Variability Management in Domain-Specific Languages

David Méndez-Acuña, Benoit Combemale,
Benoit Baudry, Jérôme Le Noir

INRIA - Centre de recherche
Rennes, Bretagne Atlantique

VaryMDE: Thales Project Report (Deliverable J7)

Rennes, France

August 2014

Variability Management in Domain-Specific Languages

David Méndez-Acuña
INRIA. Rennes, France
david.mendez-acuna@inria.fr

Benoit Combemale
INRIA. Rennes, France
benoit.combemale@inria.fr

Benoit Baudry
INRIA. Rennes, France
benoit.baudry@inria.fr

Jérôme Le Noir
Thales Research and Technology
jerome.lenoir@thalesgroup.com

Abstract

Domain-specific languages (DSLs) allow domain experts to express solutions directly in terms of relevant domain concepts and, for example, use generative mechanisms to transform DSL' specifications into software artifacts (e.g. code, configuration files or documentation). Thus, abstracting away from the complexity of the rest of the system and the intricacies of its implementation. As a result, the construction of DSLs is becoming a recurrent activity during the development of software intensive systems. However, the construction of DSLs is a challenging task due to the specialized knowledge it requires; in order to successfully perform such activity, an engineer must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, interpreters, among others.

The situation becomes even more challenging in the context of multi-domain companies (e.g. Thales) where several domains coexist across the business units and, consequently, there is a need to deal with families of DSLs. A family of DSLs is a set of DSLs that share some commonalities and that differ by some variability. Recent research works have demonstrated the benefits of the use of software product lines engineering (SPLE) in the construction of families DSLs. All of these works agree on the need of a modularization approach that enables the decomposition of a DSL into independent modules and a variability management mechanism for effectively dealing with the differences and commonalities among the DSLs members of the family. The research summarized in this document is aimed to contribute to this study. Concretely speaking, we present a technology for implementing families of DSLs based on the Kermeta 3 framework and we instantiate the approach in the problem of families of languages for expressing finite state machines.

1 Introduction

1.1 Purpose

VaryMDE[1] is a bilateral collaboration (2011 - 2015) between the DiverSE team at INRIA and the MDE lab at Thales Research & Technology (TRT). This partnership explores variability management issues at both the modeling level and the metamodeling level (i.e., design and implementation

[1] VaryMDE project website: http://varymde.gforge.inria.fr

of software languages). This document is the deliverable number seven (i.e., milestone 7) of the second issue and it presents the proposed technology and the first experimentations.

1.2 Research Context

The engineering of complex software intensive systems involves many different stakeholders, each with their own domain of expertise. It is particularly true in the context of systems engineering in which rather than having everybody working with code/model defined in general-purpose (modeling/programming) languages, more and more organizations are turning to the use of Domain Specific Languages (DSLs). DSLs allow domain experts to express solutions directly in terms of relevant domain concepts, and use generative mechanisms to transform DSL specifications into software artifacts (e.g., code, configuration files or documentation), thus abstracting away from the complexity of the rest of the system and the intricacies of its implementation.

The adoption of DSLs has major consequences on the industrial development processes. This approach, a.k.a. Language-Oriented Programming [34], breakdowns the development process into two complementary stages (see Figure 1): the development, adaptation or evolution by *language designers* of one or several DSLs, each capitalizing the knowledge of a given domain, and the use of such DSLs by *language users* to develop the different system concerns. Each stage has specific objectives and requires special skills. Figure 1 depicts the two interdependent processes that continuously drive each other's. The main objective of the language engineering process is to produce a DSL which tackles a specific concern encountered by engineers in the development of a complex system, together with its tooling.

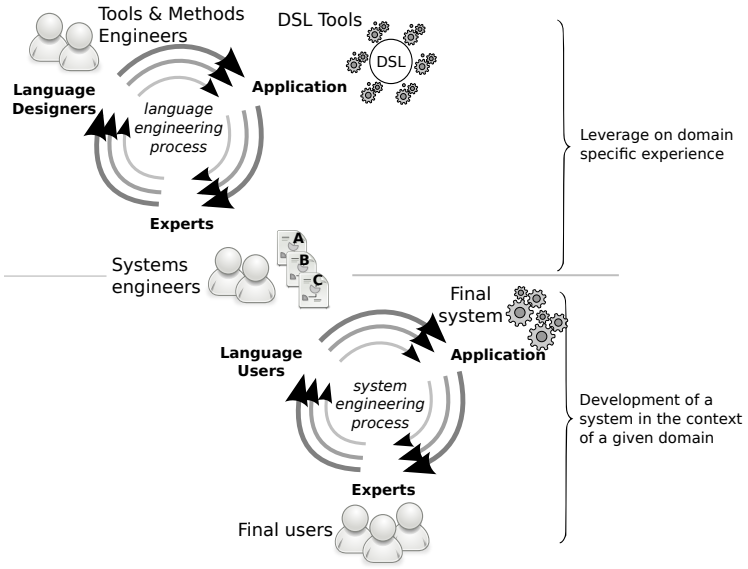

Figure 1: Language engineering stakeholders

Once an appropriate DSL is made available to systems engineers, it is used to express the solution to this specific concern in the final system. However, by definition, DSLs are bounded to evolve with the domain they abstract. Consequently, systems engineers need to be well aware of end users' expectations in order to report their new requirements to the language designers. A new evolved DSL is then produced by the language designers, which is in turn used by systems engineers and so on and so forth. It is worthwhile to note that, although this is unlikely in large companies, these roles can be alternatively played by the same people in smaller organizations.

As a matter of fact, while DSLs have been found useful for structuring development processes and providing abstractions to stakeholders [18], their ultimate value has been severely limited by their user-understanding ambiguity, the cost of tooling and the tendency to create rigidity, immobility and paralysis (the evolution of such languages is costly and error-prone). The development of software

languages is a challenging task also due to the specialized knowledge it requires. A language designer must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, and interpreters. The situation becomes even more challenging in the context of multi-domain companies (e.g. Thales) where several domains coexist across the business units and, consequently, there is a need to deal with families of DSLs. A family of DSLs is a set of related DSLs that share some commonalities but that are distinguished each other for certain particularities. Languages for expressing state machines are a good example of such as families where some of the most known variation points are found at the level of the semantics: the same construct of the language can be interpreted differently [7].

The research presented in this document explores the problematic around families of DSLs. In particular, we present an analysis of the challenges to overcome towards the construction of an approach that supports their construction and, besides, we introduce a first prototype of a tool that facilitates the definition of such families of DSLs on top of Kermeta 3.

1.3 Organization of the Document

The reminder of the document is structured as follows: Section 2 introduces some important definitions needed for the understanding of the document. Section 3 analyses the challenges towards the construction of an approach that supports the implementation of families of DSLs. Section 4 presents the proposed approach that is illustrated by using a case study presented in section 7. Section 10 concludes the document and presents the perspectives. Section 8 shows a tool demonstration of the technology through some screen-casts that illustrate the use of the tool for the case study and, finally, 9 summarizes the related work.

2 Preliminary Definitions

2.1 Domain-Specific Languages

Typically, the implementation of a DSL is conducted at two different levels: the description of the language itself and the construction of the usage tooling. The description of a DSL refers to its specification in terms of three dimensions: (1) **abstract syntax** that refers to the domain concepts of the language domain and the relationships among them; (2) **concrete syntax** that refers to the syntactical rules that allow to use those concepts; and (3) **semantics** that refers the meaning of those concepts often accompanied with certain domain constraints [29]. In the general case, this description is written by using dedicated formalisms that offer the expressiveness enough for specifying each dimension of the specification. Since those formalisms are languages intended to specify languages, they are known as meta-languages [20] and they vary depending on the technological space used of the implementation of the language. For instance, in the case of metamodel-based language specifications with textual concrete syntax, abstract syntax can be specified by using meta-models written in the MOF [28] language whereas concrete syntax in BNF-Like grammars. Model transformation languages such as QVTo [27] or ATL [21] or action languages such as Kermeta [19] can be used to specify language semantics.

Note also that there are some dependency relationships between description dimensions. In the aforementioned example, there is a dependency relationship between the concrete syntax specification unit and the abstract syntax specification unit that represents the mapping between the metaclasses in the metamodel that specifies the abstract syntax, and the production rules of the grammar that specifies the concrete syntax. Similarly, there is a relationship between the semantic specification unit and the abstract syntax specification unit that represents the mapping between the metaclasses in the metamodel and the transformations rules or the metamodel actions (depending if the semantics are defined with transformations or as operational semantics).

On the other hand, the construction of the usage tooling refers to the implementation of all the infrastructure that allows users to interact with the language e.g., editors, compilers, interpreters, or type checkers. As shown in Figure 2 the language description is the base for the construction of the usage tooling. Indeed, thanks to the current advances in software languages engineering, most part of the usage tooling can be automatically produced from the description.

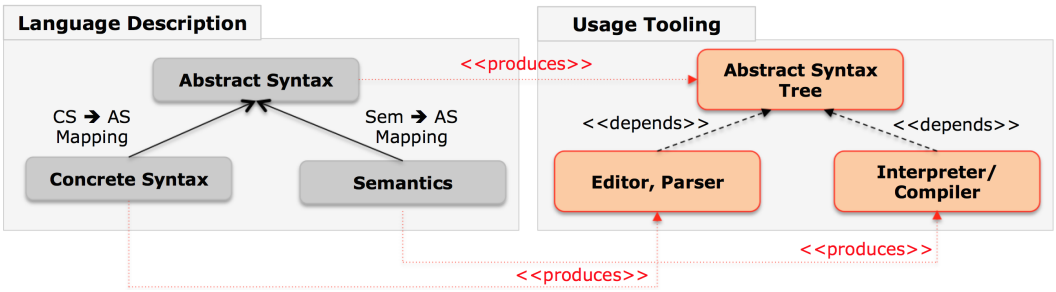

Figure 2: DSLs description versus usage tooling

2.2 Language Features and Language Units

According to [23], dealing with families of DSLs is dealing with sets of related languages that share some commonalities and that differ in some particularities, consequently, *language features* can be viewed as abstractions that allow to capture these commonalities and particularities in such a way that can be used latter in feature models for expressing the variability of the family [35].

At the language description level, such commonalities and particularities are segments of the language descriptions that can be either shared by different members of the family or that may be specific for particular members of the family. Along this paper we refer as **language features** as the domain abstractions that can be used in a feature models for expressing a capability of the language family whereas **language units** as segments of the specification that implement the language features. Note that language features are platform independent since they only depend on the domain analysis. In turn, language units are defined in terms of the technological space used for the implementation of a language since those encapsulate the software artifacts that actually implement the domain abstractions.

2.3 Variability within Families of Domain-Specific Languages

It is worth noting that differences and particularities among members of a family of DSLs can be found in one or several dimensions of the specification. For example, a domain concept can have different semantics in two members of the family. Indeed, the work presented in [6] provides a classification of the possible types of variability that can be found within families of DSLs. A brief summary of this classification is introduced below:

- **Functional variability:** One of the motivations for implementing families of DSLs is to offer customized languages that provide only the constructs required by certain type of users. The hypothesis is that the user will adopt the language easier if the language only offers the constructs he/she actually needs. If there are additional concepts the complexity of the language (and the tools) needlessly increases and *"the users are forced to rely on abstractions that might not be naturally part of the abstraction level at which they are working"* [35]. Functional variability refers to the capability of selecting the desired language constructs for a particular type of user. Because of the abstract syntax of the language is the base of the specification –i.e., there can not be definitions neither in the concrete syntax nor the semantics for concepts that do not exist in the abstract syntax– the functional variability relies on the activity of selecting the subset of the abstract syntax required for a particular user.
- **Syntactic variability:** Depending on the context and, again, on the type of user, the use of certain types of concrete syntax may be more appropriate than other one. Consider for example the dichotomy between textual or graphical notations. Empirical studies such as the presented in [26] show that, for a specific case, graphical notations are more appropriate than textual notations whereas other evaluation approaches argue that textual notations have advantages in cases where models become large [10]. More intermediate perspectives (e.g., [17]) state that graphical and textual notations more that mutually exclusive are complementary. Syntactical variability refers to the capability of supporting different rep-

resentations for the same language construct. In terms of the specification, the syntactical variability can be viewed as different implementations of the concrete syntax specification unit for a given abstract syntax specification unit.

- **Semantic variability:** Another problem that has gained attention in the literature of software languages engineering is the semantic variation points existing in software languages. A semantic variation point appears where the same construct can have several interpretations. Consider for example the semantics differences that exist between state machines languages explored in [7]. For example, the construct *fork* can be interpreted as a concurrency point where all the output transitions are dispatched simultaneously or simply as a bifurcation point where the output transitions are dispatched sequentially. Semantic variability refers to capability of supporting different interpretations to the same language construct. In terms of the specification, semantic variability can be viewed as different implementations of the semantic specification unit for a given abstract specification unit.

It is important to mention that language features should be defined in a degree of granularity that enables the support of the three different types of variability. A possibility for addressing that requirement is to divide in language features each dimension of the specification. Hence, there may be language units for abstract syntax, for concrete syntax or for semantics and there would be a **one-to-one relationship** between language features and language units. Figure 3 shows a simple example of the relationships between language features and language units by presenting a set of language features organized in a features model and the corresponding set of language units that actually implement the functionality. In particular, the features model is divided in three main branches each one for a specification dimension. Then, a set of features hierarchically organized are provided when each concrete feature is implemented in a language unit. Note that dependencies between language units should be reflected as dependencies between language features.

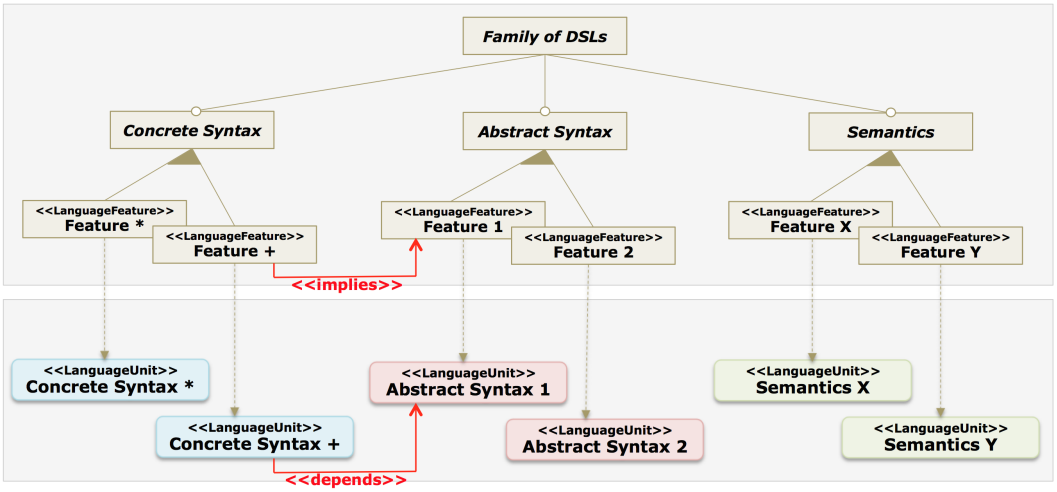

Figure 3: Language features, language units, and families variability

3 Research Challenges

At this point we can state that, to offer an approach for the implementation of families of DSLs, there should exist: (1) a mechanism for **languages modularization** so a language family can defined in terms of composable language units; (2) a **variability management approach** that allows to model the variability of the management and that provides a configuration mechanism that facilitates the selection of the desired features for a particular product; and (3) a strategy for **language features composition** so the selection of a user can be successfully integrated in a language that actually works. Each of these challenges is briefly explained below.

- **Languages modularization:** The first challenge towards the implementation of families of DSLs corresponds to the decomposition of a language specification in several language

modules (a.k.a., language units). Specifically, this challenge must be addressed at both the technological and the design levels. In the case of the technological level, a workbench for languages modularization should be provided. This benchmark needs to offer a mechanism for independently implementing separate language units than can be composed latter. At the design level, it is important understand how the language units should be defined so they can actually capture commonalities and variability of the family e.g., to define the level of granularity in which the language unit should be decomposed.

- **Multi-stage orthogonal variability modeling:** Modeling and dealing with the variation points existing in a family of the DSLs is the second challenge to overcome. The multi-dimensional nature of this type of variability implies that a variability modeling approach must be aware not only of the functional but also syntactic and semantic variability. Also, approaches should mind the relationships between specification units since they represent dependency relationships: both syntactic variability and semantic variability depend on the functional variability because it makes no sense to select a syntactic rule for a language construct that has not been included as part of the language product. In other words, the configuration of syntax and semantics must be performed only for the construct selected in the functional variability resolution.
- **Language units composition:** After successfully modeling the variability, the next challenge is to compose a concrete DSL from a configuration of the family. In other words, after selecting the required language units, they have to interact each other for constituting a DSL that actually works. To do so, a mechanism for language unit composition is required. On the other hand, it is important to mention that, in the general case of software product lines, modularization and variability management are strongly linked. Each concrete feature of the family, expressed in the variability model, must correspond to a software component in the architecture so a given configuration can be derived in a concrete functional product. In the case of DSLs each concrete feature should be mapped to one (ore more) language units that offer(s) the corresponding functionality. Hence, it is important to have a synchrony between the modularization approach and the variability management mechanism so the decisions made during a DSL configuration can be successfully translated to a subset of composable language units that, actually, provide a correct DSL.

4 Languages Modularization

Typically, languages modularization has been addressed at the level of the specification of the language. What is actually decomposed is not the language tooling itself (i.e., editors, interpreters, compilers) but the specification used for generate them. Consequently, composing two language units corresponds to merge to parts of the specifications for later generate the tooling of the composed language [25]. We propose a different strategy for DSLs modularization that orchestrates the services offered/required by the language unit's tooling rather than composing specifications. This modularization approach is transversal to the three dimensions of the DSL specification. In order to tackle the binding between the modularization approach and the variability management mechanism, we use the proposal of variability realization techniques proposed by van der Linden et al in [24] (Figure 4) for aligning components-based software architectures with variability management approaches in the general case of software development. The variability management approach is still part of our future work.

Because our approach is based on the Kermeta 3 framework, we start by giving a brief introduction to that tool. Then, we explain the modularization schema in terms of the variability realization techniques.

4.1 Kermeta 3 (K3): Executable meta-modeling

K3 is a framework that allows to enhance metamodels written in Ecore language with structural features (such as new attributes or invariants) and behavior i.e., dynamic semantics. Because K3 is implemented on the top of the Eclipse Modeling Framework (EMF) [31], it is possible to use it in junction with tools such as xText [3], EMFText [1], or Sirius [9] for generating textual or graphical editors. As a result, we can completely construct DSLs by using this framework: abstract syntax with metamodels in Ecore, semantics with K3, and concrete syntax with editors generators.

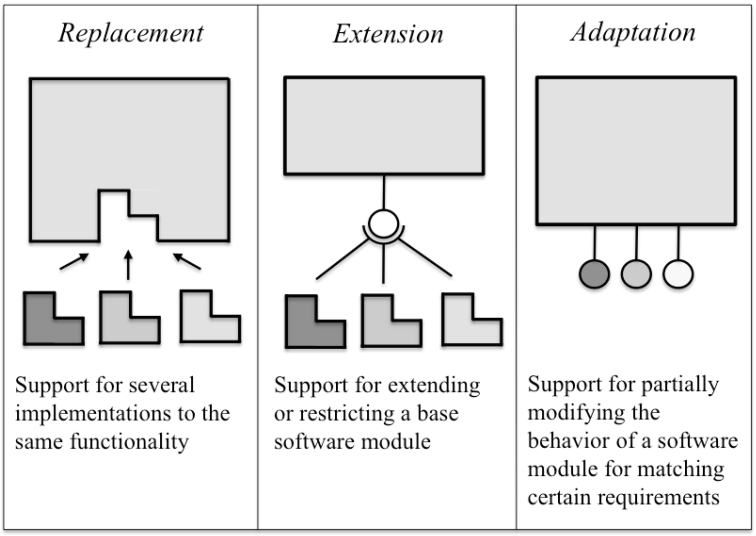

Figure 4: Variability realization techniques [24]

It is important to mention that K3 is based on the idea of *"aspects"*. In this context an aspect is an entity that extends certain meta-class existing in a given metamodel. In the case of dynamic semantics, aspects permit to specify not only operational semantics (for building interpreters) but also translational semantics (for building compilers).

K3 is built on the top of xTend [2] and it uses the notion of *"active annotations"* for expressing aspects. Figure 5 illustrates this fact by using a simple example. At the left part of the figure, there is a metamodel that contains two classes `X`, and `Y`; The class `X` contains elements of type `Y` by means of the containment relation `yes`. The right side of the figure shows the operational semantics attached to this metamodel by using aspects in K3 (i.e., `AspectX` and `AspectY`). The metaclass `X` is enriched with the operation `eval()` that contains a loop that sequentially invokes the operation defined for the class `Y`. This operation is also defined by using one aspect; in this case extending the metaclass `Y`. Note that the language engineer is free to name the aspects as he/she considers correct. In this example, we use the convention `Aspect+<<metaclass.name>>`. The reader will find further information in the K3 website[2].

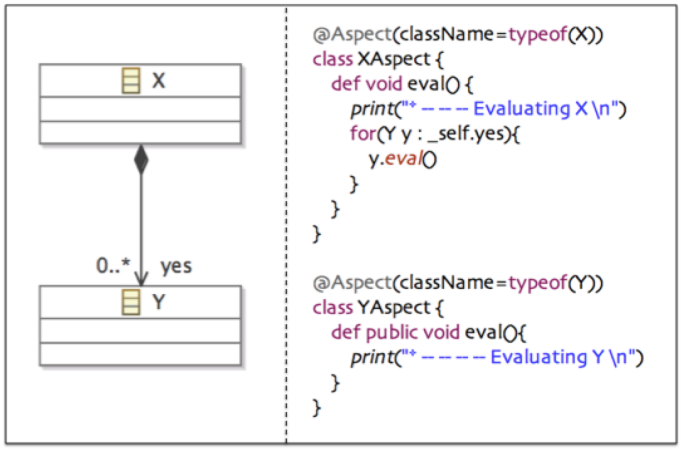

Figure 5: A simple example that illustrates the use of K3

[2]K3 website: http://diverse-project.github.io/k3/

4.2 Variability realization techniques in action

Our modularization approach quite inspired on the idea of *"services registries"* such proposed for the classical solutions for Service Oriented Architecture (SOA) [8]. Provided services are stored in a global registry and the required services perform a look them up when required by means of a canonical name. Currently, we offer support for the orchestration at the level of the abstract syntax and at the level of the semantics, thus, we need two different registries: the structural one (that store structural services or concepts) and the behavioral one (that stores services or methods); a method is defined for a concept so there is a mapping between concepts and methods.

- **Replacement:** In replacement there is an incomplete language unit (the requiring language unit) that requires some services from other one (the providing language unit) for working correctly. The providing language unit can be any one that satisfies the requirements established by the requiring one. Figure 6 illustrates this issue by means of two language units `A` and `B`. Language unit `A` requires some services provided by the language unit `B`. At the abstract syntax level, the required service is the meta-class `X` referenced by the meta-class `Q`. At the semantics level, the required service is an abstract method added to the meta-class `X`. Note that we use the annotation `@Required` that (at runtime) is a lookup to the services registry. The language unit `B` implements the class `X` and registers the service `X.eval()` to the services registry by means of the `@Provided` annotation.

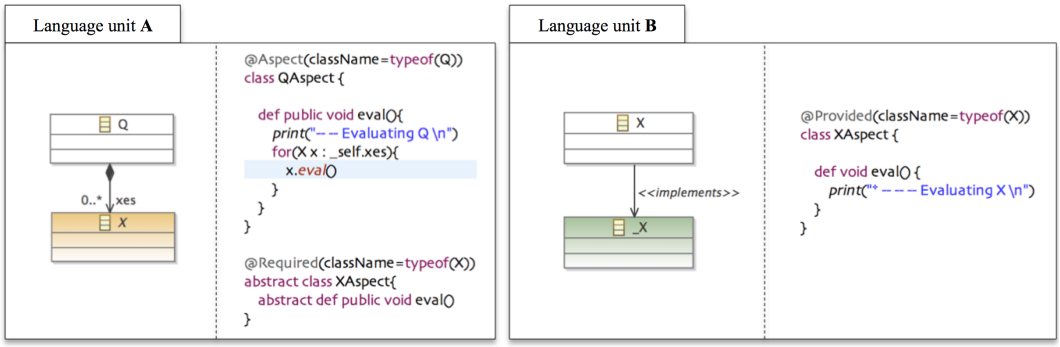

Figure 6: An example for the *replacement* variability realization technique

- **Adaptation:** Adaptation appears when the services offered by a providing language unit do not exactly match the required services of a requiring language unit and, consequently, some modification of the services is needed. In this case, there should be a language unit that serves as *"adapter"* and whose responsibility is to offer services that match the requirements of the requiring language unit. Note that the annotations we offer for the replacement operator are enough for implementing adapters. Figure 7 illustrates this realization technique. In this case, language unit `A` requires of a meta-class called `X` and with an operation `eval`. However, the meta-class provided by the language unit `B` is called `P` and the operation is called `exec`. Hence, an adapter that offer the services exactly as required by the language unit `A` and that uses the services offered by language unit `B`. Although this example considers only naming mismatching the adaptation can be more complex.
- **Extension:** In the case of extension, there is base language unit that can be extended (or specialized) by other one(s) that we call contributions. Figure 8 illustrates this extension mechanism. In this case, language unit `B` extends language unit `A` by offering a contribution over the meta-class X. To do so, we offer the annotation `@Contribution` that indicates that the annotated class is a contribution of certain extension point indicated on the parameter "extensionPoint".

5 Variability management

After being presented the modularization approach, we present the strategy for dealing with the variability existing in a family of DSLs.

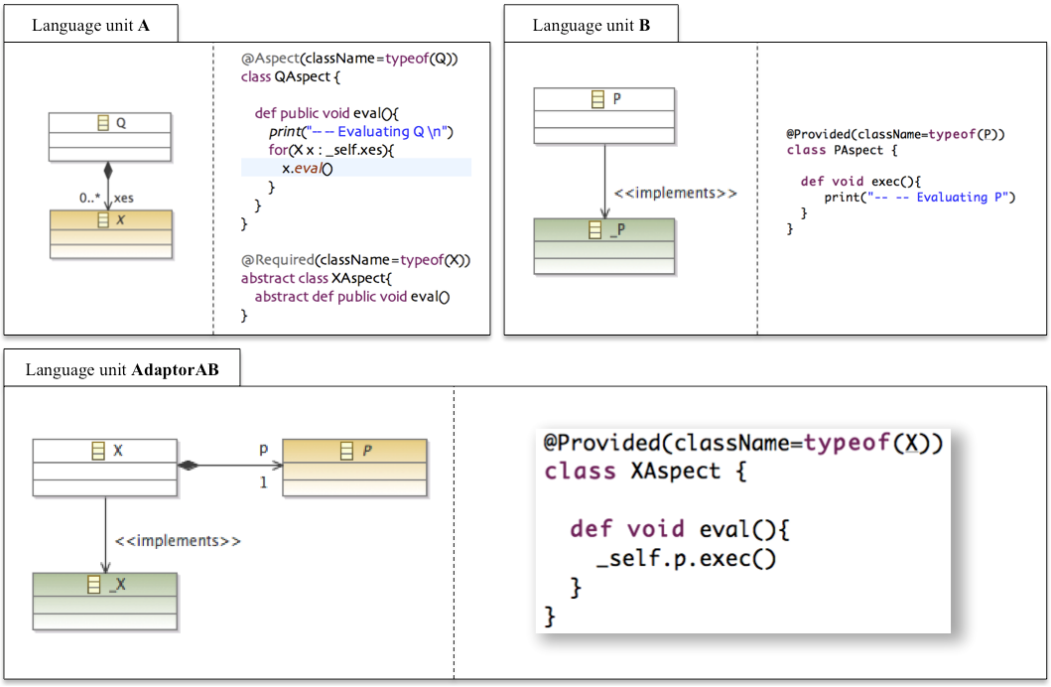

Figure 7: An example for the *adaptation* variability realization technique

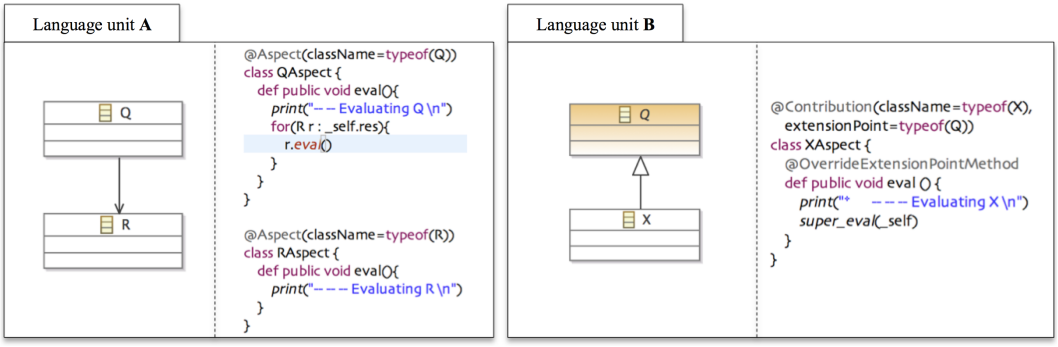

Figure 8: An example for the *extension* variability realization technique

5.1 CVL as variability modeling language

In our case, we use the choice diagram proposed by CVL (**C**ommon **V**ariability **L**anguage) [15] very similar to an attributed feature diagram with cardinalities. CVL defines a variability model and a resolution model which act in tandem on a base model to derive a resolved model. It facilitates the specification and resolution of variability over any instance of any language defined using a MOF-based meta-model.

Figure 9 which has been taken directly from the CVL specification [16], depicts a high level view of the underlying design. The variability model defines two models: the variability abstraction model and the variability realization model. The variability abstraction model defines the features of the domain in a tree structure, thus creating hierarchical relationships as the same as feature models. The nodes of such a tree structure are defined as `VSpecs`, and in the current specification there exists three types: `Choices`, `Variables` and `Classifiers`. The `Choices` are `VSpecs` that can be resolved to yes or no (through `ChoiceResolution`), `Variables` are `VSpecs` that allows the capturing of a value which can be used in the resolution(`VariableValue`) and `Classifiers` are `VSpecs` that suppports the creation of instances and then providing per-instance resolutions

(`VInstances`). Constraints between features can also be defined, thereby traversing the tree structure which ensures that the set of features selected result in a valid combination of features. The variability realization model defines the mapping and the choises and the concrete language units that actually implement them. The variation point defines a set of actions that would need to be applied to a base model. A variation point is the point on the base model which would need to be modified, when the associated feature is selected or when not selected (rejected).

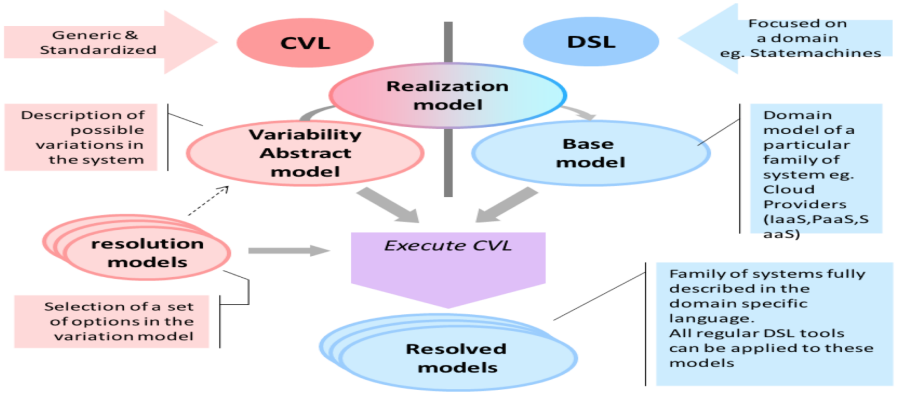

Figure 9: Overview of the CVL principles (taken from [16])

5.2 Dealing with multi-stage variability

For dealing with multi-stage variability, we take some ideas from some previous work on multi-dimensional variability [30]. It is important to mention however that this is a first draft of the variability management approach and that a rigorous well-engineered one is part of our ongoing work. For now, we use a simple strategy (illustrated in Figure 10) where each variability dimension is a direct child of the root of the variability tree. Because of the scope of this paper is limited to abstract syntax and operational semantics, we only include two children: the first one models the functional variability and the second one models the interpretation variability. It is important to mention that are dependencies between the operational semantics and the abstract syntax that must be minded. As a result, there are some implication relationships between abstract syntax features and semantics features. Note that each semantic feature need to be mapped to one abstract syntax feature and several semantics features can be mapped to the same abstract syntax. When this occurs, we will say that there is a semantic variation point for the same concept i.e., the same construct may be interpreted differently.

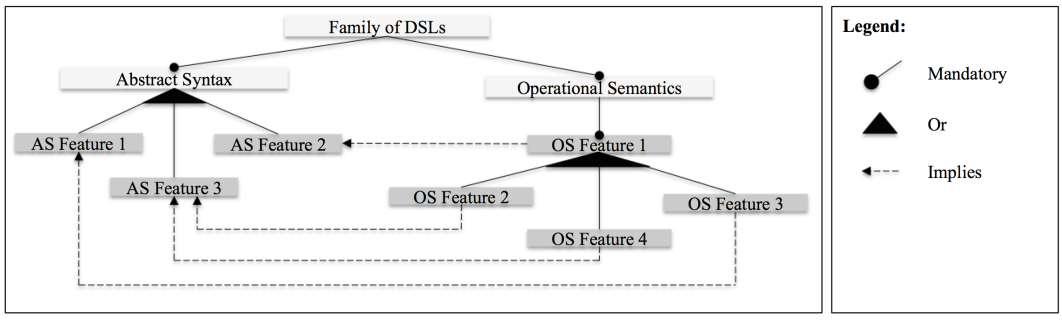

Figure 10: Example of variability model dealing with multi-stage variability

6 Language derivation

The input for the language derivation phase is a configuration of the family i.e., a correct set of features from the feature model. Once we have that, our tool is able to compose the corresponding language. In our case, because we are using the interpretation mode, the derivation process corresponds to the registry of the services –both, structural and behavioral– corresponding to the selected features. To do so, we simply navigate the feature model founding the selected features and then we found the corresponding implementations in the references to the realization model. Of course, there is a different treatment for structural registry than for behavioral registry since the types of artifacts are different: While structural features are registered as concepts in the structural registry, the behavioral features are register as services in the behavioral registry. We need that the realization object contains not only the reference to the concrete artifact to register (i.e, the class) but also some additional information.

7 Case study: A family of languages for finite state machines

In this section we introduce as a case study a family of DSLs for modeling state machines. It corresponds to a real business requirement for Thales since, despite the company uses state machines in several scenarios, there is not a unified language for expressing state machines. Instead, the company uses several formalisms each of which defines its own abstract syntax, concrete syntax and semantics. As a result, there is a big amount of state machine models and languages that are not necessary compatible each other: a model in a formalism A may be not well-formed in another formalism B. Moreover, there are some cases where, despite the model is well-formed in the two formalisms, the semantics is different. Then, the same model may have completely different behaviors when executed in two different interpreters. As the reader may imagine, maintaining an infrastructure for supporting such as heterogeneity of languages is quite expensive and impractical.

Currently there are some works in the literature that deal with this problem of families of languages for expressing state machines. For instance, the work presented by Crane et. al in [7] discusses the variation points –not only at syntactical level but also at the semantical one– among three of the most used state machines formalisms: UML [12], Classical Harel's statecharts [14] and Rhapsody [13]. There are also tools such as Polyglot [4] whose contribution is the implementation of such as families in integrated platforms that includes several interpreters where the models can be shared since they are syntactically compatible. However, it is not enough where hybrid approaches are needed (i.e., situations where the semantics for some constructs are interpreted as in one formalism but other are interpreted by following the philosophy of other formalism).

Figure 11 presents a part of the variability model we proposed for the state machines. Note that this feature model is highly inspired on the work presented in [7]. On the abstract syntax part, we have the functional variability that allow us to select some optional constructs. For example, timed transitions is a construct that is not supported by Rhapsody users since this formalism assumes that each transition takes exactly zero time to be executed. In the part of the semantics variation points we focus our study in the synchronization problematics. This is divided in three issues: (perfect) synchrony hypothesis, events synchrony, and fork parallelism. The (perfect) synchrony hypothesis refers to the fact that each transition take zero time for being executed. In that sense, there are formalism such as classical state charts that support this assumption and there are other formalisms where a transition may take some time so they offer support for timed transitions. Note that these later approaches also support the the synchrony hypothesis since the time of a transition can be defied as zero. Simultaneous events refers to the capability of supporting the arrival of several events at the same time. Classical state machines support this capability whereas UML state machines follow the run for completion principle that states that events are attended sequentially and no event can start after the previous one have finished.

In turn, Figure 12 summarizes the configurations for each of the mentioned formalisms with respect to the aforementioned synchrony problematic. Note that there are several other variation points concerning, for example, to history pseudo states or conflictive transitions resolution. Nevertheless, this first experiments demonstrates the capability of our tool for supporting not only functional variability but also interpretation variability.

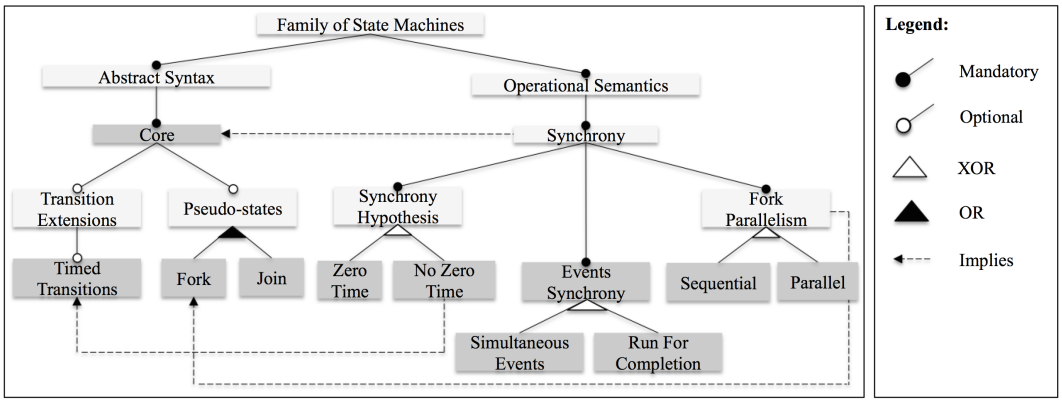

Figure 11: A subtract of the variability model for the family of state machines

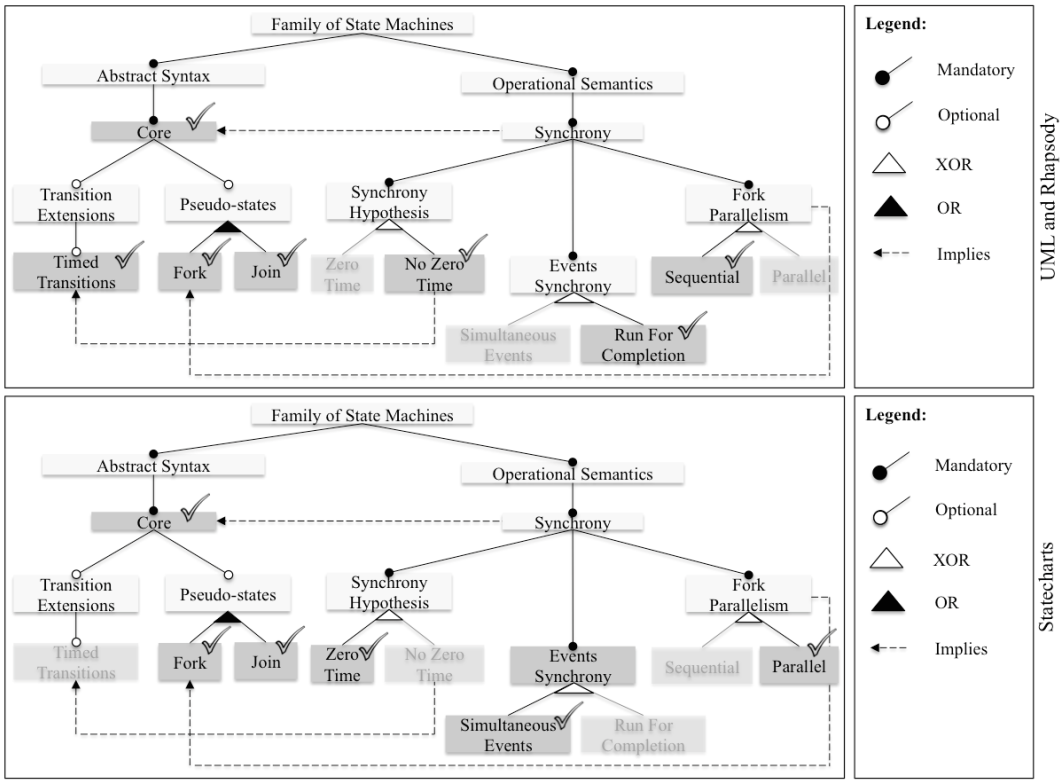

Figure 12: Two configurations for the family of languages for state machines

8 Tool demonstration

Two screencasts present the current status of the prototype, illustrated on the family of DSLs for state machines. The first one shows the execution of the simple state machine (shown in figure 13) that contains only the concepts of `State` and `Transition`. Then, in the second link you can find the execution of the state machine (shown in figure 14) that has been extended with the concept of `TimedTransition`. Note that the transition `o` takes three seconds for being executed.

For each case, we first show the configuration of the variability model for including/excluding the extension `TimedTransition`. Then, we show the corresponding instance model that contains the state machine. Finally, we execute the application and, by using the console of Eclipse, we interact with the state machine.

- **Screencast 1 - Simple state machines:** `http://quick.as/4Zxrc0z7`

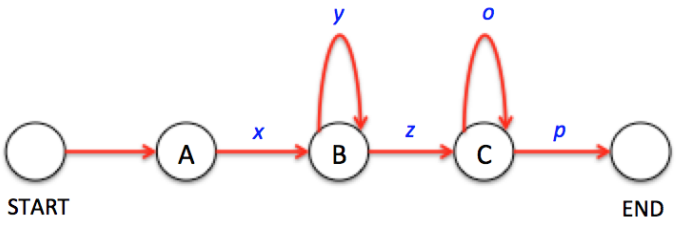

Figure 13: A simple state machine

- **Screencast 2 - Timed state machines:** `http://quick.as/2k74Hkev`

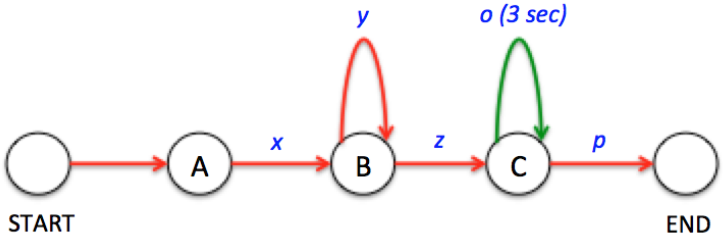

Figure 14: A state machines with timed transitions

9 Related work

It is worth nothing that the idea of families of languages has already been discussed previously in some research work such as [33, 6, 23, 32]. In general, approaches agree on the need of a modularization approach that enables the decomposition of a language into independent *language features* that latter can be composed again according to a specific configuration in the variability management mechanism. The reminder of this section is dedicated to explore how some of the most relevant works in the area, deal with the aforementioned challenges.

In the work presented in [33] the modularization problem is treated by using Neverlang [5]: a tool that allows to express a software language in separate *language units* that can be later composed for generating an interpreter only for the selected language units. The variability management mechanism is addressed by using the Common Variability Language (CVL). In that work a language unit is composed (at the same time) of the abstract syntax, the concrete syntax and the semantics. As a result, for supporting different semantics for the same concept, two language units should be expressed repeating the syntax definition and variating the semantics implementation. We can state that, although syntactic and semantic variation points may be supported, this approach is mainly focused on functional variability. Differently, the work presented in [6] is focused on syntactic and semantic variability where abstract and concrete syntax are expressed in a languages benchmark called MontiCore [22] whereas semantics are specified in Isabelle/HOL [11]. The variability management mechanism is based on feature models that allows to users the configuration of a language according to the requirements of an application domain. Note that despite MontiCore is an approach intended to support languages modularization, neither the variability management mechanism consider functional variation points nor the semantics are modularized. Consequently, functional variability is not supported. The work presented in [23] go further and discusses not only functional variability but also semantic and syntactic variability under the concept of *crosscutting modularization*. That is the capability of decomposing a language not only into different language units but also decomposing

a language feature into several tool features. Each tool feature represents a dimension of the language specification (e.g., syntax, semantics, constraints, documentation). In a first view, crosscutting modularization would enable the support for addressing all the variability dimensions of families of DSLs. However, the authors do not conduct an experiment that demonstrates that the same language feature can be configured differently by selecting different segments units.

10 Conclusions and perspectives

In this paper we presented our experience on dealing with families of DSLs in an industrial context, specifically in the Thales company for the concrete case study of a family of state machines languages. To do so, we first presented the main challenges towards this end and, then, we present the implementation of our approach in terms of the tools facilities we built for language engineers. Our approach is built on top of the language workbench Kermeta 3.

There still are some steps left to be able to completely introduce the use of the family in the company. In particular, it is important to include in the scope of the approach the support for supporting concrete syntax of DSLs so our approach can be used easier by final users.

References

[1] Eclipse, emftext, June 2014.

[2] Eclipse, xtend: Modernized java, June 2014.

[3] Eclipse, xtext, June 2014.

[4] Daniel Balasubramanian, CorinaS. Psreanu, Gbor Karsai, and MichaelR. Lowry. Polyglot: Systematic analysis for multiple statechart formalisms. In Nir Piterman and ScottA. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 523–529. Springer Berlin Heidelberg, 2013.

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

[6] María Victoria Cengarle, Hans Grnniger, and Bernhard Rumpe. Variability within modeling language definitions. In Andy Schrr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 670–684. Springer Berlin Heidelberg, 2009.

[7] MichelleL. Crane and Juergen Dingel. Uml vs. classical vs. rhapsody statecharts: not all models are created equal. *Software & Systems Modeling*, 6(4):415–435, 2007.

[8] Jrgen Dunkel and Carsten Kleiner. On managing services in service-oriented architectures. In Ezendu Ariwa and Eyas El-Qawasmeh, editors, *Digital Enterprise and Information Systems*, volume 194 of *Communications in Computer and Information Science*, pages 410–424. Springer Berlin Heidelberg, 2011.

[9] Eclipse. Sirius, July 2014.

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

[11] Hans Gronniger, JanOliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In David Lee, Antnia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems*, volume 5522 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 2009.

[12] O. M. G. Group. UML Specification, Version 2.0.

[13] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In Hartmut Ehrig, Werner Damm, Jrg Desel, Martin Groe-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkmper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.

[14] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[15] Oystein Haugen. Cvl: Common variability language or chaos, vanity and limitations? In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 1:1–1:1, New York, NY, USA, 2013. ACM.

[16] Oystein Haugen. Cvl specification, July 2014.

[17] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In RichardF. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009.

[18] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480. ACM, 2011.

[19] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In JooM. Fernandes, Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 201–221. Springer Berlin Heidelberg, 2011.

[20] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and Franois Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[21] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: A qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

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

[23] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented language families: A case study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

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

[25] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9):2451 – 2464, 2013.

[26] B. Mora, F. Garca, F. Ruiz, and M. Piattini. Graphical versus textual software measurement modelling: an empirical study. *Software Quality Journal*, 19(1):201–233, 2011.

[27] The Object Management Group (OMG). Mof specification, July 2014.

[28] The Object Management Group (OMG). Qvto specification, July 2014.

[29] Andreas Prinz, Markus Scheidgen, and MereteS. Tveit. A model-based standard for sdl. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007: Design for Dependable Systems*, volume 4745 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2007.

[30] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-dimensional variability modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 11–20, New York, NY, USA, 2011. ACM.

[31] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[32] Edoardo Vacchi, Walter Cazzola, Benoit Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *SPLC'14 - 18th International Software Product Line Conference*, Florence, Italie, September 2014. ACM.

[33] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benot Combemale. Variability support in domain-specific language development. In Martin Erwig, RichardF. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 76–95. Springer International Publishing, 2013.

[34] M. P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[35] Steffen Zschaler, Pablo Snchez, Joo Santos, Mauricio Alfrez, Awais Rashid, Lidia Fuentes, Ana Moreira, Joo Arajo, and Uir Kulesza. Vml* a family of languages for variability management in software product lines. In Mark van den Brand, Dragan Gaevi, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer Berlin Heidelberg, 2010.