.
[33] M. Fowler, Language workbenches: The killer-app for domain specific languages, http://martinfowler.com/articles/languageWorkbench.html, accessed: 2016-09-07 (2005).
[34] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, G. Konat, A language designer's workbench: A one-stop-shop for implementation and verification of language designs, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, ACM, 2014, pp. 95–111.
[35] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al., Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Computer Languages, Systems & Structures 44 (2015) 24–47.
[36] F. J. v. d. Linden, K. Schmid, E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
[37] K. Pohl, G. Böckle, F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
[38] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. rep., DTIC Document (1990).
[39] D. Benavides, S. Segura, A. Ruiz-Corts, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (6) (2010) 615 – 636. doi:http://dx.doi.org/10.1016/j.is.2010.01.001.
URL http://www.sciencedirect.com/science/article/pii/S0306437910000025
[40] R. Macala, J. Stuckey, L.D., D. Gross, Managing domain-specific, product-line development, Software, IEEE 13 (3) (1996) 57–67. doi:10.1109/52.493021.
[41] W. Cazzola, D. Poletti, DSL evolution through composition, in: Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE '10, ACM, New York, NY, USA, 2010, pp. 6:1–6:6. doi:10.1145/1890683.1890689.
[42] C. Hofer, K. Ostermann, T. Rendel, A. Moors, Polymorphic embedding of DSLs, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE '08, ACM, New York, NY, USA, 2008, pp. 137–148. doi:10.1145/1449913.1449935.
[43] T. Kosar, S. Bohra, M. Mernik, Domain-specific languages: A systematic mapping study, Information and Software Technology 71 (2016) 77 – 91. doi:http://dx.doi.org/10.1016/j.infsof.2015.11.001.
[44] L. Marques do Nascimento, D. Leite Viana, P. Silveira Neto, D. Martins, V. Cardoso Garcia, S. Meira, A systematic mapping study on domain-specific languages, in: Proceedings of the 7th International Conference on Software Engineering Advances, ICSEA 2012, Lisbon, Portugal, 2012.
[45] S. Erdweg, P. G. Giarrusso, T. Rendel, Language composition untangled, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA '12, ACM, New York, NY, USA, 2012, pp. 7:1–7:8. doi:10.1145/2427048.2427055.
URL http://doi.acm.org/10.1145/2427048.2427055
[46] I. Ober, L. Féraud, C. 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) 21–28. doi:10.1007/s11334-009-0117-0.
[47] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger, Reuse in model-to-model transformation languages: are we there yet?, Software & Systems Modeling 14 (2) (2013) 537–572. doi:10.1007/s10270-013-0343-7.
[48] A. Rodrigues da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Computer Languages, Systems & Structures 43 (2015) 139 – 155. doi:http://dx.doi.org/10.1016/j.cl.2015.06.001.
[49] B. Kitchenham, R. Pretorius, D. Budgen, O. Pearl Brereton, M. Turner, M. Niazi, S. Linkman, Systematic literature reviews in software engineering - a tertiary study, Inf. Softw. Technol. 52 (8) (2010) 792–805. doi:10.1016/j.infsof.2010.03.006.
[50] J. Sánchez Cuadrado, J. García Molina, A model-based approach to families of embedded domain-specific languages, IEEE Transactions on Software Engineering 35 (6) (2009) 825–840. doi:10.1109/TSE.2009.14.
[51] S. Tripakis, Automated module composition, in: H. Garavel, J. Hatcliff (Eds.), Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003, Springer Berlin Heidelberg, Warsaw, Poland, 2003, pp. 347–362. doi:10.1007/3-540-36577-X_25.
[52] A. Haber, M. Look, A. Navarro Perez, P. Mir Seyed Nazari, B. Rumpe, S. Volkel, A. Wortmann, Integration of heterogeneous modeling languages via extensible and composable language components, in: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, Scitepress, Angers, France, 2015.
[53] M. Torgersen, The expression problem revisited, in: M. Odersky (Ed.), Object-Oriented Programming, Vol. 3086 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 123–146. doi:10.1007/978-3-540-24851-4_6.
[54] M. Keating, Hierarchical state machines, in: The Simple Art of SoC Design, Springer New York, 2011, pp. 47–54. doi:10.1007/978-1-4419-8586-6_4.
[55] M. Mernik, An object-oriented approach to language compositions for software language engineering, Journal of Systems and Software 86 (9) (2013) 2451 – 2464. doi:http://dx.doi.org/10.1016/j.jss.2013.04.087.
[56] X. Wu, B. R. Bryant, J. Gray, M. Mernik, Component-based {LR} parsing, Computer Languages, Systems & Structures 36 (1) (2010) 16 – 33. doi:http://dx.doi.org/10.1016/j.cl.2009.01.002.
[57] B. H. Liskov, J. M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (6) (1994) 1811–1841.

doi:10.1145/197320.197383.
[58] B. Mora, F. García, F. Ruiz, M. Piattini, Graphical versus textual software measurement modelling: an empirical study, Software Quality Journal 19 (1) (2011) 201–233. doi:10.1007/s11219-010-9111-x.
[59] H. Eichelberger, K. Schmid, A systematic analysis of textual variability modeling languages, in: Proceedings of the 17th International Software Product Line Conference, SPLC '13, ACM, New York, NY, USA, 2013, pp. 12–21. doi:10.1145/2491627.2491652.
[60] M. L. Crane, J. Dingel, UML vs. classical vs. rhapsody statecharts: not all models are created equal, Software & Systems Modeling 6 (4) (2007) 415–435. doi:10.1007/s10270-006-0042-8.
[61] M. Rosenmüller, N. Siegmund, T. Thüm, G. Saake, Multi-dimensional variability modeling, in: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '11, ACM, New York, NY, USA, 2011, pp. 11–20. doi:10.1145/1944892.1944894.
[62] K. Czarnecki, S. Helsen, U. Eisenecker, Staged configuration using feature models, in: R. Nord (Ed.), Software Product Lines, Vol. 3154 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 266–283. doi:10.1007/978-3-540-28630-1_17.
[63] T. Dinkelaker, M. Monperrus, M. Mezini, Supporting variability with late semantic adaptations of domain-specific modeling languages, in: Proceedings of the First International Workshop on Composition and Variability co-located with AOSD'2010, 2010.
[64] T. Kühn, W. Cazzola, Apples and oranges: Comparing top-down and bottom-up language product lines, in: Proceedings of the 20th International Software Product Line Conference, SPLC '16, ACM, Beijing, China, 2016.
[65] M. Mernik, V. Žumer, M. Lenič, E. Avdičaušević, Implementation of multiple attribute grammar inheritance in the tool LISA, SIGPLAN Not. 34 (6) (1999) 68–75. doi:10.1145/606666.606678.
[66] M. Mernik, V. Žumer, Incremental programming language development, Computer Languages, Systems & Structures 31 (1) (2005) 1 – 16. doi:http://dx.doi.org/10.1016/j.cl.2004.02.001.
[67] D. Rebernak, M. Mernik, P. R. Henriques, M. J. ao Varanda Pereira, AspectLISA: An aspect-oriented compiler construction system based on attribute grammars, Electronic Notes in Theoretical Computer Science 164 (2) (2006) 37 – 53, proceedings of the Sixth Workshop on Language Descriptions, Tools, and Applications (LDTA 2006). doi:http://dx.doi.org/10.1016/j.entcs.2006.10.003.
[68] J. Porubän, M. Sabo, J. Kollár, M. Mernik, Abstract syntax driven language development: Defining language semantics through aspects, in: Proceedings of the International Workshop on Formalization of Modeling Languages, FML '10, ACM, New York, NY, USA, 2010, pp. 2:1–2:5. doi:10.1145/1943397.1943399.
[69] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A meta-language for modular and reusable development of DSLs, in: 8th International Conference on Software Language Engineering (SLE), Pittsburgh, United States, 2015.
URL https://hal.inria.fr/hal-01197038
[70] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented meta-languages, in: L. Briand, C. Williams (Eds.), Model Driven Engineering Languages and Systems, Vol. 3713 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 264–278. doi:10.1007/11557432_19.
[71] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, F. Fouquet, Mashup of metalanguages and its implementation in the kermeta language workbench, Software & Systems Modeling 14 (2) (2015) 905–920. doi:10.1007/s10270-013-0354-4.
[72] T. Cleenewerck, Component-based DSL development, in: F. Pfenning, Y. Smaragdakis (Eds.), Generative Programming and Component Engineering, Vol. 2830 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 245–264. doi:10.1007/978-3-540-39815-8_15.
[73] D. Ratiu, M. Vöelter, Z. Molotnikov, B. Schaetz, Implementing modular domain specific languages and analyses, in: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa '12, ACM, New York, NY, USA, 2012, pp. 35–40. doi:10.1145/2427376.2427383.
[74] T. Cleenewerck, I. Kurtev, Separation of concerns in translational semantics for DSLs in model engineering, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, ACM, New York, NY, USA, 2007, pp. 985–992. doi:10.1145/1244002.1244218.
[75] J. de Lara, E. Guerra, Deep meta-modelling with MetaDepth, in: J. Vitek (Ed.), Objects, Models, Components, Patterns, Vol. 6141 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 1–20. doi:10.1007/978-3-642-13953-6_1.
[76] B. Meyers, A. Cicchetti, E. Guerra, J. de Lara, Composing textual modelling languages in practice, in: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM '12, ACM, New York, NY, USA, 2012, pp. 31–36. doi:10.1145/2508443.2508449.
[77] I. Melo, M. Sánchez, J. Villalobos, Composing graphical languages, in: Proceedings of the First Workshop on the Globalization of Domain Specific Languages, GlobalDSL '13, ACM, New York, NY, USA, 2013, pp. 12–17. doi:10.1145/2489812.2489816.
[78] L. Pedro, M. Risoldi, D. Buchs, B. Barroca, V. Amaral, Composing visual syntax for domain specific languages, in: J. Jacko (Ed.), Human-Computer Interaction. Novel Interaction Methods and Techniques, Vol. 5611 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 889–898. doi:10.1007/978-3-642-02577-8_97.
[79] L. Pedro, M. Risoldi, D. Buchs, V. Amaral, Developing domain-specific modeling languages by metamodel semantic enrichment and composition: A case study, in: Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM '10, ACM, New York, NY, USA, 2010, pp. 16:1–16:6. doi:10.1145/2060329.2060364.
[80] S. Živković, D. Karagiannis, Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development, in: K. Gaaloul, R. Schmidt, S. Nurcan, S. Guerreiro, Q. Ma (Eds.), Enterprise, Business-Process and

Information Systems Modeling, Vol. 214 of Lecture Notes in Business Information Processing, Springer International Publishing, 2015, pp. 413–428. doi:10.1007/978-3-319-19237-6_26.

[81] C. Wende, N. Thieme, S. Zschaler, A role-based approach towards modular language engineering, in: M. Brand, D. Gasevic, J. Gray (Eds.), Software Language Engineering, Vol. 5969 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 254–273. doi:10.1007/978-3-642-12107-4_19.

[82] T. Dinkelaker, C. Wende, H. Lochmann, Implementing and composing mdsd-typical dsls, Tech. Rep. TUD-CS-2009-0156, Technische Universität Darmstadt (2009).

[83] H. Krahn, B. Rumpe, S. Vlkel, Integrated definition of abstract and concrete syntax for textual languages, in: G. Engels, B. Opdyke, D. Schmidt, F. Weil (Eds.), Model Driven Engineering Languages and Systems, Vol. 4735 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 286–300. doi:10.1007/978-3-540-75209-7_20.

[84] H. Krahn, B. Rumpe, S. Völkel, MontiCore: modular development of textual domain specific languages, in: R. F. Paige, B. Meyer (Eds.), Objects, Components, Models and Patterns, Vol. 11 of Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2008, pp. 297–315. doi:10.1007/978-3-540-69824-1_17.

[85] W. Cazzola, I. Speziale, Sectional domain specific languages, in: In Proceedings of the 4th Domain Specific Aspect-Oriented Languages (DSAL09, ACM, 2009, pp. 11–14.

[86] W. Cazzola, Domain-specific languages in few steps: The neverlang approach, in: In Proc. of Intl. Conf. on Software Composition (SC), volume 7306 of LNCS, Springer, 2012, pp. 162–177.

[87] W. Cazzola, E. Vacchi, Neverlang 2 – componentised language development for the JVM, in: W. Binder, E. Bodden, W. Löwe (Eds.), Software Composition, Vol. 8088 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 17–32. doi:10.1007/978-3-642-39614-4_2.

[88] E. Vacchi, W. Cazzola, Neverlang: A framework for feature-oriented language development, Computer Languages, Systems & Structures 43 (2015) 1 – 40. doi:http://dx.doi.org/10.1016/j.cl.2015.02.001.

[89] P. Klint, A meta-environment for generating programming environments, ACM Trans. Softw. Eng. Methodol. 2 (2) (1993) 176–201. doi:10.1145/151257.151260.

[90] M. G. J. van den Brand, J. Heering, P. Klint, P. A. Olivier, Compiling language definitions: The ASF+SDF compiler, ACM Trans. Program. Lang. Syst. 24 (4) (2002) 334–368. doi:10.1145/567097.567099.

[91] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. a language and toolset for program transformation, Science of Computer Programming 72 (12) (2008) 52 – 70, special Issue on Second issue of experimental software and toolkits (EST). doi:http://dx.doi.org/10.1016/j.scico.2007.11.003.

[92] J. Y. Marchand, B. Combemale, B. Baudry, A categorical model of model merging and weaving, in: Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 70–76.

[93] E. Vacchi, W. Cazzola, S. Pillay, B. Combemale, Variability support in domain-specific language development, in: M. Erwig, R. Paige, E. Van Wyk (Eds.), Software Language Engineering, Vol. 8225 of Lecture Notes in Computer Science, Springer International Publishing, 2013, pp. 76–95. doi:10.1007/978-3-319-02654-1_5.

[94] E. Vacchi, W. Cazzola, B. Combemale, M. Acher, Automating Variability Model Inference for Component-Based Language Implementations, in: P. Heymans, J. Rubin (Eds.), SPLC'14 - 18th International Software Product Line Conference, ACM, Florence, Italie, 2014.

[95] B. Mayoh, Attribute grammars and mathematical semantics, SIAM Journal on Computing 10 (3) (1981) 503–518. arXiv:http://dx.doi.org/10.1137/0210037, doi:10.1137/0210037.

[96] F. Jouault, I. Kurtev, Transforming models with ATL, in: J.-M. Bruel (Ed.), Satellite Events at the MoDELS 2005 Conference, Vol. 3844 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 128–138. doi:10.1007/11663430_14.

[97] W. Cazzola, D. Olivares, Gradually learning programming supported by a growable programming language, Emerging Topics in Computing, IEEE Transactions on 4 (3) (2016) 104–415. doi:10.1109/TETC.2015.2446192.

[98] D. Kolovos, R. F. Paige, T. Kelly, F. A. C. Polack, Requirements for domain-specific languages, in: Proceedings of 1st ECOOP Workshop on Domain-Specific Program Development, DSPD 2006, 2006.

[99] L. P. Tizzei, M. Dias, C. M. Rubira, A. Garcia, J. Lee, Components meet aspects: Assessing design stability of a software product line, Information and Software Technology 53 (2) (2011) 121 – 136. doi:http://dx.doi.org/10.1016/j.infsof.2010.08.007.

[100] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, W. E. Wong, Component-Based Software Quality: Methods and Techniques, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, Ch. Component-Based Software: An Overview of Testing, pp. 99–127. doi:10.1007/978-3-540-45064-1_6.

[101] J. A. Galindo, H. Turner, D. Benavides, J. White, Testing variability-intensive systems using automated analysis: an application to android, Software Quality Journal (2014) 1–41doi:10.1007/s11219-014-9258-y.

[102] H. Wu, J. Gray, M. Mernik, Unit testing for domain-specific languages, in: W. M. Taha (Ed.), Proceedings of the Working Conference on Domain-Specific Languages, DSL 2009, Springer Berlin Heidelberg, Oxford, UK, 2009, pp. 125–147. doi:10.1007/978-3-642-03034-5_7.

[103] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, D. Varró, Formal validation of domain-specific languages with derived features and well-formedness constraints, Software & Systems Modeling (2015) 1–36doi:10.1007/s10270-015-0485-x.

[104] S. Erdweg, T. Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. Vlist, G. H. Wachsmuth, J. Woning, The state of the art in language workbenches, in: M. Erwig, R. F. Paige, E. Wyk (Eds.), In Proceedings of the 6th International Conference on Software Language Engineering, SLE 2013, Springer International Publishing, Indianapolis, IN, USA, 2013, pp. 197–217. doi:10.1007/978-3-319-02654-1_11.

[105] M. Inoki, Y. Fukazawa, Software product line evolution method based on kaizen approach, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, ACM, New York, NY, USA, 2007, pp. 1207–1214. doi:10.1145/1244002.1244266.

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

[107] P. Laird, S. Barrett, Towards dynamic evolution of domain specific languages, in: M. Brand, D. Gašević, J. Gray (Eds.), Proceedings of 2nd International Conference on Software Language Engineering: Second International Conference, SLE 2009, Springer Berlin Heidelberg, Denver, CO, USA, 2010, pp. 144–153. doi:10.1007/978-3-642-12107-4_11.

[108] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Planning, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 89–116. doi:10.1007/978-3-642-29044-2_8.
URL http://dx.doi.org/10.1007/978-3-642-29044-2_8

[109] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho, Extracting and Evolving Mobile Games Product Lines, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 70–81. doi:10.1007/11554844_8.
URL http://dx.doi.org/10.1007/11554844_8

[110] H.-K. Kim, Applying Product Line to the Embedded Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 163–171. doi:10.1007/11751595_18.
URL http://dx.doi.org/10.1007/11751595_18