Reverse Engineering Language Product Lines

David Méndez-Acuña, José A. Galindo, Benoit Combemale, Arnaud Blouin, Benoit 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. In this context, an emerging phenomenon is the existence of DSLs variants, which are different versions of a DSL adapted for different purposes but that still share certain commonalities among them. In such a case, 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. DSLs previously engineered can be leveraged during the creation of new related DSLs to minimize their development cost. To this end, recent research in software language engineering proposes the use of product line engineering, thus introducing the notion of language product lines.

In this article, we aim to contribute in the field of language product line engineering. This is your solution. What is the scientific challenges you tackle in this paper ? what are the scientific challenges regarding the def of lge PL? Concretely speaking, we provide a set of meta-languages and structures to support the definition of language product lines. Afterwards, we provide a reverse engineering process that permits to automatically produce a language product line from a set of existing DSLs, thus adopting a bottom-up perspective. We validate our approach through a case study include several different DSLs implementing different formalisms for modeling finite state machines. More details on the validation: how, what, some results, what it demonstrates

Keywords: Language product lines, software languages engineering, domain-specific languages, reverse-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 of this need as an alternative to express software solutions in relevant domain concepts, thus favoring separation of concerns and hiding fine-grained implementation details [2]. DSLs are software languages whose expressiveness is focused to a well defined domain, and which provide abstractions a.k.a., *language constructs* that address a specific purpose [3]. The adoption of such a language-oriented vision has produced a large variety of DSLs. There are, for example, DSLs to build graphical user interfaces [4], to specify security policies [5], or to ease off games prototyping [6].

Despite all the advantages furnished by DSLs in terms of abstraction and separation of concerns, this approach has also important drawbacks that put into question its benefits [7]. One of those drawbacks is associated to the elevated costs of the language development process. The construction of DSLs is a time-consuming activity that requires specialized background [2]. Language designers must own solid modeling skills as well as technical knowledge to conduct the definition of complex artifacts such as metamodels, grammars, interpreters, or compilers [2]. It is worth the effort to build a DSL? That question is not always easy to answer [7]. The development of DSLs

The situation becomes more complex when we consider that, as the same as natural languages, DSLs often have different *dialects* [8]. A language dialect is a variation of a given DSL that introduces certain differences in terms of syntax and/or semantics. This type of variations appears under two situations. The first situation is when the complexity of a given domain demands the construction of several DSLs with different purposes. In such a case, the domain abstractions of the DSLs are similar but the specificities of the specific purposes require adaptations. Suppose, for example, two DSLs: the former is a DSL for specification and verification of railway scheme plans [9]; the latter is a DSL for modeling and reasoning on railway systems' capacity and security [10]. Certainly, these DSL share the same domain (i.e., railway management); hence, they share certain domain abstractions. However, each of DSL requires different semantics and specialized constructs to achieve its purpose. They both their specific

The second situation that favors the existence of DSLs variants is the use of well-known formalisms through different domains. As an example, consider the case of finite-state machines (FSMs). Generally speaking, a FSM is a an

Email addresses: david.mendez-acuna@inria.fr (David Méndez-Acuña), jagalindo@inria.fr (José A. Galindo), benoit.combemale@inria.fr (Benoit Combemale), arnaud.blouin@inria.fr (Arnaud Blouin), benoit.baudry@inria.fr (Benoit Baudry)

directed graph where nodes are states and arcs are transitions between the states. However, FSMs have been used in the construction of DSLs for a large spectrum of domains and application contexts. Naturally, a DSL inspired in FSM for building graphical user interfaces [4] requires different semantics and specialized constructs *w.r.t.* a DSL based on FSMs for games prototyping [6].

Note that the phenomenon of DSLs variants is not a problem itself. Contrariwise, it shows how different issues in a same domain can be addressed by different and complementary DSLs. Besides, it reflects the abstraction power of certain well-known formalisms –such as state machines or petri nets– that with proper adaptations result useful in several domains. However, when the same team of language designers has to deal not only with the construction of DSLs, but also with the definition of several variants, then their work becomes even more challenging and the costs of the language development process increase. After all, at implementation level each variant is a complete language itself requiring all the associated tooling such as editors, validators, interpreters, compilers, and so on.

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 [11]. The objective is to leverage previous engineering efforts to minimize implementation from scratch. To this end, the research community in software language engineering has proposed the use of Software Product Line Engineering (SPLE) in the construction of DSLs [12]. Indeed, the notion of *Language Product Line Engineering (LPLE)*, –i.e., the construction of software product lines where the products are languages– has been recently introduced [11, 13].

Similarly than in the general case of software product lines, language product lines can be built from two different approaches: top-down and bottom-up [14]. In top-down approach, a language product line is build up through a domain analysis process where the knowledge owned by experts and final users is capitalized in a language modular design and variability models. Differently, in the bottom-up approach the language product line is built up from a set of existing DSL variants. Nowadays, we can find several approaches supporting top-down language product line engineering. However, the bottom-up approach is rarely studied.

In this article, we aim to contribute in the construction of bottom-up language product lines. Concretely, we introduce a set of meta-language facilities that support the specification of a language product line. Then, we provide a reverse engineering process to automatically produce a language product line from a set of existing DSLs. This process starts with some static analysis that takes the existing DSLs and identify the commonalities and particularities to extract them a set of interdependent language modules. We also provide an approach to synthesize the corresponding variability models that permit the configuration of particular DSLs. We validate our approach through the implementation of a case study on finite state machines.

The remainder of this article is structured as follows: Section 2 describes the development scenario behind bottom up language product lines, thus characterizing the situation in which our approach results useful. Section 3 introduces our approach. Section 4 presents the case study that we use as a validation of our approach. Section 5 presents a comparison of our approach with the related work. Finally, Section 7 concludes the article.

2. Problem statement

In this section, we deeply describe the problem addressed in this article. Concretely, we characterize the development scenario in which our approach results useful. This description has two purposes. On one hand, we aim to provide an archetypal description of the development process behind bottom-up language product lines. On the other hand, we aim to clearly define the scope of our approach. That means that the solution presented in this article is useful when the development project satisfies the constraints described in this scenario; other situations will need other type of solutions.

2.1. The development scenario

Like in software product line engineering [15], in language product line engineering the development process is divided into two phases: domain engineering and application engineering (see Fig. 1). During the *domain engineering* phase, the objective is to build the language product line itself. This process includes: the design and implementation of a set of interdependent language modules that implement the language features; as well as the construction of variability models encoding the rules in which those features can be combined to produce valid DSLs.

In turn, during the *application engineering* phase, the objective is to derivate DSLs according to the needs of specific sets of final users. Such derivation process comprises: the selection of the features that should be included in the DSL, i.e., language configuration; as well as the assembly of the corresponding language modules, i.e., language modules composition.

The top-down and bottom up approaches are different ways to face the life-cycle of a language product line. In the top-down approach, the domain engineering phase is performed first, and 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.

In the bottom-up approach, the application engineering phase is performed before the domain engineering phase.

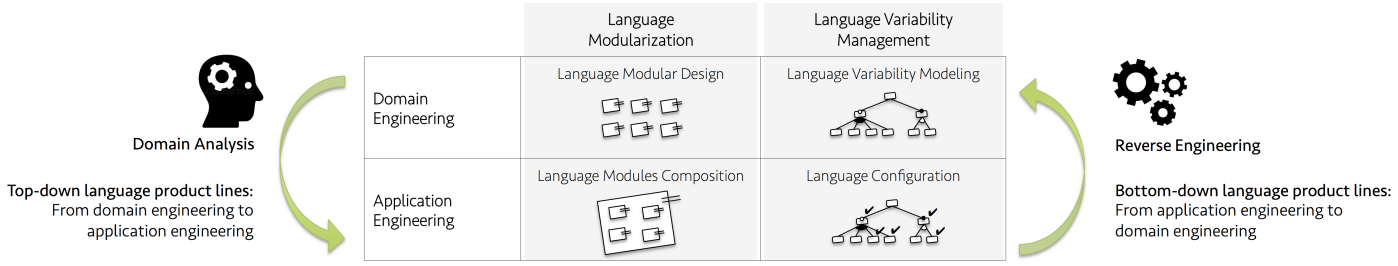

Figure 1: Phases of the life cycle of a language product line

Language designers are asked to build an initial DSL with in a well defined domain and for a specific purpose. However, with the evolution of the language development project, some variants of the DSLs are needed in order to address new requirements. Then language designers create a new variants of the initial DSL with the corresponding adaptations. At some point, language designers realize that there is potential reuse among these DSLs to build up a language product line to capitalize reuse and facilitate the construction of future DSLs. Hence, they launch a re-engineering process where the DSLs are analyzed to extract a modular language design and the corresponding variability models.

The clone-and-own approach. In this article, we are interested in bottom-up language product lines. Specifically, we aim to contribute with some strategies to reverse engineering language product lines from sets of existing DSLs which have been built through the *clone-and-own* approach. Under this approach, new variants are created by cloning existing versions of the DSL and performing the corresponding modifications. While several research works have shown that such a practice is quite common in software development projects [16, 17], in a recent work [18] we provided some empirical evidence showing that it is also a common practice in language development. We performed an analysis of a large pool of DSLs obtained from GitHub, and we detected a relevant amount of specification clones among them.

A running example. Let us illustrate the development scenario described above through a running example. Suppose a team of language designers working on the construction of the DSL for finite state machines. To this end, language designers follow the UML specification [19] thus defining language constructs such as states, regions, transitions, triggers, and so on. Those language constructs are specified in terms of their syntax and semantics so, at the end of the language development process, they release a DSL which can be executed and which behavior complies the UML specification.

Once this first DSL is released, a team of language designers is asked to build a new variant of the DSL. This time, the DSL must comply the Rhapsody specification [20] (i.e., another formalism to finite state machines). This new variant shares many commonalities with UML but it is not exactly the same. There are differences at the level of both syntax and semantics. At this point, language designers face the problem of reusing as much as possible the constructs defined in the DSL for UML state machines.

One of the possibilities that language designers have to deal with this challenge is to use the clone-and-own approach by copy-pasting the specification of the first DSL in a new project, and then performing the corresponding adaptations. Although the disadvantages of this practice are usually well known by software developers (including language designers), it permits a fast prototyping of the second DSL. After this process, language designers have two different DSLs that share many specification clones.

Suppose now that the final users realize that they need support for other formalisms for state machines. Hence, the team of language designers is asked to implement two more variants of the DSL: one complying the Stateflow specification [21], and the other one complying classical Harel state machines [22]. If the language designers use again the clone-and-own approach, then the result would be a set of four DSLs for state machines that share some commonalities in the form of specification clones, and which own certain particularities that make them unique.

The problem that language designers face at this point is two fold. First, they will have to produce a new variant of the DSL for each state machines formalism needed by final users. Even worst, suppose for example that final users need to combine some specifications in order to produce hybrid formalisms. Language designers will have to produce a new version of the DSL for each desired combination and the clone-and-own approach becomes impractical. Second, the existence of specification clones increases the maintenance costs of the involved DSLs. If language designers detect bugs in one of the specifications, it is quite probable that such bugs is replicated in all the DSLs sharing the specification clone.

This type of situations have been largely discussed in the software engineering community. There are several approaches that exploit the notion of clones to produce software product lines from existing product variants [23, 24, 25]. In this article, do the proper for the case of language product lines.

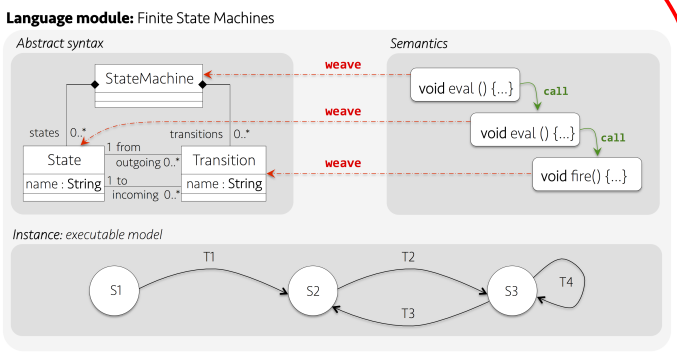

Figure 2: A simple DSL for finite state machines

2.2. Technological scope

It is worth clarifying that all the ideas presented in this article are focused to executable domain specific modeling languages (xDSMLs) where the abstract syntax is specified through *metamodels*, and the dynamic semantics is specified operationally as a set of *domain specific actions* [26]. Domain specific actions are Java-like methods that introduce behavior in the metaclasses of a given metamodel. Such injection is performed via weaving as the same as in aspect-oriented programming [27]. Concrete syntax, and hence concrete syntax variability, are out of the scope of this article.

Fig. 2 illustrates this type of DSLs through a simple example on finite states machines. In that case, the metamodel that implements the abstract syntax contains three metaclasses: *StateMachine*, *State*, and *Transition*. There are some references among those metaclasses representing the relationships existing among the corresponding language constructs. The domain specific actions at the right of the Fig. 2 introduce the operational semantics to the DSL. In this example, there is one domain specific action for each metaclass. Note that the interactions among domain specific actions can be internally specified in their implementation by means of the *interpreter pattern*, or externalized in a model of computation [26].

3. Proposed Approach

In this section, we introduce our approach to support bottom-up language product lines. Our approach is explained in two parts. First, we present some meta-language facilities needed to define a language product line in terms of language modules, features, and variability models. Then, we introduce a reverse engineering algorithm to automatically build a language product line from a set of existing DSL variants that have been built through the clone-and-own approach.

3.1. Meta-language Facilities to Define Language Product Lines

As we explained earlier, the definition of a language product line requires support for languages modularization and variability management. This support should be materialized in some facilities at the level of the meta-languages used during the language development process. In the following, we explain how we use and extend current meta-languages to this end.

3.1.1. Support for Language 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 language module provides a set of language constructs, and a DSL is obtained from the composition of two or more language modules.

Just as in typical software modularization [28], 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.

Achieving separability through language interfaces. As aforementioned, we propose the use of language interfaces to achieve separability of language modules. In particular, we propose the classical required and provided interfaces explained below.

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

As the reader might infer from the last paragraph, the needs of a language module can be materialized in the form of required constructs. In assuming so, the specification of a language module would be composed of a set of actual constructs, which are being implemented by the module; and a set of required constructs, which represent needs to other modules. This approach would result useful to support the modularization scenario called aggregation where the needs of language modules are entire constructs implemented in foreign modules. However, a finer level of granularity might be necessary to support other modularization scenarios such as extension where the needs are

i do not understand here. You propose a bottom-up approach, so starting from legacy DSLs, and here you say that langauge designer creates dsls through language module.

avoid "some"

have to define earlier what is a meta-language, their use.

what is the relation between these 2 paragraphs?

Scope of the approach

Too brutal: focus on metamodels, concrete synt for future work.

why talking about weaving in technological scope? For me scope = limits of the proposal

why bottom-up? Did you motivate that in the intro and here?

i do not understand. Bottom-up means that you start from existing languages. So, why "define"? Maybe "represent"? Before talking about that, explain the big picture of a bottom-up approach: analysis of legacy dsls, then...

Hard to understand.

Keep in mind that each paragraph must be the logical consequence of the previous one

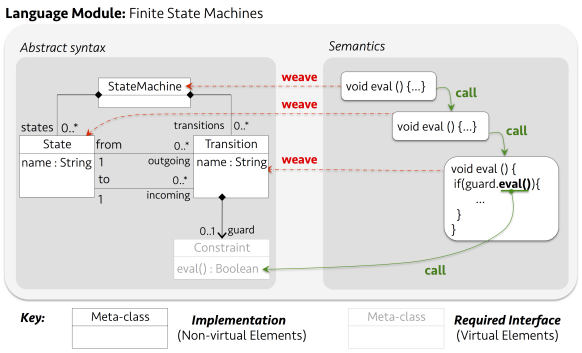

Figure 3: Example of the use of required interfaces

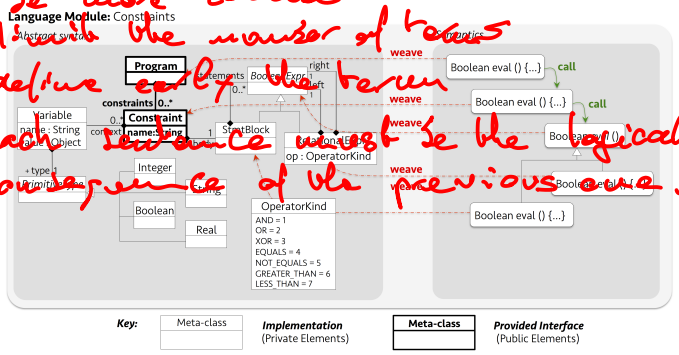

Figure 4: Example of the use of provided interfaces

not necessarily entire constructs but finer elements such as properties or operations. The details of the use of the required interfaces in extension will be explained later in this chapter.

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

Fig. 3 illustrates the use of required interfaces through the example introduced before regarding a language module for finite state machines that requires a constraint language. In that example, the constructs proper the state machines are non-virtual elements since they correspond to actual implementation of such abstractions. In turn, the constraints to express the guards in the transitions are expressed as a virtual construct called *Constraint* that provides a virtual operation action called *eval()* which is used in the specification of the semantics of the meta-class *Transition*.

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

To support the definition of provided interfaces, we propose to extend EMOF with the notion of *module visibility*. This extension allows to classify a certain specification element as either ***public*** or ***private*** according to its nature. For example, a language designer can classify a meta-class as ***public*** meaning that it represents essential functionality of the module so can be used by external modules and it belongs to the provided interface. Naturally, if the meta-class is classified as ***private*** it cannot be used by external modules and it cannot be considered as part of the provided interface. Note that the notion of module visibility is different from the notion of visibility already defined in EMOF. The later is associated to certain access constraints of model elements with respect to the package in which they are implemented.

Fig. 4 illustrates the use of provided interfaces through the example introduced before regarding the constraints module. Since the main functionality of the module is to define and evaluate constraints, the meta-classes included in the provided interface (so those one defined as ***public*** in terms of module visibility) are: *ConstrainedProgram* and *Constraint* including the operations that implement their semantics.

Achieving composability through interfaces binding. Now, we need to define the way in which those interfaces interact each other at the moment of the composition. In doing so, it is important to conciliate two different (and potentially conflicting) issues. Firstly, safe composition of the involved language modules should be guaranteed; we need to check the compatibility between a providing and a requiring language module by verifying that the functionality offered by the former actually fulfills the needs of later. Secondly, there must be some place for substitutability; compatibility checking should offer certain flexibility that permits to perform composition despite some differences in their definitions. This is important because when language modules are development independently of each other, their interfaces and implementations not always match [29].

To deal with the aforementioned issues, we propose an approach for compatibility checking. It is at the same time strict enough to guarantee safe composition, and flexible enough to permit substitutability under certain con-

Too much text. Figures to add to illustrate and lighten the text.

ditions. Concretely, we propose to extract both required and provided interfaces in the form of *model types* [30]. The model type corresponding to the required interface contains the virtual specification elements of a language module whereas the model type corresponding to the provided interface the model type contains its public specification elements. The relationship between a model type and a language module is called ***implements*** and it is introduced by Degueule et al. [31].

In order to perform compatibility checking, we use the sub-typing relationship –introduced by Guy et al. [32]– between the model types corresponding to the provided and the required interfaces. This relationship imposes certain constraints that guarantee safe composition while permitting some freedom degrees thus introducing some flexibility. In particular, under this definition of sub-typing the most obvious manner to guarantee safe composition is to check two conditions: (1) all the needs expressed in the requiring model type are furnished in the providing model type (**total** sub-typing); and (2) the two model types have exactly the same shape (**isomorphic** sub-typing). However, this definition of sub-typing also provides two dimensions of flexibility: **partial** sub-typing and **non-isomorphic** sub-typing.

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

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

The principle behind isomorphic sub-typing is that the needs in the requiring interface are not always expressed exactly as the functionality offered by the providing module is expressed in the providing interface. For example, model type in Fig. 3 expresses the needs in terms of constraints of state machines through a class *Constraint* with an operation *eval()*. If we want to use OCL to satisfy these needs, then we will find that there is not a class constraint but *OclExpr*. Besides, the operational semantics might be implemented differently. In this case, we need an adapter that permit to find the correspondences among the elements of the model types.

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

Language modules composition. Once the compatibility between two language modules is correctly checked, the next step is to compose the language modules to integrate their functionality i.e., the needs of the requiring module are fulfilled with the services offered by the provided one. In our approach, this composition is performed in two phases. First, there is a matching process that identifies one-to-one matches between virtual and public elements from the required and provided interface respectively. This match can be identified automatically by comparing names and types of the elements (where applicable). However, the match can be also specified manually in the case of non-isomorphism.

Once the match is correctly established, the composition process continues with a merging algorithm that replaces virtual elements with public ones. That means also to replace all the possible references existing to the virtual element to point out to the corresponding public element. When the process is finished, we re-calculate both provided and required interfaces. The provided interface of the composition is re-calculated as the sum of the public elements of the two modules under composition. In turn, the required interface of the composition is re-calculated as the difference of the required interface of the required module minus the provided interface of the providing module.

3.1.2. Support for Language Variability Management

The challenge towards supporting the variability existing in a language product line is that such variability is multi dimensional. Because the specification of a DSL involves several implementation concerns, then there are several dimensions of variability i.e., abstract syntax variability, concrete syntax variability, and semantic variability [33, 34]. Abstract syntax variability refers to the capability of selecting the desired language constructs for a particular type of user. In many cases, constructs are grouped in *language features* to facilitate the selection. Such grouping is motivated by the fact that selecting constructs can be difficult because a DSL usually has many constructs, so a coarser level of granularity is required. In turn, concrete syntax variability refers to the capability of supporting different representations for the same language construct. Finally, semantic variability refers to the capability of supporting different interpretations for the same language construct.

In this section we present a strategy to deal with this type of variability. To this end, we present an approach to represent the variability existing in a language product line. Then, we explain how we can use the variability models to configure concrete DSLs. As the same as our approach to language modularization, our approach to

Idem

variability management is scoped to abstract syntax and semantics; concrete syntax –and hence, concrete syntax variability– is not being considered in the solution.

Modeling multi-dimensional variability. A solution to represent abstract syntax variability and semantic variability should consider two main issues. Firstly, the definition of the semantics has a strong dependency to the definition of the abstract syntax –the domain-specific actions that implement the semantics of a DSL are weaved in the meta-classes defined in the abstract syntax–. Hence, these dimensions of variability are not isolated each other. Rather, the decisions made in the configuration of the abstract syntax variability impact the decisions that can be made in the configuration of the semantic variability. For example, there is a variation point for the case of state machines that proposes different ways to deal with conflicting transitions. However, conflicting transitions are only possible within hierarchical state machines. Hence, if the state machine DSL does not support hierarchical states, then it makes no sense to configure this semantic variation point.

The second issue to consider at the moment of dealing with language variability management is that a semantic variation point might be transversal to several meta-classes. For example, there is a semantic variation point in hierarchical state machines that corresponds to decide if the state machine follows either the run-to-completion policy or if it supports simultaneous events [35]. In any case, the implementation of this semantics involves code in domain-specific actions for the *StateMachine* and *Region* meta-classes. Moreover, if the involve meta-classes are introduced by different language modules in the abstract syntax, then the semantic variation point depends of two features. Hence, the relationship between a feature in the abstract syntax and a semantic variation point is not necessarily one-to-one.

Currently, we can find several approaches to support multi dimensional variability (e.g., [36]). Some of those approaches have been applied concretely to language product lines. In particular, they propose to model all the dimensions of variability through feature models and then to establish dependencies among them. However, the differ in the way in which the features are organized. For example, in some cases there is a unique variability model with one branch for each dimension of variability whereas there are other approaches that propose the use of one feature model for each dimension and then they use cross-tree constraints to represent the dependencies among the dimensions.

In this article we propose a different approach to deal with language variability management. Concretely, we propose to combine the use of feature models with orthogonal variability models; feature models are used to model abstract syntax variability and orthogonal variability models are used to model semantic variability. Fig. 5 shows illustrates our approach. At the top of the figure, we find a feature model which each feature represents one language

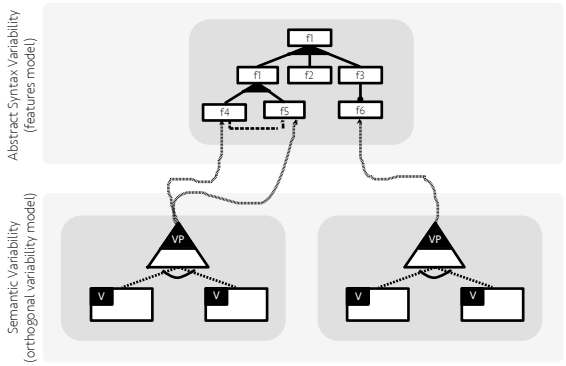

Figure 5: Approach to represent multi-dimensional variability in language product lines

module. At the bottom, we have an orthogonal variability model that represents the semantic variation points existing among those language modules. As the reader might imagine, each of the variants of the variation points are associated not only to a particular module but also to the corresponding domain-specific actions associated to its implementation.

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

(1) The structure of orthogonal variability models is more appropriated. As explained by Roos-Frantz et al. [?], feature models and orthogonal variability models are similar. However, they have some structural differences. One of those differences is that whereas a feature model is a tree that can have many levels, an orthogonal variability model Is set of trees each of which has two levels. Each tree represent one variability point and its children represent variants to that variation point.

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

(2) The meaning of orthogonal variability models is more appropriated. According to [37], a language feature is a characteristic provided by the language which is visible to the final user. This definition can be associated abstract syntax variability and the use of feature models can be appropriated to represented it. All the approaches

on language product line engineering use feature models to this end showing that it is possible and appropriated.

The case of the semantic variability is different, however. A semantic decision is not a characteristic of a language that we can select or discard. The semantic of a DSL should be always specified if the DSLs is intended to be executable. Rather, a semantic decision is more a variation point that can have different interpretations captured as variants. Note vocabulary fits better in the definitions provided by orthogonal variability models. More than features, we have variation points and variants, which also suggest that the use orthogonal variability models is more appropriate to represent semantic variability.

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

Multi-staged configuration was introduced by Czarnecki et al. [38] for the general case of software product lines, and discussed by Dinkelaker et al. [39] 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 [39]. In that case, the configuration process is as follows: the language designer provides an initial configuration. Then, the configuration is continued by the final user that can use the DSL as long as the configuration is complete. In doing so, it is important to decide what decisions correspond to each stakeholder.

Suppose the scenario introduced in Fig. 6 where the language designer is responsible to configure the abstract syntax variability whereas the language user is responsible to configure the semantics. When the language designer finishes its configuration process, the orthogonal variability models will be available so the final user can perform the configuration of the semantics. This orthogonal variability model will only include the variation points that are relevant to the features included in the configuration of the abstract syntax. Moreover, because each of the semantic variation points are represented separately in a different tree, then we can imagine a scenario where the language designer is able to configure not only the abstract syntax but also some semantic variation points, and then delegate to the final user only the decisions that he/she can take according to its knowledge.

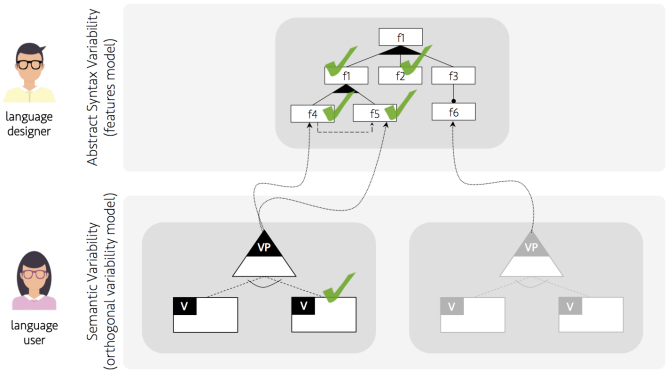

Figure 6: Approach to support multi-staged configuration of language product lines

3.2. The reverse engineering process

In this section, we introduce a reverse engineering strategy for bottom up language product line engineering. Fig. 7 presents an overview of the approach. It receives as input a set of DSLs implemented according to the development process described in Section 2.1 and under the technological space introduced in Section 2.2.

The first step of the process is the extraction of the language modules that implement the features included in the language product line. The process continues with the synthesis of a variability model that captures both abstract syntax and semantic variability. Such a model can be later used to configure and derive concrete DSLs.

3.2.1. Reverse engineering reusable language modules

Let us start the description of our reverse engineering strategy by explaining the way in which we extract the language modules. Roughly speaking, our approach takes the DSLs given in the input and break them down into several language modules. The purpose of such a breaking down process is to remove all the specification clones existing among the given DSLs, thus reducing the maintenance costs associated to the DSLs[1]. To this end, we analyze the given DSLs to detect the existing specification clones. Then, we extract those clones in separate language modules that can be later included from the involved DSLs.

Principles for language modules' reverse engineering. Our strategy for reverse engineering reverse engineering reusable language modules is based on five principles that will be introduced in this section. Then, we explain how we use those principles to extract a catalog of reusable language modules that implement the language features provided by the language product line.

Principle 1: *DSL specifications are comparable. Hence, specification clones can be detected automatically.* Two

[1]Note that we assume the existence such as clones; after all, the involved DSLs were built up by using the clone-and-own approach.

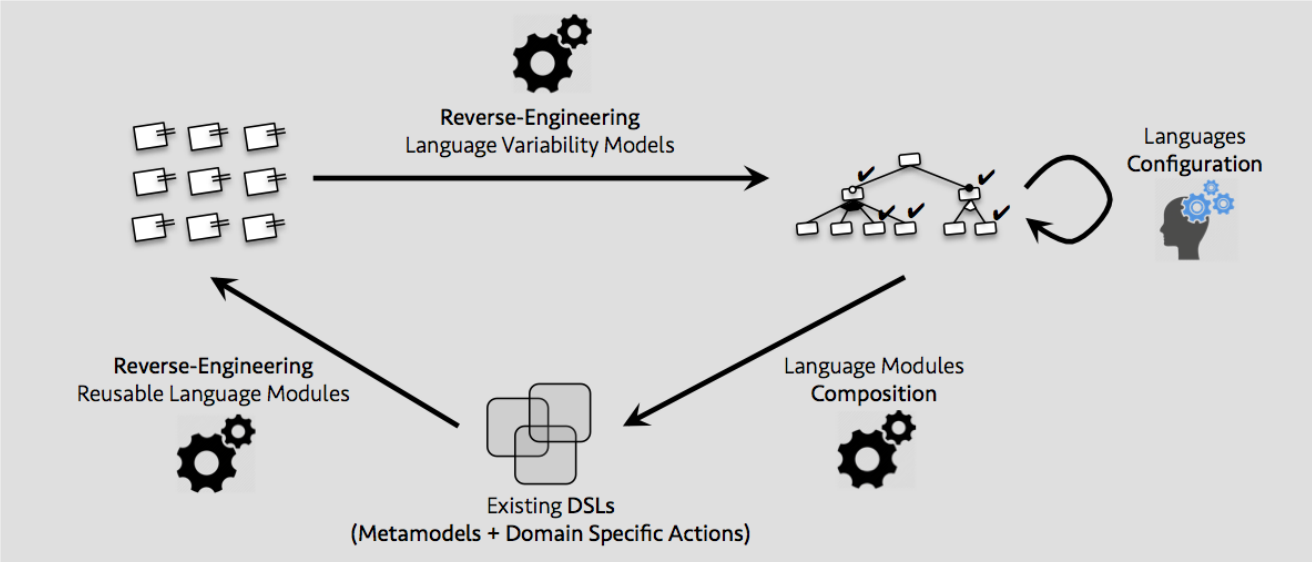

Figure 7: Reverse engineering language product lines: approach overview

DSL specifications can be compared each other. This comparison can be either coarse grained indicating if the two specifications are equal regarding both syntax and semantics, or fine-grained detecting segments of the specifications that match. The latter approach permits to identify specification clones between two DSLs and supposes the comparison of each specification element.

For the technological space discussed in this paper, specification elements for the abstract syntax are metaclasses whereas specification elements for the semantics are domain specific actions.

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

$$\doteqdot \ : MC \times MC \rightarrow bool \tag{1}$$

$$\begin{aligned} & MC_A \doteqdot MC_B = true \implies \\ & \quad MC_A.name = MC_B.name \wedge \\ & \quad \forall a_1 \in MC_A.attr \mid (\exists a_2 \in MC_B.attr \mid a_1 = a_2) \wedge \\ & \quad \forall r_1 \in MC_A.refs \mid (\exists r_2 \in MC_B.refs \mid r_1 = r_2) \wedge \\ & \quad |MC_A.attr| = |MC_B.attr| \wedge \\ & \quad |MC_A.refs| = |MC_B.refs| \end{aligned} \tag{2}$$

Comparison of domain specific actions. For comparison for domain specific actions we need to take into account that –like methods in Java– domain specific actions have a signature that specifies its contract (i.e., return type, visibility, parameters, name, and so on), and a body where the behavior is implemented. Two domain specific actions are equal if they have the same signature and body.

Whereas comparison of signatures can be performed by syntactic comparison of the signature elements, comparison of bodies can be arbitrary difficult. If we try to compare the behavior of the domain-specific actions, then we will have to address the semantic equivalence problem, which is known to be undecidable [40]. To address this issue, we conceive bodies comparison in terms of its abstract syntax tree as proposed by Biegel et al. [41]. In other words, to compare two bodies, we first parse them to extract their abstract syntax tree, and then we compare those trees. Note that this decision makes sense because we are detecting specification clones more than equivalent behavior. Formally, comparison of domain-specific actions (DSAs) is specified by the operator $\coloneqq$.

$$\coloneqq \ : DSA \times DSA \rightarrow bool \tag{3}$$

$$\begin{aligned} & DSA_A \coloneqq DSA_B = true \implies \\ & \quad DSA_A.name = DSA_B.name \wedge \\ & \quad DSA_A.returnType = DSA_B.returnType \wedge \\ & \quad DSA_A.visibility = DSA_B.visibility \wedge \\ & \quad \forall p_1 \in DSA_A.params \mid \\ & \quad (\exists p_2 \in DSA_B.params \mid p_1 = p_2) \wedge \\ & \quad |DSA_A.params| = |DSA_B.params| \wedge \\ & \quad DSA_A.AST = DSA_B.AST \end{aligned} \tag{4}$$

Principle 2: *Specification clones can be viewed as sets overlapping.* If a DSL specification is viewed as sets of metaclasses and domain specific actions, then specification clones can be viewed as intersections (a.k.a., overlapping)

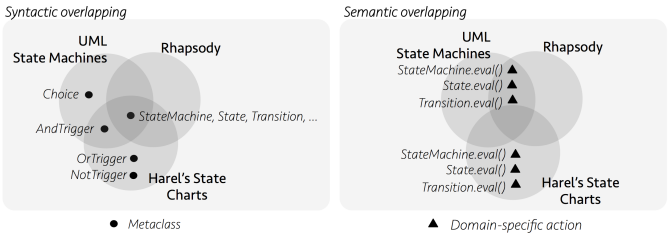

Figure 8: Syntactic and semantic overlapping in a set of DSLs

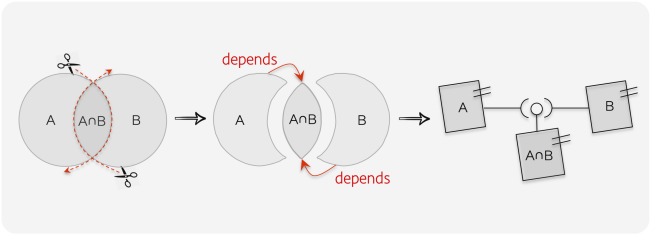

Figure 9: Breaking down overlapping to obtain language modules

of those sets. Figure 8 illustrates this observation for the case of the motivation scenario introduced in Section 2. We use two Venn diagrams to represent both syntax and semantic overlapping.

In the case of abstract syntax overlapping, the Venn diagram shows that the classical concepts for state machines such as StateMachine, State, and Transition are in the intersection of the three DSLs given in the input i.e., UML state machines, Rhapsody, and Harel's state machines. In turn, there are certain particularities for each DSL. For example, the concept AndTrigger is owned by UML and Harel state machines but not for Rhapsody. Concepts such as OrTrigger and NotTrigger are only provided by Harel state machines since the concept of Choice is exclusive of UML state machines.

For the case of semantic variability, we can see that the intersection is empty which means that there is not a common semantic for the DSLs. In other words, there is not a behavior that we can identify as common among the three DSLs. Rather, UML state machines and Rhapsody share the domain specific actions corresponding to the concetps of State Machine, State, and Transition. In turn, the implementation of Harel state machines is different. Note that this figure is a simplification of the semantic overlapping. All the details will be given later in this article.

In that case, the fact that the expression language is used in all the DSLs is represented by the intersection in the center of the diagram where the three sets overlap the metaclass `Expression` (and its domain-specific actions). In turn, the intersection between the state machines DSL and Logo shows that they overlap the metaclass `Constraint` that belongs to the constraint language. Note that the identification of such overlapping is only possible when there are comparison operators (principle 1) that formalize the notion of equality.

Principle 3: *Breaking down overlapping produces reusable modules.* According to principle 2, overlapping between two DSLs implies the existence of repeated metaclasses and/or domain specific actions (i.e., specification clones). Those repeated elements can be specified once and reused in the two DSLs [42, p. 60-61]. Hence, reusable language modules can be obtained by breaking down the overlapping existing among DSL specifications as illustrated in Figure 9; each different intersection is encapsulated in a different language module.

Principle 4: *Abstract syntax first, semantics afterwards.* Since the abstract syntax of a DSL specifies its structure in terms of metaclasses and relationships among them, the domain-specific actions add executability to the metaclasses. Hence, the abstract syntax is the backbone of the DSL specification, and so, the process of breaking down overlapping should be performed for the abstract syntax first. Afterwards, we can do the proper for the semantics. In doing so, we need to take into consideration the phenomenon of semantic variability. That is, two cloned metaclasses might have different domain-specific actions. That occurs when two DSLs share some syntax specification but differ in their semantics.

Principle 5: *Breaking down a metamodel is a graph partitioning problem.* The metamodel that specifies the abstract syntax of a DSL can be viewed as a directed graph G.

$$G =< V, A >$$

where:

- **V**: is the set of vertices each of which represents a metaclass.
- **A**: is the set of arcs each of which represents a relationships between two meta-classes (i.e., references, containments, and inheritances).

This observation is quite useful at the moment of breaking down a metamodel to satisfy the principle 4. Breaking down a metamodel can be viewed as a graph partitioning problem where the result is a finite set of subgraphs. Each subgraph represents the metamodel of a reusable language module.

The 5 principles in action. The reverse-engineering strategy to produce a catalog of reusable modules is illustrated in Figure 10. It is composed of two steps: identifying overlapping and breaking down.

Identifying overlapping: match and merge. To identify syntactic overlapping in a given set of DSLs, we start by producing a graph for each DSL according to the principle 5. Then, we identify specification clones (the matching phase) using the comparison operators defined in principle 1. After that, we have a set of graphs (one for each DSL) and a set of matching relationships among some of the vertex. At that point we can proceed to create the

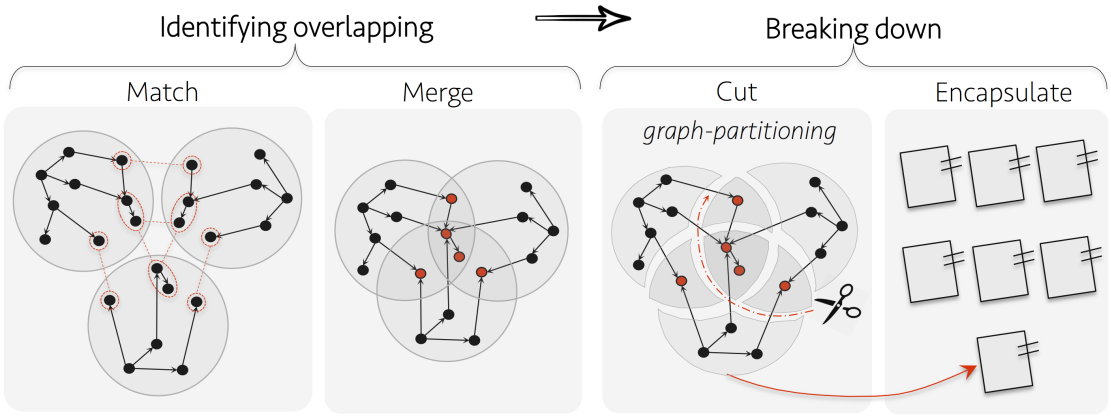

Figure 10: Breaking down the input set by cutting overlapping

overlapping defined in principle 2. To this end, we merge the matched vertex as illustrated in the second square of Figure 10. This merging permits to remove cloned metaclasses.

To identify semantic overlapping, we check whether the domain-specific actions of the matched metaclasses are equal as well. If so, they can be considered as clones in the semantic specification, so there is semantic overlapping. In that case, these domain-specific actions are merged. If not all the domain-specific actions associated to the matched metaclasses are the same, different clusters of domain-specific actions are created, thus establishing semantic variation points.

Breaking down: cut and encapsulate. Once overlapping among the DSLs of the portfolio has been identified, we extract a set of reusable language modules. This process corresponds to break-down the graph produced in the last phase using a graph partitioning algorithm. The algorithm receives the graph(s) obtained from the merging process and returns a set of vertex clusters: one cluster for each intersection of the Venn diagram. Arcs defined between vertices in different clusters can be considered as cross-cutting dependencies between clusters. Then, we encapsulate each vertex cluster in the form of language modules. Each module contains a metamodel, a set of domain-specific actions, and a set of dependencies towards other language modules.

Note that the dependencies between language modules can be viewed through the interfaces introduced in Section 3.1. Those interfaces are reverse engineered from each module. Required interfaces are generating by creating a virtual element for those elements that are required by the module but that are not part of its definition. The provided interface is generated by defining all the specification elements of the language module as public. If there are specification elements that should be hidden, then the language designer should modify the generated definition.

3.2.2. Reverse Engineering Language Variability Models

Once we have obtained a set of reusable language modules from a given set of DSLs, we need to represent the variability of the language product line. In other words, once we have broken down a set of existing DSLs by identifying commonalities and particularities, we need to represent those commonalities and particularities in a model that permits to configure concrete DSLs.

To achieve such a challenge, we propose a reverse engineering algorithm to synthesize variability models from a set of reusable language modules. Our algorithm produces not only a feature model with the abstract syntax variability, but also an orthogonal variability model representing the semantic variability. An overview of the approach is presented in Fig. 11, and the remainder of this section is dedicated to explain it in detail.

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

In the literature, there are several approaches for reverse engineering feature modules from dependencies graphs (consider for example the approach presented by Assunçao et al. [43], or the one presented by She et al., [44]). In our case, we opt for an algorithm that produces a simple feature model where each language module is represented in a concrete feature, and where the dependencies between language modules are encoded either by parent-child relationships or by the classical *implies* relationship. Our algorithm was inspired from the approach presented by Vacchi et al. [45] which fulfills the aforementioned require-

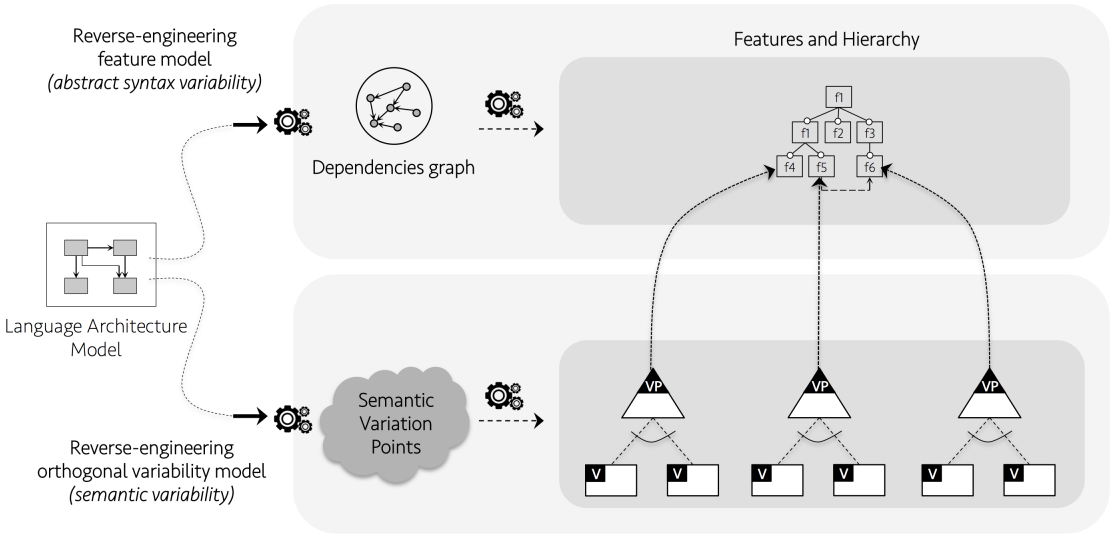

Figure 11: Reverse-engineering variability models for language product lines.

ments. Besides, it has been applied for the particular case of languages variability.

The tooling that supports our algorithms is flexible enough to permit the use of other approaches for synthesis of feature model. To this end, we provide an interface called `ISynthesis`. In order to extend our approach with a new algorithm for synthesis of feature models, language designers just need to implement such interface. In addition to the one proposed by Vacchi et al. [45], we have integrated our approach with the one provided by Assunçao et al., [43].

Reverse-engineering orthogonal variability models representing semantic variability. Once the feature model encoding abstract syntax variability is produced, we proceed to do the proper with the orthogonal variability model encoding semantic variability. To this end, we need to analyze the results of the process for extracting the language modules. As explained in Section 3.2.1, according to the result of the comparison of the semantics, a language module might have more than one cluster of domain specific actions. This occurs when the two DSLs share constructs that are equal in terms of the abstract syntax, but differ in their semantics. Since this is the definition of semantic variation point, we materialize those clusters in semantic variation points of an orthogonal variability model.

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

4. Case study: Finite State Machines

To evaluate of our approach, we use as case study the set of DSLs for state machines that we have used as running example all along this article. This case study is inspired from the analysis of variability on languages for finite state machines provided by Crane et al. [35], and it is composed of three different DSLs: UML state diagrams, Rhapsody, and Harel's state charts. These three DSLs have some commonalities since they are intended to express the same formalism. According to the development scenario we address in this paper, these commonalities will be materialized as clones in the DSL specifications. In this section, we summarize both commonalities and differences existing in the case study. Then, we apply our approach and we present the obtained results.

4.1. Description of the commonalities

Generally speaking, state machines are graphs where nodes represent states and arcs represent transitions between the states [46]. The execution of a state machine is performed in a sequence of *steps* each of which receives a set of events that the state machine should react to. The reaction of a machine to set of events can be understood as a passage from an initial configuration (t_i) to a final configuration (t_f). A configuration is the set of active states in the machine.

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

transition is fired if and only if the evaluation of the guard returns true at the moment of the trigger arrival.

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

All of the DSLs included in this case study support the notion of region. A state machine might be divided in several regions that are executed concurrently. Each region might have its own initial and final (pseudo)states. In addition, the DSLs also support the definition of different types of actions. States can define entry/do/exit actions, and transitions can have effect actions.

4.2. Description of the variability

Abstract syntax variability. Differences at the level of the abstract syntax between the DSLs under study correspond to the diversity of constructs each of those DSLs provide. In particular, there are differences in the support for transition's triggers and pseudostates.

In the case of transitions' triggers, whereas Rhapsody only supports atomic triggers, both Harel's statecharts and UML provide support for composite triggers. In Harel's statecharts triggers can be composed by using `AND`, `OR`, and `NOT` operators. In turn, in UML triggers can be composed by using the `AND` operator.

In the case of pseudostates, whereas all the DSLs support `Fork`, `Join`, `ShallowHistory`, and `Junction`, there are two psueudostates i.e., `DeepHistory` and `Choice` that are only supported by UML. The `Conditional` pseudostate is only provided by Harel's state charts. Figure 12 shows the language constructs provided by each DSL.

Semantic variability. Semantic differences between the DSLs under study can be summarized in three issues:

(1) Events dispatching policy: The first semantic difference in the operational semantics of state machines refers to the way in which events are consumed by the state machine. In a first interpretation, simultaneous events are supported i.e., the state machine can process more than one event in a single step. In a second interpretation, the state machine follows the principle of run to completion i.e., the state machine is able only of supporting one event by step so several events require several steps.

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

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

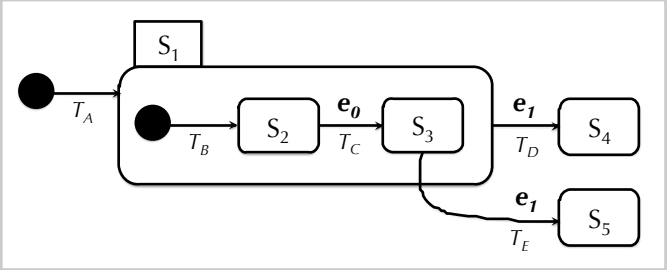

Figure 13: Example of a state machine with conflicting priorities

The first way of executing the effects of a transition is by respecting the order in which they are defined. This is due to the fact that transitions effects are usually defined by means of imperative action script languages where the order of the instructions is intrinsic. The second interpretation to the execution of transitions' effect is to execute them in parallel. In other words, the effects are defined as a set of instructions that will be executed at the same time so no assumptions should be made with respect to the execution order.

UML and Rhapsody execute the transition effects in parallel. Harel's statecharts execute transition effects simultaneously.

(3) Priorities in the transitions: Because several transitions can be associated to the same event, there are cases in which more than one transitions are intended to be fired in the same step. In general, all the DSLs for state machines agree in the fact that all the activated transitions should be fired. However, this is not always possible because conflicts might appear. Consider for example the state machine presented in Fig 13. The transitions T_D and T_E are conflictive because they are activated by the same event i.e., e_2, they exit the same state, and they go to different target states. Then, the final configuration of the state machine will be different according to the selected transition.

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

In the example presented in Fig 13 the dispatched transition according to this policy would be the transition T_E so the state machine would move to the state S_5. The second mechanism for solving conflicts in the transition is

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

Figure 12: Diversity of constructs provided by the DSLs for state machines

to select the transition with the higher scope. That is, the higher transition w.r.t. the hierarchy of the state machine. In the example presented in Fig 13 the dispatched transition according to this policy is the transition T_D so the state machine will move to the state S_4.

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

4.3. Applying our approach

The starting point of the applicability of our approach in the case study is a set of DSLs implementing each of the specifications explained above. Hence, at the beginning we have three different DSLs for state machines that can be accessed in a GitHub repository[2]. Using these specifications as input, we proceed to apply our approach and we obtained the following results:

Reverse engineering a language product line. Once we have the result of the static analysis where specification clones are detected, we proceed to reverse engineering the language product line. The results are summarized in Fig. 14. At the left of the figure we present the set of language modules we obtained as well as the language interfaces existing among them. Those modules group the language constructs according to the heuristic introduced in Section 3.2.1 on breaking down overlapping. At the right of the figure we show the corresponding variability models. Each feature of the feature models is associated to a given language module. In turn, the semantic variability points in the orthogonal model are associated to clusters of domain specific actions.

5. Related work

Under construction... :)

6. Discussion: Broadening the spectrum

The approach presented in this article is only useful when language designers follow the development scenario described in Section 2.1 while using the technological space mentioned in Section 2.2. Along this paper we show how we can reduce maintenance costs and exploit variability if those conditions are fulfilled. In this section we open the broaden by discussing potential directions to support more diverse scenarios.

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

The answer to that question relies on the definition of more complex comparison operators. As we deeply explain in Section 3.2, the very first step of our reverse engineering strategy is to perform a static analysis of the given DSLs and apply two comparison in order to specify specification clones. If what we want is to find commonalities that are not necessarily materialized in specification clones but in "equivalent functionality", then we need to enhance the comparison operators in order to detect such as equivalences.

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

7. Conclusions

In this article we presented a reverse-engineering process that allows to automatically produce a language product line from a set of existing DSLs. Our reverse-engineering

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

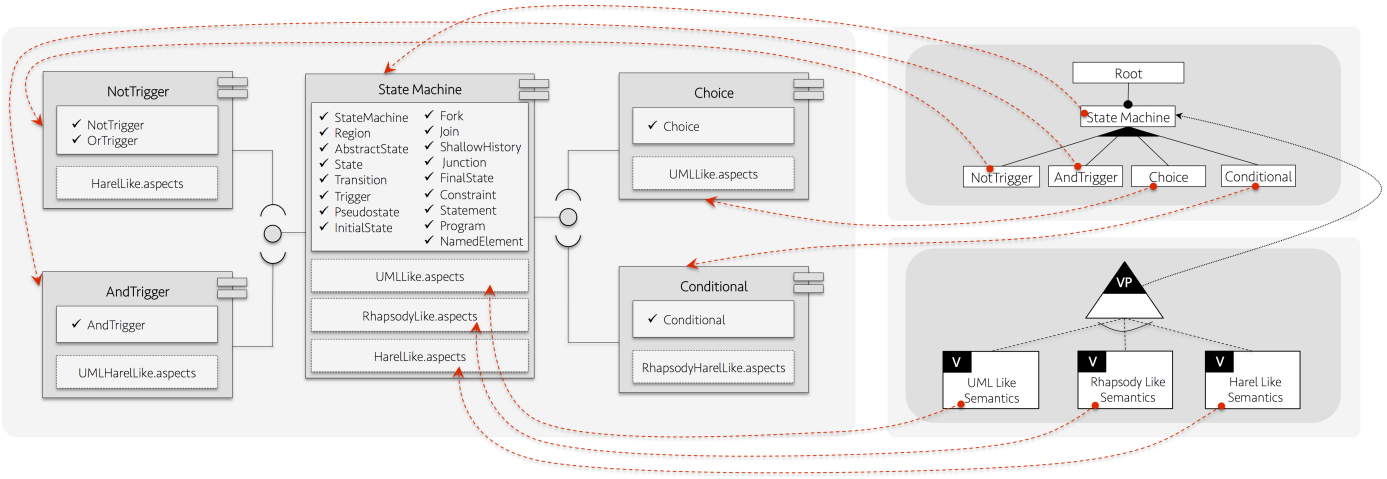

Figure 14: Language product line produced for the case study of the finite state machines.

process starts by breaking down the DSLs into a set of reusable language modules. Then, the process goes through the synthesis of the models that capture the variability existing among the extracted language modules.

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] 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

[4] 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

[5] 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

[6] M. Funk, M. Rauterberg, PULP Scription: A DSL for Mobile HTML5 Game Applications, ICEC 2012, Springer Berlin Heidelberg, Bremen, Germany, 2012, pp. 504–510. doi:10.1007/978-3-642-33542-6_65

[7] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, J.-P. Tolvanen, Dsls: The good, the bad, and the ugly, in: Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA Companion '08, ACM, New York, NY, USA, 2008, pp. 791–794. doi:10.1145/1449814.1449863
URL http://doi.acm.org/10.1145/1449814.1449863

[8] M. Homer, T. Jones, J. Noble, K. B. Bruce, A. P. Black, Graceful Dialects, ECOOP 2014, Springer Berlin Heidelberg, Uppsala, Sweden, 2014, pp. 131–156. doi:10.1007/978-3-662-44202-9_6

[9] P. James, M. Roggenbach, Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans, Mathematics in Computer Science 8 (1) (2014) 11–38. doi:10.1007/s11786-014-0174-0

[10] A. Iliasov, I. Lopatkin, A. Romanovsky, The SafeCap Platform for Modelling Railway Safety and Capacity, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 130–137. doi:10.1007/978-3-642-40793-2_12
URL http://dx.doi.org/10.1007/978-3-642-40793-2_12

[11] 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

[12] 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.

[13] 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

[14] 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.

[15] 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.

[16] J. Mayrand, C. Leblanc, E. M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: Software Maintenance 1996, Proceedings., International Conference on, 1996, pp. 244–253. doi:10.1109/ICSM.1996.565012

[17] J. Rubin, K. Czarnecki, M. Chechik, Cloned product variants: from ad-hoc to managed software product lines, International Journal on Software Tools for Technology Transfer 17 (5) (2015) 627–646. doi:10.1007/s10009-014-0347-9
URL http://dx.doi.org/10.1007/s10009-014-0347-9

[18] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, Reverse-engineering reusable language modules from legacy domain-specific languages, in: Proceedings of the International Conference on Software Reuse, ICSR 2016, Springer, Limassol, Cyprus, 2016.

[19] O. M. G. (OMG), Uml 2.4.1 superstructure specification (2011).

[20] D. Harel, H. Kugler, The rhapsody semantics of statecharts (or, on the executable core of the uml), in: H. Ehrig, W. Damm, J. Desel, M. Groe-Rhode, W. Reif, E. Schnieder, E. Westkmper (Eds.), Integration of Software Specification Techniques for Applications in Engineering, Vol. 3147 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 325–354. doi:10.1007/978-3-540-27863-4_19
URL http://dx.doi.org/10.1007/978-3-540-27863-4_19

[21] N. Martaj, M. Mokhtari, Stateflow, in: MATLAB R2009, SIMULINK et STATEFLOW pour Ingénieurs, Chercheurs et Étudiants, Springer Berlin Heidelberg, 2010, pp. 513–586. doi:10.1007/978-3-642-11764-0_13
URL http://dx.doi.org/10.1007/978-3-642-11764-0_13

[22] D. Harel, A. Naamad, The statemate semantics of statecharts, ACM Trans. Softw. Eng. Methodol. 5 (4) (1996) 293–333. doi:10.1145/235321.235322
URL http://doi.acm.org/10.1145/235321.235322

[23] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, A. Egyed, An assessment of search-based techniques for reverse engineering feature models, Journal of Systems and Software 103 (2015) 353 – 369. doi:http://dx.doi.org/10.1016/j.jss.2014.10.037
URL http://www.sciencedirect.com/science/article/pii/S0164121214002349

[24] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Bottom-up adoption of software product lines: a generic and extensible approach, in: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, 2015, pp. 101–110. doi:10.1145/2791060.2791086
URL http://doi.acm.org/10.1145/2791060.2791086

[25] J. Martinez, T. Ziadi, T. F. Bissyand, J. Klein, Y. l. Traon, Automating the extraction of model-based software product lines from model variants (t), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, 2015, pp. 396–406. doi:10.1109/ASE.2015.44

[26] B. Combemale, C. Hardebolle, C. Jacquet, F. Boulanger, B. Baudry, Bridging the chasm between executable metamodeling and models of computation, in: Proceedings of the International Conference on Software Language Engineering, SLE 2012, Springer, Dresden, Germany, 2013, pp. 184–203.

[27] 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.

[28] 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

[29] T. Gschwind, Automated Adaptation of Component Interfaces with Type Based Adaptation, Springer London, London, 2012, pp. 45–61. doi:10.1007/978-1-4471-2350-7_5

[30] J. Steel, J.-M. Jézéquel, On model typing, Software & Systems Modeling 6 (4) (2007) 401–413. doi:10.1007/s10270-006-0036-6
URL http://dx.doi.org/10.1007/s10270-006-0036-6

[31] 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

[32] C. Guy, B. Combemale, S. Derrien, J. R. H. Steel, J.-M. Jézéquel, On model subtyping, in: A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, D. Kolovos (Eds.), Proceedings of the 8th European Conference on Modelling Foundations and Applications, ECMFA 2012, Springer Berlin Heidelberg, Lyngby, Denmark, 2012, pp. 400–415. doi:10.1007/978-3-642-31491-9_30

[33] 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

[34] 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

[35] M. Crane, J. Dingel, Uml vs. classical vs. rhapsody statecharts: not all models are created equal, Software & Systems Modeling 6 (4). doi:10.1007/s10270-006-0042-8

[36] 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

[37] 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

[38] 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

[39] 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.

[40] D. Lucanu, V. Rusu, Program equivalence by circular reasoning, in: Proceedings of the International Conference on Integrated Formal Methods, IFM 2013, Springer, Turku, Finland, 2013, pp. 362–377.

[41] B. Biegel, S. Diehl, Jccd: A flexible and extensible api for implementing custom code clone detectors, in: Proceedings of the International Conference on Automated Software Engineering, ASE 2010, ACM, Antwerp, Belgium, 2010, pp. 167–168.

[42] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth, DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013.

[43] W. K. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Extracting variability-safe feature models from source code dependencies in system variants, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, ACM, New York, NY, USA, 2015, pp. 1303–1310. doi:10.1145/2739480.2754720
URL http://doi.acm.org/10.1145/2739480.2754720

[44] S. She, U. Ryssel, N. Andersen, A. Wasowski, K. Czarnecki, Efficient synthesis of feature models, Information and Software Technology 56 (9) (2014) 1122 – 1143, special Sections from Asia-Pacific Software Engineering Conference (APSEC), 2012 and Software Product Line conference (SPLC), 2012. doi:http://dx.doi.org/10.1016/j.infsof.2014.01.012

[45] E. Vacchi, W. Cazzola, S. Pillay, B. Combemale, Variability Support in Domain-Specific Language Development, SLE 2013, Springer International Publishing, Indianapolis, IN, USA, 2013, pp. 76–95. doi:10.1007/978-3-319-02654-1_5

[46] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231 – 274. doi:http://dx.doi.org/10.1016/0167-6423(87)90035-9