Reverse Engineering Language Product Lines from Existing DSL Variants

David A Méndez-Acuña, José A Galindo, Benoit Combemale, Arnaud Blouin, Benoit Baudry

Reverse Engineering Language Product Lines from Existing DSL Variants

David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, Benoît Baudry

INRIA/IRISA and University of Rennes 1
Campus de Beaulieu, Rennes, France

Abstract

The use of domain-specific languages (DSLs) has become a successful technique to develop complex systems. In this context, an emerging phenomenon is the existence of DSL variants, which are different versions of a DSL adapted to specific purposes but that still share commonalities. In such a case, the challenge for language designers is to reuse, as much as possible, previously defined language constructs to narrow implementation from scratch. To overcome this challenge, recent research in software languages engineering introduced the the notion of language product lines. Similarly to software product lines, language product lines are often built from a set of existing DSL variants.

In this article, we propose a reverse-engineering technique to ease-off such a development scenario. Our approach receives a set of DSL variants which are used to automatically recover a language modular design and to synthesize the corresponding variability models. The validation is performed in a project involving industrial partners that required three different variants of a DSL for finite state machines. This validation shows that our approach is able to correctly identify commonalities and variability.

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 software is designed and implemented [1]. The use of domain-specific languages (DSLs) has emerged in response to this need as an alternative to express software solutions in relevant domain concepts, thus favoring separation of concerns and hiding fine-grained implementation details [2]. DSLs are software languages whose expressiveness is focused on 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 motivated the construction of 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 mobile applications' 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 and technical knowledge to conduct the definition of complex artifacts such as metamodels, grammars, interpreters, or compilers [2].

The development of DSLs becomes more complex when we consider that DSLs often have many *variants*. A variant is a new version of a given DSL that introduces certain differences in terms of syntax and/or semantics [8]. Typically, language variants appear under two situations. The first situation is the use of well-known formalisms through different domains. Consider the case of finite state machines, which have been used in a the construction of DSLs for a large spectrum of domains such as definition of graphical user interfaces [4] or games prototyping [9]. Those DSLs share typical state machine concepts such as states or transitions. However, each DSL adapts those abstractions to address the particularities of its domain.

The second situation that favors the existence of DSL variants is when the complexity of a given domain requires the construction of several DSLs with different purposes. In such a case, the domain abstractions of the DSLs are similar, but their concrete implementations require adaptations. For instance, suppose two DSLs: the former is a DSL for specification and verification of railway scheme plans [10]; the latter is a DSL for modeling and reasoning on railway systems' capacity [11]. These DSL share certain domain abstractions —i.e., railway management—. However, they both require different semantics and specialized constructs to achieve their purposes.

The phenomenon of DSL variants is not a problem itself but reflects the abstraction power of certain well-known formalisms —such as state machines or petri nets— that, with proper adaptations, can fit various domains. Besides, it shows how different issues in a same domain can be addressed by diverse and complementary DSLs. Nevertheless, 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. After all, at implementation level each DSL variant is a complete language itself

Email addresses: damenac@gmail.com (David Méndez-Acuña), jagalindo@inria.fr (José A. Galindo), benoit.combemale@inria.fr (Benoît Combemale), arnaud.blouin@inria.fr (Arnaud Blouin), benoit.baudry@inria.fr (Benoît Baudry)

requiring tooling such as editors, interpreters, compilers, and so on.

In this context, the challenge for language designers is to take advantage of the commonalities existing among DSL variants by reusing, as much as possible, formerly defined language constructs [12]. The objective is to leverage previous engineering efforts to minimize implementation from scratch. To achieve such a challenge, the research community in software language engineering has proposed the use of Software Product Line Engineering (SPLE) in the construction of DSLs [13, 14]. This led to the notion of *Language Product Line Engineering (LPLE)* —i.e., the construction of software product lines where the products are languages [12, 15]—.

Similarly to software product lines, language product lines can be built from a set of existing DSL variants through reverse-engineering techniques [16]. Those techniques should provide mechanisms for: (1) recovering of a language modular design including all the language constructs existing in the DSL variants; and (2) synthesis of the corresponding variability models.

In a previous work [17, 18] we introduced an approach to automatically infer a language modular design from a given set of DSL variants. In this article we extend that work to provide a complete reverse engineering technique that produces not only the language modular design, but the entire language product line. In that sense, the delta of this article with respect to the previous one is the synthesis of the variability models specified in terms of well-known formalisms —i.e., feature models (FM) and orthogonal variability models (OVM)—. Those models encode the variability of a language product line in a compact way while considering the diverse dimensions that such a variability may present. We also show how language variability models can be used to configure and assembly new DSL variants.

We validate our approach within an industrial project which is composed of three variants of a DSL for finite-state machines [19]. In that project we manually developed an oracle to know in advance the existing variation points. Then, we execute our approach on these DSL variants and we compare the produced results against the expected ones. The result of this comparison shows that our reverse engineering technique is correct since all the detected variation points correspond to real differences in the DSL variants. Also, this validation allowed us to identify certain threats to validity regarding the level of granularity of the detected variation points.

The remainder of this article is structured as follows: Section 2 describes the problem statement. Section 3 introduces our approach. Section 4 presents the experiments executed in the context of the VaryMDE project that are used as a validation. Section 5 discusses the related work. Finally, Section 6 concludes the article.

2. Problem Statement

Similarly to software product lines [20], the development process of language product lines can be divided into two phases: domain engineering and application engineering. During the domain engineering phase, language designers build the language product line. This process includes the design and implementation of a set of interdependent language modules that implement the language features and the construction of variability models encoding the rules in which those features can be combined to produce valid DSL variants. During the application engineering phase, diverse DSL variants are derived according to specific needs. Such a derivation process comprises the selection of the features to include in a given DSL variant —i.e., language configuration— and the assembly of the corresponding language modules —i.e., language modules composition—.

Note that in bottom-up language product lines the domain engineering phase is performed through reverse-engineering techniques [16]. Indeed, in such a case the starting point of the development process is a set of DSLs variants that are built along a language engineering project while addressing different requirements and application scenarios. At some point, language designers realize that there is potential reuse among the variants. Hence, they launch a reverse-engineering process where the existing DSL variants are used to build up a language product line. Using this language product line, language designers can go through the application engineering phase in order to create new DSL variants.

2.1. Motivating scenario

Suppose a team of language designers working on the construction of the DSL for finite state machines. Initially, language designers followed the UML specification [21] to define language constructs such as states, regions, transitions, and triggers. Those language constructs are specified in terms of their syntax and semantics. So, at the end of the language development process, language designers release an executable DSL whose behavior complies the UML specification.

Once this first DSL was released, the language designers are asked to build a new variant which must comply the Rhapsody specification [22] —i.e., another formalism to finite state machines—. This new variant shares many commonalities with UML state machines, but introduces differences at both syntax and semantics levels [19]. After building this second variant, language designers obtained two different DSLs implementing different formalisms of state machines. Those DSL variants have some commonalities among them. And at the same time, the DSL have some particularities that make them unique.

Note that this process is repeated each time language designers have to build a new variant of the DSL for each new FSM formalism —e.g., Stateflow [23] or Harel state machines [24]—. This becomes specially challenging when final users need to combine some specifications to define hybrid formalisms. While several approaches have been proposed to reverse engineering software product lines from existing product variants [25, 26, 27], in this article we propose techniques to reverse engineering language product lines from existing DSL variants. In that sense, our work can be compared with approaches such as the ones presented by Kühn et al. [15] and by Vacchi et al. [28, 29].

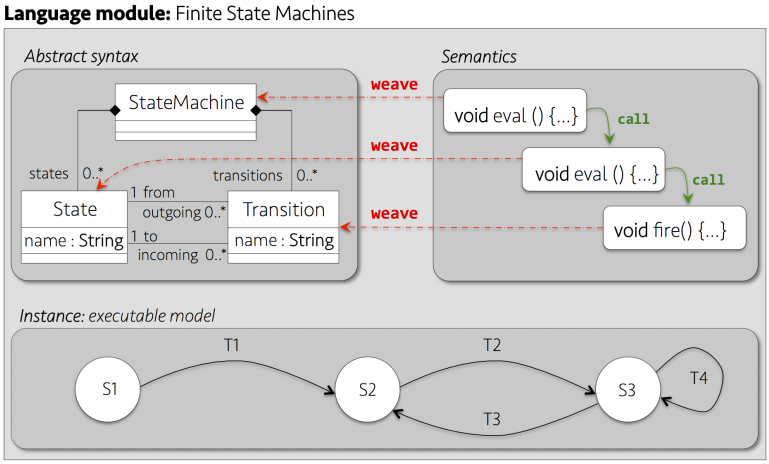

Figure 1: A simple DSL for finite state machines

2.2. Scope: *Executable Domain Specific Languages*

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* [30]. Whereas metamodels are class diagrams that represent language constructs and relationships among them, domain specific actions are Java-like methods that introduce behavior in the metaclasses of a given metamodel [31].

Fig. 1 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. 1 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 [30].

3. Proposed Approach: *Reverse-Engineering Language Product Lines*

In this section, we present our reverse engineering technique to support the construction of bottom-up language product lines. As shown in Fig. 2, the proposed technique is composed of four steps. During the first step, we automatically recover a language modular design for the language product line. Such a modular design is composed of a set of language modules and a set of dependencies among them. During the second step, language modules' dependencies are used to synthesize a variability model that can be used, during the third step, to configure concrete DSL variants. Finally, during the forth step the DSL variant is assembled by composing the involved language modules.

3.1. Recovering a Language Modular Design

Let us start the description of our reverse engineering technique by explaining the way in which we identify the set of language modules and dependencies that constitute the language modular design of the product line. The details of this recovering process are explained below as well as the way in which language modules are specified to guarantee their composability.

3.1.1. Language Modules. *How to identify them?*

To identify the language modules of a language product line, we define some comparison operators that facilitate the identification of language constructs replicated in the DSL variants. These operators take into account both syntax and semantics of language constructs. Then, we extract replicated constructs into interdependent language modules whose dependencies are expressed through interfaces guaranteeing that those language modules can be later assembled among them. Such a strategy to extract reusable language modules is based on four principles explained in the following:

Principle 1: *DSL specifications are comparable. So, replicated language constructs can be automatically detected.* To detect replicated language constructs in a given set of DSL variants, we need to define some criteria to compare the DSL specifications to decide when a language construct is equal to another. Whithin the scope of this article, a language construct is defined in terms of a metaclass and a set of domain specific actions.

Comparison of metaclasses. To compare metaclasses, we need to take into account that a metaclass is specified by a name, a set of attributes, and a set of references to other metaclasses. Two metaclasses A and B are considered as equal if all those elements match i.e., their names are equal; for all attributes in A there exists an equivalent attribute in B; and for each reference in A there exists an equivalent reference in B. Formally, comparison of metaclasses is formalized by the operator $\div$.

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

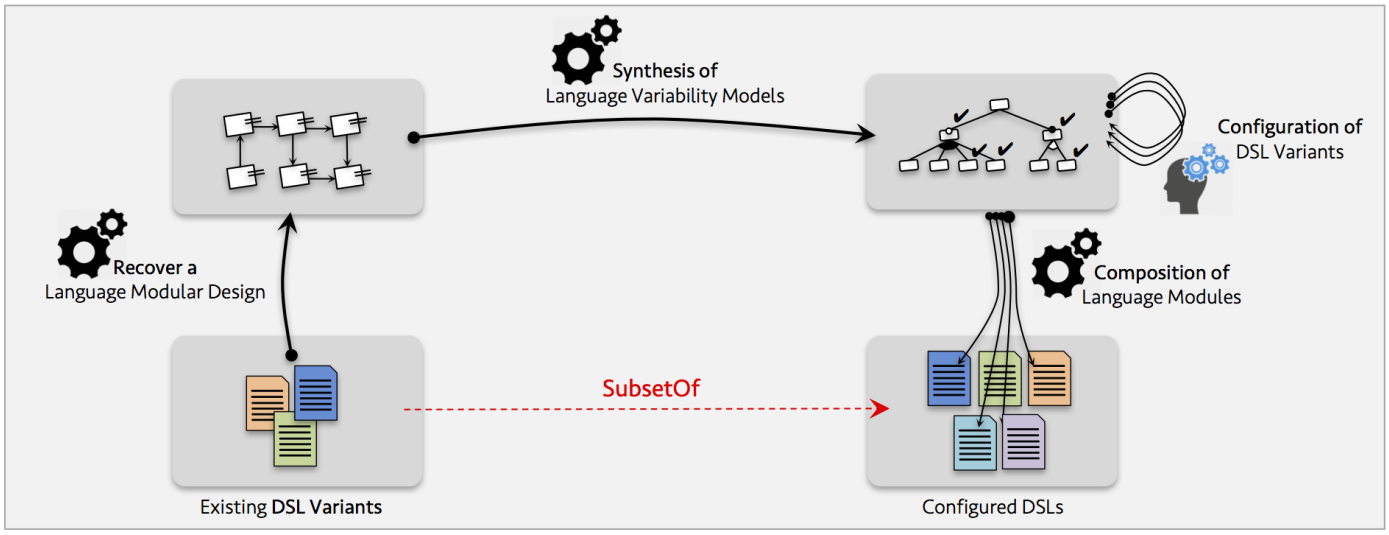

Figure 2: Reverse engineering language product lines: approach overview

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

Comparison of domain-specific actions. To compare domain specific actions we need to consider that —similarly to methods in Java— domain specific actions have a signature that specifies their contract and a body where the behavior is implemented. Two domain specific actions are equal if their signatures and bodies are equivalent.

Whereas comparison of signatures can be performed by syntactic comparison of the signature elements —i.e., checking if the names, return types, visibility rules—, 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 [32]. To address this issue, we conceive bodies comparison in terms of its abstract syntax tree as proposed by Biegel et al. [33]. In other words, to compare two bodies, we first parse them to extract their abstract syntax tree, and then we compare those trees. Formally, comparison of domain-specific actions (DSAs) is specified by the operator $\fallingdotseq$.

$$
\fallingdotseq \; : DSA \times DSA \rightarrow bool \tag{3}
$$

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

Note that these comparison operators are structural for both syntax and semantics of language constructs. They result useful when the DSL variants were built-up by using practices such as clone-and-clone. To enhance the scope of the approach, other comparison operators that take into accoung not only the structure of the constructs but their runtime behavior can be introduced. The article presented by Bousse et. al. [34] presents some ideas in that direction.

Principle 2: *Replicated constructs can be viewed as sets' intersections, which is useful to factorization.* A DSL specification can be seen as a set of metaclasses and a set of domain specific actions. In doing so, replicated constructs correspond to intersections among those sets. Those intersection elements can be specified once and reused in several DSL variants [35, p. 60-61]. Hence, we can factorize replication constructs by breaking down the intersections existing among DSL specifications.

Fig. 3 illustrates this observation through the running example introduced in Section 2. At the left of the figure, we show two Venn diagrams to represent both syntax and semantic intersections. The Venn diagram corresponding to the abstract syntax shows that the classical constructs for state machines such as StateMachine, State, and Transition are in the intersection of the three given DSL variants i.e., UML state machines, Rhapsody, and Harel's state machines. In turn, there are certain particularities for each DSL. For example, the concept AndTrigger is owned by UML and Harel state machines but not for Rhapsody. Concepts such as OrTrigger and NotTrigger are only provided by Harel state machines since the concept of Choice is exclusive of UML state machines.

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

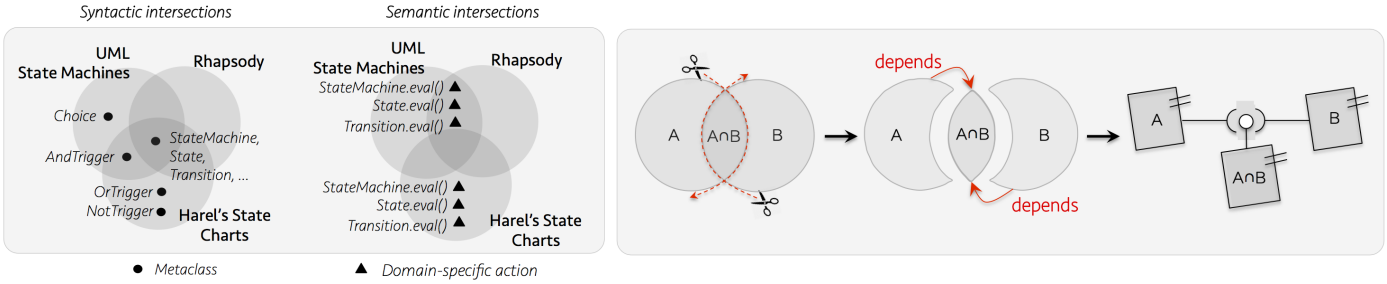

Figure 3: Factorizing replicated language constructs from DSL variants

This way to conceive DSL specifications is useful to factorize replicated language constructs as illustrated at the right of Fig. 3. Each different intersection is separated in a separate subset that, as we will explain later, is encapsulated in a language module.

Principle 3: *Abstract syntax first, semantics afterwards.* The abstract syntax is the backbone of the DSL specification; it specifies its structure in terms of metaclasses and relationships among them whereas the domain-specific actions add executability to the metaclasses. Hence, the process of breaking down intersections should be performed for the abstract syntax first, thus identifying the way in which metaclasses should be grouped into the different language modules. Afterwards, we can do the proper for the semantics. In doing so, we need to take into consideration the phenomenon of semantic variability. That is, two replicated metaclasses might have different domain-specific actions. That occurs when two DSLs share some syntax specification but differ in their semantics.

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

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

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

The principles in action. Fig. 4 shows the way in which we recover a language modular design through the principles explained above. It is composed of two steps: unification and breaking down.

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

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

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

3.1.2. *Language Modules.* ***How to specify them?***

We have explained the way in which we recover a language modular design by identifying clusters of language constructs and dependencies among them. However, it is unclear how to specify those clusters in concrete implementation artifacts that can be later composed. To deal with this issue, we propose to specify a language module in terms of (1) a metamodel containing the metaclasses corresponding to each construct cluster; and (2) a set of domain specific actions implementing the semantics of their metaclasses —see Fig. 5—. If there is semantic variability, then a language module can have several clusters of domain specific actions. The dependencies among language modules are materialized through required and provided interfaces.

Required interfaces. A required interface is a mechanism 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, those needs can

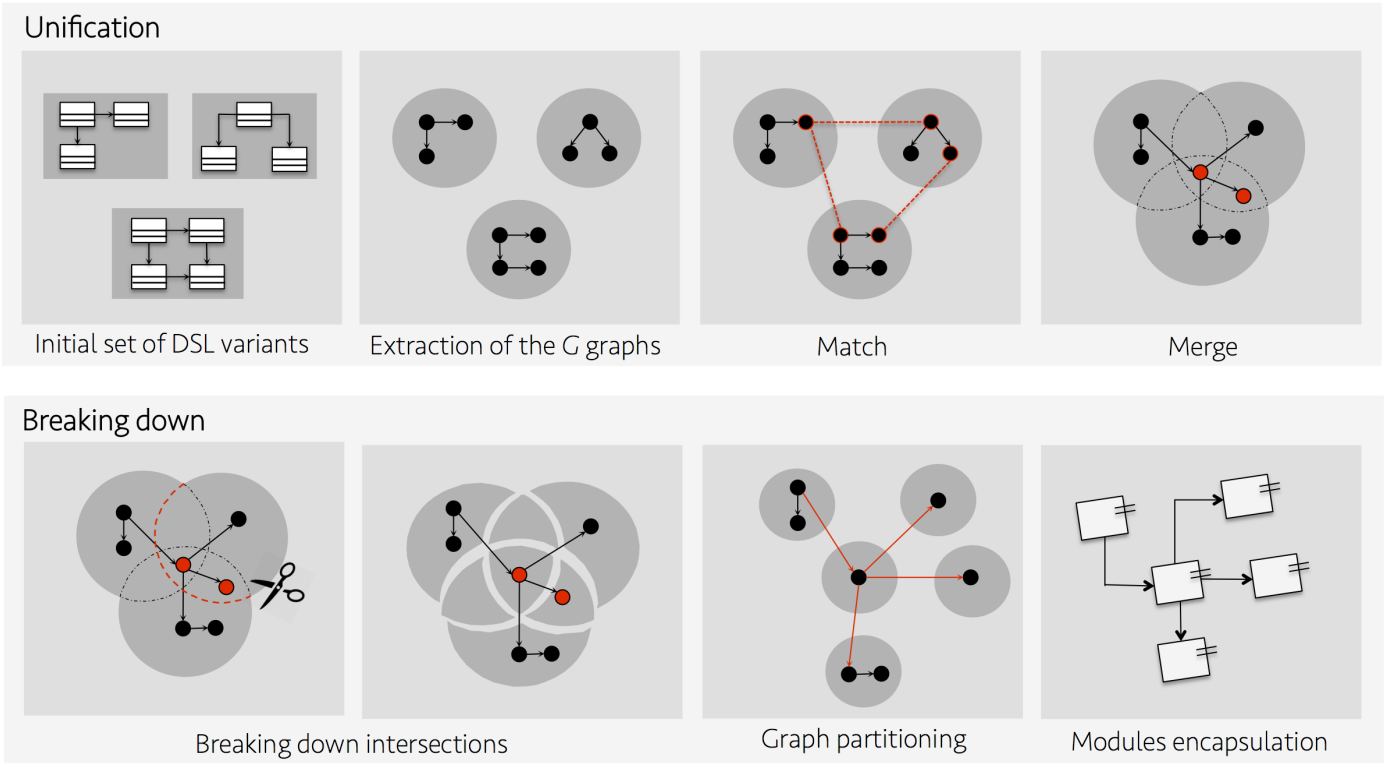

Figure 4: Unifying and breaking down for recovering a language modular design

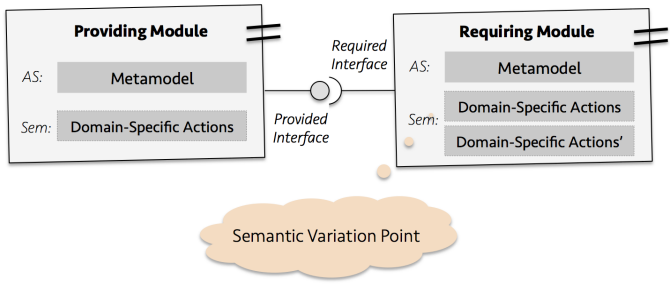

Figure 5: Specification of language modules

be declared as a set of required constructs —e.g., *Constraint*—.

We propose a mechanism to distinguish whether a given language specification element —i.e., meta-class, property, operation, parameter, enumeration, etc— corresponds to an actual implementation or a required declaration. The proposed mechanism is an extension to the EMOF meta-language that introduces the notion of "requirement", so we can define *required specification elements* in metamodels. When encapsulating clusters of concepts in language modules, all the constructs contained in the cluster are defined as actual implementations. In turn, all the references to specification elements that belong to other clusters are defined as required specification elements, so they are included in the required interface.

Provided interfaces. The purpose of provided interfaces is to expose the functionality offered by the language module. 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.

When encapsulating a language module from a cluster of constructs, all the constructs that are used by external clusters are defined as public.

3.2. Synthesizing Language Variability Models

Once we have recovered a language modular design for the language product line, we need to represent the existing variability in a model that permits to configure concrete DSLs. To this end, we need to find out an appropriated formalism to express that model, and then to conceive a strategy to synthesize

those models from the language modular design.

3.2.1. Language Variability. ***How to express it?***

The challenge towards representing the variability existing in a language product line is that such variability is multi dimensional. Because the specification of a DSL involves several implementation concerns[1], then there are several dimensions of variability that we must manage: abstract syntax variability, concrete syntax variability, and semantic variability [37, 38].

Abstract syntax variability refers to the capability of selecting the desired language constructs for a particular type of user. 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. As the same as our approach to language modularization, our approach to variability management is scoped to abstract syntax and semantics; concrete syntax —and hence, concrete syntax variability— is not being considered in the solution.

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

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

Currently, we can find several approaches to support multi dimensional variability —e.g., [39]—. Some of those approaches have been applied concretely to language product lines [40]. The most common practice is to use feature models to represent all the dimensions of variability. Each dimension is specified in a different tree and dependencies among decisions in those dimensions are expressed as cross-tree constraints. In this article, we propose a different approach based on the combination of feature models with orthogonal variability models. Feature models are used to model abstract syntax variability and orthogonal variability models are used to model semantic variability.

Fig. 6 illustrates our approach. At the top of the figure, there is a feature model in which each feature represents a language module. As aforementioned, each language module is composed of a metamodel and a set of domain specific actions. Hence, such a feature model is enough for language product lines where there is not semantic variability i.e., each language module has only one set of domain specific actions. Differently, when there are one or more language modules containing several sets of domain specific actions, then we have semantic variability that must be represented in the variability model. To represent such a variability, we include an orthogonal variability model as illustrated at the bottom of Fig. 6 which contains a variation point for each feature that represents a language module with more than one set of domain specific actions.

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

(1) The structure of orthogonal variability models is more appropriated. As explained by Roos-Frantz et al. [41], feature models and orthogonal variability models are similar. However, they have some structural differences. One of those differences is that whereas a feature model is a tree that can have many levels, an orthogonal variability model is a set of trees each of which has two levels. Each tree represent one variability point and its children represent variants to that variation point.

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

(2) The meaning of orthogonal variability models is more appropriated. According to [40], a language feature is a characteristic provided by the language which is visible to the final user. This definition can be associated to the abstract syntax variability and the use of feature models can be appropriated to represent it. All the approaches on language product line engineering use feature models to this end showing that it is possible and appropriated.

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

3.2.2. Language Variability. ***How to synthesize it?***

Once we established an approach to represent language variability, we define an algorithm to synthesize variability models

[1] Just as traditional general purpose languages, domain specific languages are typically defined through three implementation concerns: abstract syntax, concrete syntax, and semantics [36].

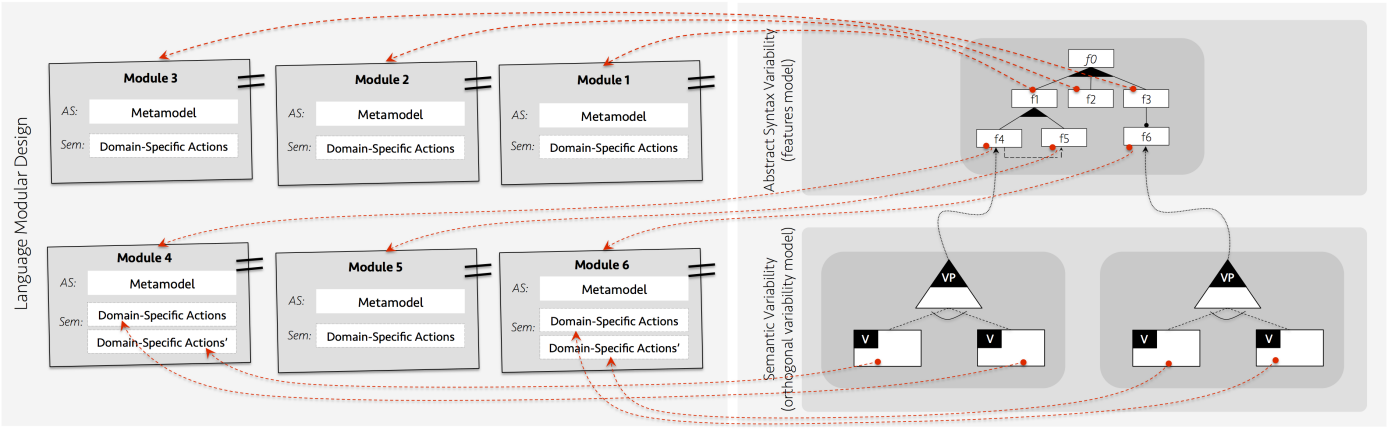

Figure 6: Approach to represent multi-dimensional variability in language product lines

from a given language modular design. This algorithm produces not only a feature model with the abstract syntax variability, but also an orthogonal variability model representing the semantic variability. An overview of the approach is presented in Fig. 7.

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

In the literature, there are several approaches for reverse engineering feature models from dependencies graphs —consider for example the approach presented by Assunçao et al. [42], or the one presented by She et al., [43]—. 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. [28] which fulfills the aforementioned requirements. In particular, we re-use the logical strategy they propose to build a features model from a set of language components.

The tooling that supports our algorithms is flexible enough to permit the use of other approaches for synthesis of feature model. To this end, we provide an extension point that language designers can use to add new synthesis algorithms. In addition to the one proposed by Vacchi et al. [28], we have integrated our approach with the one provided by Assunçao et al., [42].

Synthesizing semantic variability. Once the feature model encoding abstract syntax variability is produced, we proceed to do the proper with the orthogonal variability model encoding semantic variability. To this end, we need to analyze the results of the process for extracting the language modules. As explained in Section 3.1, 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. For each one, we verify if it has more than one cluster of domain specific actions. If so, we create a semantic variation point where each variation references one cluster. Finally, the semantic variation point is associated with the feature that represents the language module owning the clusters.

3.2.3. DSL Variants Configuration

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

Multi-staged configuration was introduced by Czarnecki et al. [44] for the general case of software product lines, and discussed by Dinkelaker et al. [45] 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 [45]. 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. 8 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 avail-

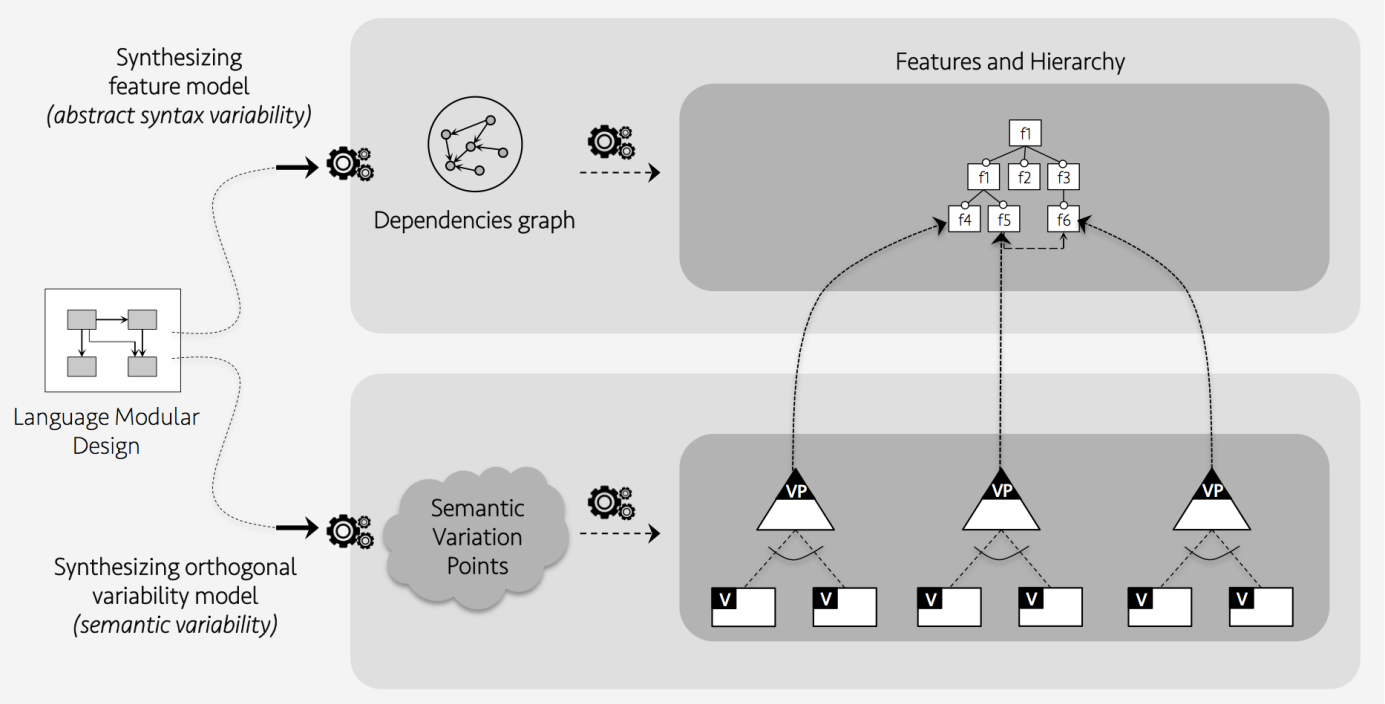

Figure 7: Reverse-engineering variability models for language product lines

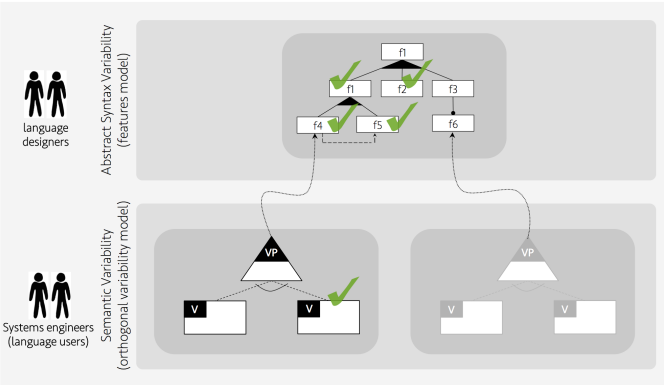

Figure 8: Approach to support multi-staged configuration of language product lines

able 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.

3.3. Language Modules Composition

The final step of the language product line engineering process is the composition of the language modules corresponding to the configuration indicated in the variability models. This composition process starts by checking the compatibility between the required and provided interfaces of the involved modules. Then, the specification of the modules are merged to produce a complete DSL variant.

Compatibility checking. To establish a compatibility checking mechanism between provided and required interfaces, we need to conciliate two different —and potentially conflicting— issues. Firstly, such a compatibility checking must guarantee safe composition of the involved language modules. Hence, we need to verify that the functionality offered by the provided interface actually fulfills the needs of required interface. Secondly, there must be some place for substitutability. Hence, 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 [46].

To deal with the aforementioned issues, we propose an approach for compatibility checking which is at the same time strict enough to guarantee safe composition, and flexible enough to permit substitutability under certain conditions. We extract both required and provided interfaces in the form of *model types* [47]. The model type corresponding to the required interface contains the required specification elements of a language module whereas the model type corresponding to the provided interface the model type contains its public specification elements [2].

[2]The relationship between a model type and a language module is called ***implements*** and it is introduced by Degueule et al. [48].

Then, we perform compatibility checking by checking the subtyping relationship —introduced by Guy et al. [49]— 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.

Merging modules' specifications. The process of merging the specification of a set of language modules is performed in two phases. First, there is a matching process that identifies one-to-one matches between required 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. 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.

4. Validation: The VaryMDE Project

In this section, we introduce the VaryMDE project which is bilateral collaboration between INRIA and Thales Research & Technology (TRT). The role of this project in the research presented in this article is two-fold. On one hand, it provides a set of research questions that motivate our work. Concretely, the research presented in this article represents an answer to some of the needs emerging in Thales in terms of language engineering. On the other hand, this project provides a realistic application scenario which we used as validation of the approach. In the reminder of this section, we present an overview of the project and we discuss the results of applying our approach in the scenario introduced by Thales. To explain this scenario we try to follow as close as possible the guidelines provided by [50] for the sake of clarity and organization.

4.1. Background to the research project

Thales is a company whose business model turns around the construction of different types of systems that solve needs in diverse domains such as transport, aerospace, security, or defense. During the construction of these systems, Thales' engineers often appeal to the use of state machine languages to express behavior.

Despite the expressibility of state machines, the diversity of the systems built by Thales imposes an accidental complexity. Depending on the type of system under construction, there are different requirements on the way in which a state machine should be executed. Hence, there is certain semantic flexibility that state machine languages should offer to support the particularities of the systems unde construction. As a result, Thales engineers are intended to build not only the devices themselves but also to adapt the state machines formalisms.

The typical development process to address the implementation of the formalisms for state machines is as follows: at the beginning, language designers build an initial DSL for state machines that fits the needs of one type of system. Then, they create new development branches when they adapt the first variant of the DSL to the needs of other types of systems. After some repetitions, language designers have a family of DSLs for state machines. Those DSLs have both syntax and semantic differences.

4.2. Design of the experiment

Objectives. One of the challenges of the VaryMDE project is to find out a way to facilitate the development process descrived above. As an answer of this challenge, we propose the use of reverse engineering techniques to create language product lines of DSLs for state machines from existing variants.

Planning. The experiment was planned in three phases as shown in Fig. 9. In the first phase, the data set is prepared. Such a data set corresponds to the set of DSL variants that will be used to reverse engineer the language product line. This phase is iterative since we need to consider the feedback comming from the Thales' engineers. During the second phase, we execute our approach using the initial set of DSL variants. Finally, in the third phase we analyse the produced language product lines.

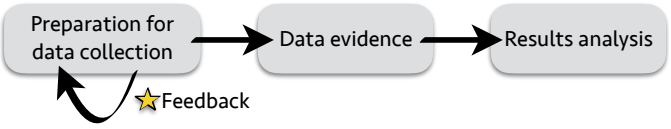

Figure 9: Project planning

4.3. Preparation for data collection

As aformentioned, the objective in this phase is to define the set of initial DSL variants that will be used to execute our approach. The most important limitation we found at this stage is that many of the implementations for state machine DSLs are built in different language workbenches and using diverse language meta-languages. Under these conditions, commonalities and particularities of DSLs are more difficult to detect.

To overcome such a limitation, we decided to implement the initial DSL variants in a unified language workbench and using the same meta-languages. Hence, the phase of preparation for data collection corresponds to a language development process where three different formalisms for state machiens were implemented: UML state machines, Rhapsody, and Harel statecharts. Those formalisms were selected because they have a complete documentation that allow to fully understand its semantics. The implementation of the formalisms was conducted on top of Melange [48] language workbench and it is avail-

able on a dedicated GitHub repository[3]. The description of commonalities and differences existing among the selected formalisms are well-studied by Crane et al and are described in Annexe A.

4.4. Collecting evidence

Once we built the initial set of DSLs variants, we execute our approach and obtained a language product line for state machines. The results are summarized in Fig. 10. 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 3.1 on breaking down intersections. At the right of the figure we show the corresponding variability models. Each feature of the feature models is associated to a given language module. In turn, the semantic variability points in the orthogonal model are associated to clusters of domain specific actions.

4.5. Analysis of collected data

Let us now discuss the results of the project regarding. As expected, we obtained a language product product line from a set of DSL variants for finite state machines. But... Does this product line identify all the variation points and commonalities existing in the DSL variants? Are those variation points properly specified in the language modular design and variability models? Since we know these variation points and commonalities, we can check whether they are appear in the produced language product line. The results of this verification are presented in Table 1.

The results are promising in the case of abstract syntax variability. According to the Table **??**, the DSL variants share 17 constructs in common. Those constructs are properly factorized in a language module that we named StateMachine. This module is correctly identified during the recovering of the language modular design, and it is properly specified as a language module in terms of a metamodel enhance with domain specific actions and offering a provided interface. Besides, the particularities of the DSL variants are also well factorized. There is a module that contains the constructs NotTrigger and OrTrigger that belong only to the variant complying the Harel' statecharts specification. Besides, there are three additional modules that contain the constructs AndTrigger, Choice, and Conditional respectively. Using this modular design, we can re-compose any of the three initial DSL variants.

The situation is different for the case of semantic variability. Although our reverse-engineering strategy is able to identify that the domain specific actions are different in the three DSL variants, the level of granularity at which those variation points are detected is coarser than one might expect. At the beginning of this section, we described three semantic variation points and their possible interpretations i.e., events dispatching policy, execution order of transitions' effects, and priorities of conflicting transitions. Using the proposed technique, we can identify just one semantic variation point indicating that the language module called StateMachines contains three different clusters of domain specific actions, which is reflected in the orthogonal variability model.

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

Table 1: Analysis of the results

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

5. Related Work

The idea of reverse engineering software product lines from product variants has been already studied in the literature. Besides, there are several approaches that address this issue for the case in which the product variants have been built using the clone-and-own approach [25, 26, 27]. Although the applicability of such idea to the specific case of language product lines is quite recent, there are some related work that we discuss in this section.

Recovering a language modular design. The first challenge during reverse engineering language of product lines is to recover a language modular design. Although this challenge has not received proper attention, we found an approach that proposes insightful advances in this direction [15]. In that work, the language modular design is achieved by defining one language module for each construct. That means that the reverse engineering process will result in a language product line containing as many features as constructs exist in the DSLs. The

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

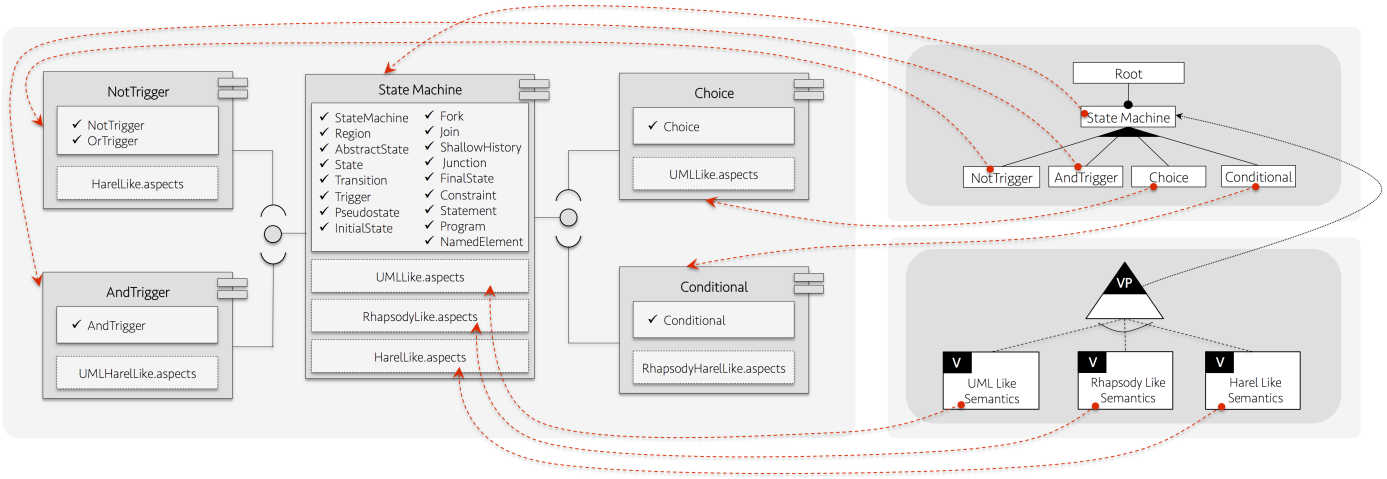

Figure 10: Language product line produced for the VaryMDE project.

approach relies on Neverlang [51] and AiDE [52] as tooling for language modularization and variability management respectively.

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

Differently, in our approach we use the notion of specification clones and intersections in order to achieve a level of granularity that captures the variability existing in the DSL variants given in the input. This permits to identify those clusters of language constructs that go always together in the given variants. This decision simplifies the language product line in the sense that the amount of language modules is lower than in the approach by Kuhn et al., [15]. In doing so, we certainly reduce the possible variants that can be configured by the language product line. This issue can be considered as a threat to validity of our approach.

Synthesizing variability models. The synthesis of variability models has been largely studied in the literature. Some of those approaches have been adapted for the particular case of variability in the context of language product lines engineering. The approach presented in [29] proposes a search-based technique to find a features model that represents the variability existing in a set of language modules while optimizing an objective function. This approach uses an ontology that describes the domain concepts of the language product line. The second approach —presented in [15]— refines the former by removing the ontology. This improvement is motivated by the difficulty behind the construction of such ontology. Then, the authors propose to annotate the BNF-like grammar with certain information that is used to create a variability model.

The aforementioned approaches support not only abstract syntax variability, but also concrete syntax and semantic variability. In the first case, the ontology can be used to identify all the existing syntactic and semantic variation points since it represents the domain from both the syntax and semantic point of view. In the second case, the annotations provide the expressiveness enough to address all these dimensions of the variability.

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

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

We claim that, in order to facilitate multi-staged configuration, the abstract syntax variability should be defined separately from the semantic variability. The main contribution of our approach constitutes an answer to that claim. We use feature models to represent abstract syntax variability, and orthogonal

variability models to represent semantics variability.

6. Conclusion and future work

In this article we presented an approach to support the construction of bottom-up language product lines. Our approach consists of a reverse-engineering process that allows to automatically produce a language product line from a set of DSL variants. Such reverse-engineering process starts by recovering a language modular design, and then produces variability models that permit configuration of new variants. We validate our approach in a research project that uses different variants of DSL for state machines. However, our approach can be applied to other contexts where the language development process ends up in the construction and maintenance of several variants of DSLs. Some examples of those contexts can be the different languages for expressing petri nets, or even the different languages supporting BPMN —i.e., Business Process Modeling Notation—.

Thinking outside the clone-and-own approach. Thanks to the definition of the comparison operators that we use in the first phase of the approach, we are able to support the case in which the DSL variants are built-up using the clone-and-own approach. 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 3.1, the very first step of our reverse engineering strategy is to perform a static analysis of the given DSLs and apply two comparison in order to specify specification clones. If what we want is to find commonalities that are not necessarily materialized in specification clones but in "equivalent functionality", then we need to enhance the comparison operators in order to detect such as equivalences.

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

Acknowledgments

This work is supported by the ANR INS Project GEMOC (ANR-12-INSE-0011); the bilateral collaboration VaryMDE between Inria and Thales; and the bilateral collaboration FPML between Inria and DGA. We also received support from the European Commission (FEDER); the Spanish government under BELi (TIN2015-70560-R) project and; the Andalusian government under the COPAS (TIC-1867) project.

Appendix A. A family of DSLs for state machines

Generally speaking, state machines are graphs where nodes represent states and arcs represent transitions between the states [53]. 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 project 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.

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. Table **??** shows the language constructs provided by each DSL.

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

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

Table A.2: Diversity of constructs provided by the DSLs for state machines

(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.

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 the state machine presented in Fig A.11. 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.

To tackle this situation, 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

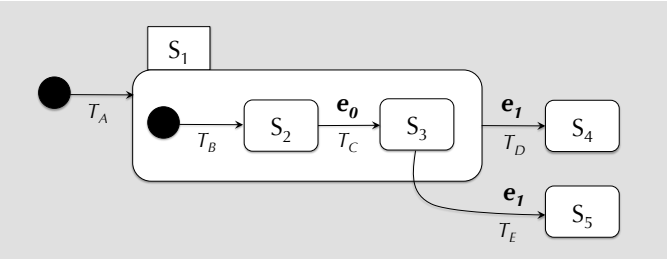

Figure A.11: Example of a state machine with conflicting priorities

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 A.11 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 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 A.11 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.

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] A. Ribeiro, A. R. da Silva, Xis-mobile: A dsl for mobile applications, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, New York, NY, USA, 2014, pp. 1316–1323. doi:10.1145/2554850.2554926.

[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.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] D. Méndez-Acuña, J. Galindo, T. Degueule, B. Combemale, B. Baudry, Leveraging software product lines engineering in the development of external dsls: A systematic literature review, Computer Languages, Systems & Structures 46 (2016) 206 – 235. doi:https://doi.org/10.1016/j.cl.2016.09.004.

[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] 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.

[17] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, G. Le Guernic, 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.

[18] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, Puzzle: A tool for analyzing and extracting specification clones in dsls, in: Proceedings of the International Conference on Software Reuse, ICSR 2016, Springer, Limassol, Cyprus, 2016, pp. 393–396. doi:10.1007/978-3-319-35122-3_26.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] E. Bousse, T. Mayerhofer, B. Combemale, B. Baudry, A Generative Approach to Define Rich Domain-Specific Trace Metamodels, Springer International Publishing, Cham, 2015, pp. 45–61. doi:10.1007/978-3-319-21151-0_4.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] F. Roos-Frantz, D. Benavides, A. Ruiz-Cortés, A. Heuer, K. Lauenroth, Quality-aware analysis in product line engineering with the orthogonal variability model, Software Quality Journal 20 (3) (2012) 519–565. doi:10.1007/s11219-011-9156-5.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A meta-language for modular and reusable development of dsls, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, ACM, New York, NY, USA, 2015, pp. 25–36. doi:10.1145/2814251.2814252.

[49] 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.

[50] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2008) 131. doi:10.1007/s10664-008-9102-8.

[51] 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.

[52] 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.

[53] 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.

[54] 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, 2010. doi:10.1007/978-3-642-12107-4_23.