HAL
archives-ouvertes.fr

Variability Support in Domain-Specific Language Development

Edoardo Vacchi, Walter Cazzola, Suresh Pillay, Benoit Combemale

To cite this version:

Edoardo Vacchi, Walter Cazzola, Suresh Pillay, Benoit Combemale. Variability Support in Domain-Specific Language Development. SLE - 6th International Conference on Software Language Engineering, Oct 2013, Indianapolis, IN, United States. pp.76-95, 10.1007/978-3-319-02654-1_5 . hal-00914715

HAL Id: hal-00914715
https://hal.inria.fr/hal-00914715

Submitted on 5 Dec 2013

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

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

Variability Support in Domain-Specific Language Development

Edoardo Vacchi[1], Walter Cazzola[1], Suresh Pillay[2], and Benoît Combemale[2]

[1] Computer Science Department, Università degli Studi di Milano, Italy
[2] TRISKELL (INRIA - IRISA), Université de Rennes 1, France

Abstract. Domain Specific Languages (DSLs) are widely adopted to capitalize on business domain experiences. Consequently, DSL development is becoming a recurring activity. Unfortunately, even though it has its benefits, language development is a complex and time-consuming task. Languages are commonly realized from scratch, even when they share some concepts and even though they could share bits of tool support. This cost can be reduced by employing modern modular programming techniques that foster code reuse. However, selecting and composing these modules is often only within the reach of a skilled DSL developer. In this paper we propose to combine modular language development and variability management, with the objective of capitalizing on existing assets. This approach explicitly models the dependencies between language components, thereby allowing a domain expert to configure a desired DSL, and automatically derive its implementation. The approach is tool supported, using Neverlang to implement language components, and the Common Variability Language (CVL) for managing the variability and automating the configuration. We will further illustrate our approach with the help of a case study, where we will implement a family of DSLs to describe state machines.

Keywords: Domain-Specific Languages, Language Design and Implementation, Variability Management, CVL and Neverlang

1 Introduction

In computer science, we call *domain-specific language* (DSL) a language that is targeted towards a *specific problem area*. DSLs use concepts and constructs that pertain to a particular domain, so that domain experts can express their intentions using a language that is closely aligned with their understanding. For instance, mathematicians often prefer MATLAB or Mathematica, while in the modeling world we often talk about domain-specific *modeling* languages (DSMLs). In the last few years, industry has shown a growing interest in DSL development [15], because complex problems are more easily expressed using problem-tailored languages. However these complex problems tend to have variations, thus requiring different language implementations.

Traditional language development is a top-down, *monolithic process*, that provides very little support in terms of reuse and management of reusable parts.

Many modern programming languages include DSL-development oriented features (e.g., Scala's parser combinators, Groovy, and so on). However, language development is still far from being within everyone's reach. Language development tools are generally not built for direct interaction with the end user of the language; but rather the language developer. Thus, although componentized development is today the norm, even in the case of language development, complete language workbenches such as Xtext [9] or MPS [25] are usually top-down, end-to-end development tools that are meant for programmers, and therefore less suited for programming-illiterate users. Componentized language frameworks such as LISA [19], JastAdd [10], or Neverlang [2,3] support *reuse* of language components, but each component may have implicit dependencies on other parts, and often these dependencies are not managed automatically by the system, but are delegated to the developer.

We believe that combining variability and modular language frameworks would bridge the gap between developers and end users, thereby further promoting re-use. In *software product lines* [6], variability models represent the family of products. Some works [22, 28] have shown that variability modeling improves code reuse in DSL development, in that it makes *explicit* the way components in a collection may cooperate, how to avoid conflicts and how to ensure that dependencies are included. Even though it has been recognized as good practice, variability modeling in language development is still an overlooked aspect, and most language frameworks usually do not natively take into account its importance. The contribution of this work is an approach to apply variability modeling to component-based language development, that focuses on reuse of existing assets. In particular, we describe

1. a method to extract structured information from the set of existing assets in the form of a graph of dependencies,
2. a strategy to construct a variability model using the extracted information,
3. an implementation of a derivation operator to generate the language implementation from the VM automatically,

thereby facilitating the collaboration between the language developer and the domain expert, to the extent that the domain expert becomes autonomous in extracting a desired language. The implementation of this approach will be demonstrated using a real working toolset applied to a family of state machine languages.

The rest of this paper is structured as follows: in Sect. 2 we provide some background in terms of a modular language implementation and variability modeling; in Sect. 3 we give an overview of the approach. In Sect 4 we describe the approach in detail starting with a set of components and in Sect. 5 we apply variability techniques. In Sect. 6 a case study of a family of statemachines is provided. Finally, in Sect. 7 we discuss the related work and in Sect. 8 we draw our conclusions.

```
module com.example.AddExpr {
  reference syntax {
    AddExpr ← Term;
    AddExpr ← AddExpr "+" Term;
  }

  role(evaluation) {
    0 { $0.value = $1.value; }
    2.{ $2.value = (Integer) $3.value + (Integer) $4.value; }.
  }
}
slice com.example.AddExprSlice {
  concrete syntax from com.example.AddExpr
  module com.example.AddExpr with role evaluation
}
```

Listing 1: A simple Neverlang slice defining the syntax and semantics for the sum. Numbers refer to nonterminals.

```
module com.example.Numbers {
  reference syntax { Integer ← /[0-9]+/; }
  role(evaluation) { ... }
}
slice com.example.NumbersSlice {
  concrete syntax from com.example.Numbers
  module com.example.Numbers with role evaluation
}
```

Listing 2: The slice that defines `Term` for sum

2 Background

In this section we present the tools that we are going to use in the description of our approach. As we already mentioned, we will employ Neverlang for the componentization of the language implementation, while we will use CVL for variability modeling and realization.

2.1 Neverlang

The *Neverlang* [2, 3] framework for DSL development promotes code reuse and sharing by making language units first-class concepts. In Neverlang, language components are developed as separate units that can be compiled and tested independently, enabling developers to share and reuse the same units across different language implementations.

In Neverlang the base unit is the **`module`** (Listing 1). A module may contain a **`syntax`** definition or a semantic **`role`**. A role defines actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique (for reference, see [1]). Syntax definitions are portions of BNF grammars, represented as sets of *grammar rules* or *productions*. Semantic actions are defined as code snippets that refer nonterminals in the grammar.

Syntax definitions and semantic roles are tied together using **`slices`**. For instance, module`neverlang.commons.AddExpr` in Listing 1 declares a reference syntax for sum, and actions are attached to the nonterminals on the right of the

```
language com.example.CalcLang {
  slices com.example.AddExprSlice com.example.MulExprSlice
         com.example.ParenExprSlice com.example.ExprAssocSlice
         com.example.NumbersSlice
  roles syntax < evaluation < ... // other roles
}
```

Listing 3: Neverlang's language construct.

two productions. Rules are attached to nonterminals by referring to their position in the grammar: numbering starts with 0 from the top left to the bottom right, so the first `AddExpr` is referred to as 0, `Term` as 1, the `AddExpr` on the second line would be 2 and so on. The slice `neverlang.commons.AddExprSlice` declares that we will be using *this* syntax (which is the **`concrete syntax`**) in our language, with that particular semantics.

Finally, the **`language`** descriptor (Listing 3) indicates which slices are required to be composed together to generate the interpreter or the compiler[3] for the language. Composition in Neverlang is therefore twofold:

1. between modules, which yields slices
2. between slices, which yields a language implementation

The result of the composition does not depend on the order in which slices are specified. The grammars are merged together to generate the complete parser for the language. Semantic actions are performed with respect to the parse tree of the input program; roles are executed in the order specified in the **`roles`** clause of the **`language`** descriptor. For lack of space, we cannot not give an in-depth description of Neverlang's syntax; for a more detailed description, see [3].

The set of generated components can be precompiled into JVM bytecode, and can be instantiated and queried for their properties using a specific API. For instance it is possible to retrieve the part of the syntax they define, the actions they include, etc. This API can be exploited to collect information from a given pool of slices.

In Neverlang the composition process is *syntax driven* and *implicit*. It is *syntax driven*, in that relations between slices are inferred from the grammar definitions that they contain. It is *implicit* in that these dependencies are implied by these definitions, and they are not stated in an explicit way. We will describe this with more detail in Sect. 4.

2.2 Variability Management and CVL

Variability modeling (VM) is a modeling approach in order to manage and express commonalities and differences in a family of products. These commonalities and differences are represented as features (particular characteristic or properties) of the family of products. Currently two approaches are possible,

[3] Although in the following we will take the liberty to always use the term *interpreter*, let it be known that Neverlang is perfectly capable of generating compilers

the first being that the underlying asset provides mechanisms to support extensions which are used to introduce variations; and the second approach is when the variability is expressed orthogonally to the asset. In the second approach a binding is required between the features and the asset. A feature model is a common approach to specifying the relationship between features defined as a set of constraints between features.
The *common variability language* (CVL)[4] [11] is a domain-independent language for specifying and resolving variability over any instance of any MOF-compliant metamodel. Inspired by feature models, CVL contains several layers. The Variability Abstraction Model (*VAM*) is in charge of expressing the variability in terms of a tree-based structure. The core concepts of the VAM are the variability specifications (*VSpecs*). The *VSpecs* are nodes of the VAM and can be divided into three kinds: Choices, Variables and Classifiers. The Choices are *VSpecs* that can be resolved to yes or no (through *ChoiceResolution*), *Variables* are *VSpecs* that requires a value for being resolved (VariableValue) and *Classifiers* are *VSpecs* that imply the creation of instances and then providing per-instance resolutions (*VInstances*). In this paper, we mainly use the *Choices VSpecs*, which can be intuitively compared to features, which can or cannot be selected during the product derivation (yes/no decision). Besides the VAM, CVL also contains a Variability Realization Model (VRM). This model provides a binding between the base model which contains the assets and the VAM. It makes possible to specify the changes in the base model implied by the *VSpec* resolutions. These changes are expressed as Variation Points in the VRM. The variation points capture the derivation semantics, i.e. the actions to perform during the Derivation. The CVL specification defines four types of variation points, namely *Existence*,*Substitution*, *Value Assignment* and *Opaque Variation Point*. An *object existence variation point* is used to determine when an object found in the base model should be included or not. Finally, CVL contains resolution models (RM) to fix the variability captured in the VAM. The RM replicates the structure of the VAM, in the case of the *Choice* it would become a *ChoiceResolution* which allows the choice to be either selected or not. Similarly *VariableValueAssignments* are used to assign values to variables. Thereby providing a mechanism to configure the features required in the desired product.

3 Approach Overview

As noted in the introduction, each component that we add to a language usually has some dependencies, such as a semantic concept, a syntactic requirement, or both of them. For instance, if we want some looping construct to terminate, be it **`for`**, **`while`**, or whichever we may pick, we might as well include some concept of *truth value* and the idea of a *condition* to test. Likewise, we would need some syntax to express this concept. Similarly, there might be concepts that, together, in the same language may *conflict*. For instance, we cannot have a three-valued

[4] CVL is currently a proposal submitted to OMG. Cf. `http://variabilitymodeling.org`.

logic and the simple boolean logic to just *coexist* in the same places: what if the condition of a loop evaluates to **`null`**? Should the loop exit or not?

Component-based language development is close to providing people with an easy way to implement a language by just selecting components, but implicit dependencies and conflicts between them creates a barrier to opening such development to a wider audience. The challenge lies in the fact that an in-depth knowledge of how the components are designed is required prior to using such an approach. Applying variability modeling to a modular language framework allows the explicit modeling of the relations between components in a manner understandable to the domain expert or end-user.

In our approach, component-based development is necessary for users to be able to selectively pick components; the feature model is necessary to represent how components may interact and to relieve users from the burden of satisfying complicated dependencies by hand. The variability model explicitly represents the constraints and the resolution model complies with these constraints, so that the result of the derivation is guaranteed to behave as expected. Typically a variability model is used to represent a family of products; in our case we will use it to represent a *language family*, that is a set of languages that share a common set of features. In a perfect world, language components would be developed from scratch, with the target variability model in mind, and therefore they would be guaranteed to compose well together. However, implementing a language from the ground requires a substantial investment. To minimize cost during component-based language development, one approach would be to maximize reuse of a set of already available language components.

We will focus on the case of Neverlang and CVL, but the approach that we present can be applied to any kind of feature modeling approach and any componentized language development tool that will fit our framework. In particular, the main requirement for the language framework is to support a way to define the language constructs in separate components. Although Neverlang includes some peculiar features [3], we believe that this approach can be applied by other modular language development frameworks (see Section 7 for other Neverlang-related work), provided that it is possible to extract from the language components the set of their relations (the *dependency graph*, see Section 4). The global approach is a two-level process: first, the reusable language components are capitalized and their possible combinations are captured in a variability model. Second, the variability model is used to select an expected set of features (or *configuration*) from which a woven model is produced by composition of the suitable reusable language components.

From a methodological perspective, we also distinguish two roles for users of our approach:

- **Language Developer**. A person experienced in the field of DSL implementation, and who knows how to break down a language into components.
- **Domain Expert**. A person that knows the concepts and the lexicon of the target domain. People in this category would also be end-users of the language.

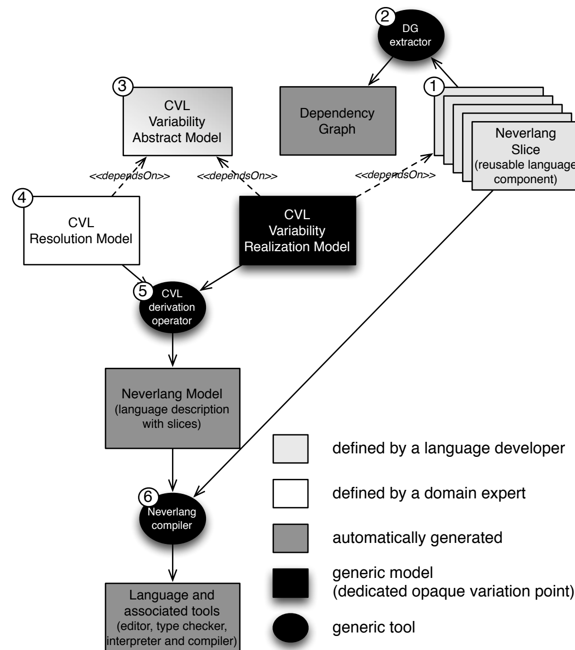


Fig. 1. CVL in Language Development

In practice, as illustrated in Fig. 1, the approach is divided into the following six steps:

① the *language developer* collects all of the available language components: these could be pre-existing components or newly created components.
② the relations between components are extracted automatically and represented as a *dependency graph*
③ the *language developer* and the *domain expert* collaborate to define a variability model using the dependency graph as a guide, in such a way as to define a language family most relevant for the given domain.
④ the *domain expert* becomes autonomous: it is now possible to extract a desired language by resolving the variability (selecting a set of features).
⑤ using a derivation operator, a list of composition directives is derived from the resolution of the variability model
⑥ the language development tool generates a complete interpreter/compiler for the desired language.

In our case, if Neverlang is the language framework, and CVL is the variability language, then we will implement the reusable language components (**`slices`**) using Neverlang (step ①) and extract the dependency graph from Neverlang (step ②); the specification of the variability (called *variability abstract model*) will use the choice diagram proposed by CVL (step ③); variability resolution will be CVL's *resolution model* (step ④); the composition directives (the **`language`**

descriptor) will be derived (step ⑤) using a dedicated derivation operator, implemented using the CVL opaque variation point (and included in the *variability realization model*); finally Neverlang (step ⑥) will compose the slices contained in the language descriptor. Please notice that Neverlang provides an additional degree of composition: composition between slices (possibly) yields a language, but, as described in Sect. 2, composition between modules yields slices. This additional degree of freedom will not be discussed here as it would go beyond the scope of this paper: code reuse at the module level would raise the problem of multi-dimensional variability, that we reserve to explore in future work.

4 From Slices to Variability Modeling

The *domain expert* and the *language developer* interact to implement the variability model and map it onto a pool of slices. In this section we show that a variability model can be reverse-engineered from language components. We will show a simple DSL to express arithmetical expressions (similar to the ARI language found in [12]) that, however, has the right level of complexity to explain our approach. The language of arithmetical expressions is known to be more complicated than it looks. For instance, the grammar is known to be non-trivial to factorize, and the semantics is hard to modularize (cf. "*the expression problem*" in [26]). In this known setting, we imagine that a *language developer* and the *domain expert* collaborate to implement a variability model on top of a set of slices that implement a family of ARI-like languages. For the sake of brevity, we will consider expressions that include only addition and multiplication over the domain of positive naturals; e.g.: $12 + 5 \times (4 + 2)$.

In our example, the language has already been developed using Neverlang, and *a pool of slices* is already available. In this context the variability model would be a representation of all the possible language variants that can be obtained from different subsets of this pool, and a *language family* (Sect. 3) would be seen as the set of languages that share a common set of slices. In particular, given this pool of slices, then the first step (Sect. 3) to design their variability model (Fig. 3) is to derive a *dependency graph* (Fig. 2).

From Slices to Dependency Graph. In Sect. 2 we briefly introduced the slices that implement the addition (Listing 1) and the definition of numbers (Listing 2), and we said that the composition process in Neverlang is *syntax-driven* and *implicit*. Slices are composed together automatically, because the

```
module neverlang.commons.ExprAssoc {
  reference syntax {
    Expr   <— AddExpr;
    Expr   <— ParenExpr;
    Term   <— MulExpr;
    Factor <— Integer;
  }
}
```

Listing 4: Traditional Associativity Rules.

nonterminals that their grammars contain already *implicitly* declare something about what they *require* and *provide*. For instance, consider the production for `com.example.Numbers`:

```
Term ⇐ /[0-9]+/
```

In this case, the right-hand side is a regex pattern, i.e., a *terminal symbol*: this is just a way to tell Neverlang's parser generator that the text of a program should contain a number, and has no implication on the way this slice composes with others. On the other hand, the *head* of the production (its left-hand nonterminal) represents something that the slice *makes available* to other slices. In other words, since this slice has `Term` in the head of its production, another production, possibly in another slice, may refer to it in its right-hand side. In this case, we might say that the slice `com.example.NumbersSlice` *provides* the nonterminal `Term`, which is bound to the high-level concept of *number* and *operand of a sum*. Similarly, a nonterminal occurring in the right-hand side of a production is predicating about what the slice *requires* to be available. For instance, in `com.example.AddExprSlice` we had:

```
AddExpr ⇐ Term
```

In this case, the *head* says that the slice *provides* `AddExpr`, but, at the same time, this slice *requires* `Term`. This constraint would be satisfied if `com.example.AddExpr` and `com.example.Numbers` were part of the same language.

These *implicit dependencies* are not enforced. Satisfying these constraints is left to the knowledge of the language developer. In Neverlang we have fostered support to variability modeling by adding a high-level API to simplify extraction of this data from a given slice. The result is that now slices can be queried for what we call their *provide set* —i.e., the collection of all the nonterminals that the slice defines— and for their *require set* —i.e., the collection of all the nonterminals that should be made available by other slices— in order for this slice to make sense in the language. For instance, the slice for the addition that we presented does not make sense alone, but rather another slice in the same language should define what a `Term` is; that is, `Term` should be found in the *provide set* of another slice. It is then quite natural, that, given a pool of slices, it is possible to derive a *dependency graph* depicting the relations.

The concept of dependency graph for a set of slices is quite intuitive, but more formally, we may say that, given a set (a pool) of slices $S = \{s_0, s_1, \ldots, s_n\}$, we define for each $s \in S$ two sets $R_s \subset N$, the *require set* and $P_s \subset N$, the *provide set*, with N being the alphabet of all the nonterminals in the grammars of all the slices in S. A *dependency* is a pair (s, X), where $s \in S$ and $X \in R_s$. We can say that the dependency (s, X) is *satisfied* if there is at least one slice $s' \in S$ such that $X \in P_{s'}$, and then that s' satisfies s. A *dependency graph* for a pool of slices can be then defined as a tuple $G = \langle S, D\rangle$, with S being the set of slices and $D = \{(s, s') \mid s' \text{ satisfies } s\}$, with a function $\ell(d) = X$ for each $d = (s, s') \in D$, such that (s, X) is a dependency satisfied by s'. For instance, given the pool of slices that constitutes the language in Listing 3, the dependency graph is shown

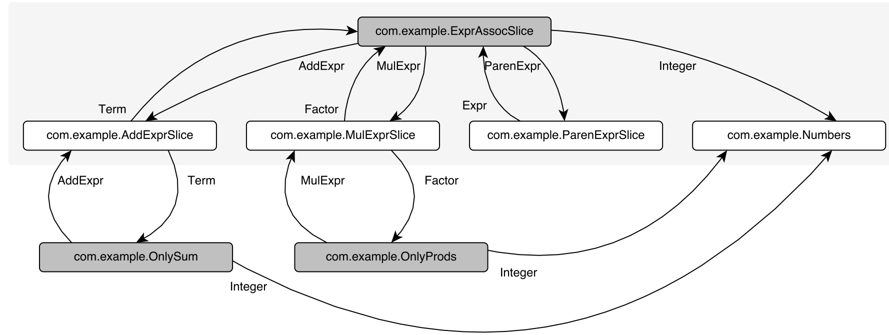


Fig. 2. A Slice Pool, including the ARI language

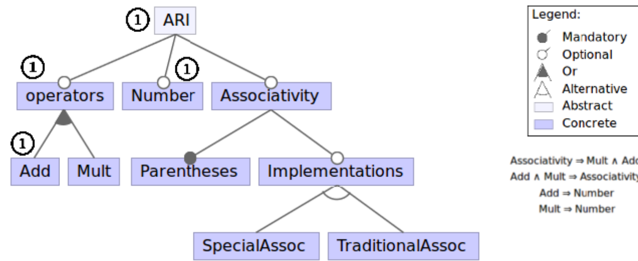


Fig. 3. VAM for the Expression Language

in Fig. 2, in the grey box. The arrows point in the direction of a dependency, and they are labeled by the nonterminal that represents that dependency. For instance, `com.example.ExprAssocSlice` (Listing 4) requires some slice to define the `MulExpr`, `AddExpr` and `ParenExpr` nonterminals. These dependencies are satisfied by the slices for multiplication, addition and parenthesized expressions, respectively; e.g., if $s =$ `com.example.ExprAssocSlice` and $X =$ `MulExpr`, then the dependency (s, X) is satisfied by `com.example.MulExprSlice`.

From Dependency Graph to VAM. The CVL variation model (VAM) presents a simple-to-use, feature-oriented view of a pool of Neverlang slices to the user. It encodes the set of choices and the constraints between choices. Constructing a VAM requires the collaboration between the language developer and the domain expert. The language developer has experience in using a language development tool (in our case, Neverlang) and can count on a code base of language components that he or his colleagues have developed over the years. Any arbitrary combination of slices from the pool of slices does not necessarily constitute a language. As each slice may have specific dependencies and conflicts may arise when combining certain slices. In order to establish such dependencies a *dependency graph* is provided. The DG Extractor is a tool that uses Neverlang to query a pool of slices and generate the corresponding dependency graph.

The language developer and the domain expert can exploit this graph as a basis to design the variability they intend to obtain implemented as a variability abstract model (VAM) (Sect. 2.2). The VAM uses a tree-based structure to define the dependency relationships between the features, typically as parent-child relations. For the purpose of this explanation we target the VAM in Fig. 3 using the dependency graph in Fig. 2.

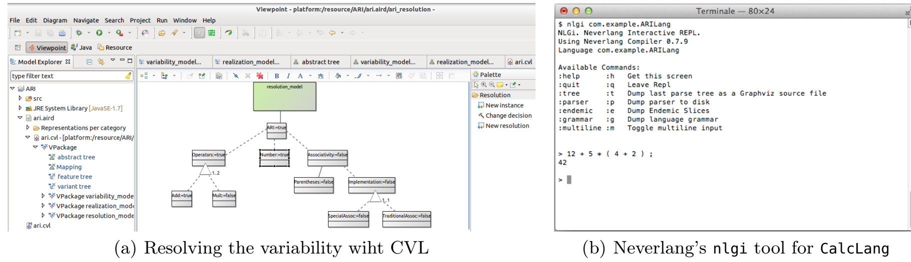
As we see in Section. 3, the join point between the domain expert and the language developer is the definition of the VAM. First of all, the language developer knows that each of the features must be realized by either one or more slices, thus the VAM could be constructed at first as having a root node with the branches being each of the slices. These node names can be then refactored in order to reflect appropriately the features they represent.

Identifying simple features. First of all, the domain expert is able to recognize that `AddExprSlice` concerns the higher-level concept of *addition* and that `MulExprSlice` concerns the higher-level concept of *multiplication*, which are both *operators*. Therefore, it is possible to re-organize the hierarchy in the VAM, by adding a parent node that groups the two, defined as the feature *operators*.

Compound features. Now, it is really apparent that the graph in Fig. 2 contains highly-connected components; there are nodes in the graph with a high number of inbound and outbound edges. These highly-connected components usually clusterize slices that are *all* required to implement some feature. For instance, the slice `OnlySums` and `AddExprSlice` depend on each other, and `OnlySums` depends on `Numbers`. A similar reasoning applies to `OnlyProds`, `MulExprSlice` and `Numbers`. Both of these highly-connected components show no dependency on `ParenExprSlice`. The *domain expert* has the knowledge to abstract away from the language components that are shown in the graph, and suggest that a language variant having only sums can be represented in Fig. 3 as the feature `Add_NoParenthesis` being one of the alternative of the feature `Add`. Similarly, the same thought process can be applied to the multiplication operation. In addition, the fact that the dependency graph shows that exists a dependency from feature `Add` and feature `Mult` to the feature `Numbers`, implies that we have a cross-cutting constraint that when any operator is selected the feature `Number` *must* be included.

Extra features. More information can be added. For instance, the domain expert might require that there exist another type of associativity, such as feature `SpecialAssoc`. In this case it is also evident that in the current set of slices this is not possible: this reflects the notion that building such a model with the domain expert can also highlight missing features in the language. It is also shown in Fig. 3 additional constraints —predicate logic statements— which would need to be captured by the domain expert and language developer.

For the ARI example, we obtain the VAM as depicted in Fig. 3. This VAM contains a root choice ARI with three optional features. The *operators* feature allows



(a) Resolving the variability wiht CVL (b) Neverlang's `nlgi` tool for `CalcLang`

Fig. 4. Example usage of the toolchain provided by CVL and Neverlang.

you to choose either addition (Add Feature) or multiplication (Mult Feature) or both can be included in a language. Since these features are also dependent on numbers, additional constraints are shown below the legend as predicate logic statements. Including associativity in the language requires that parenthesis is also included shown as a mandatory child of feature *Associativity*. Finally *implementations feature* represents alternatives in terms of the associativity implementations. The rationale for building a VAM from a dependency graph, provides a mechanism to ensure that dependencies are included and conflicts are avoided. Also when the pool of slices needs to be maintained the variability model provides an indication of what the impact could potentially be. The current implementation merely provides the dependency graph, and it is left as a manual process to completely define the VAM. In future work we intend to provide additional facilities to cope with such a task.

5 From Variability Modeling to Language Implementation

The *domain expert* would select a set of features in order to derive a desired language variant. In this section we provide details on how the process is automated in order to obtain a fully-functional language by selecting a set of features.

From VAM to Resolution Model. The Resolution model (RM) contains a set of choices defined as *Choice Resolutions* which corresponds to the features found in the VAM. In addition the RM respects the constraints defined by the VAM. In the implementation, we automatically generate a resolution model according to the VAM and its constraints (`cardinalities`, `isImpliedByParent`, `DefaultResolution`, ...). The domain expert can select or reject a feature by changing the choice resolution decisions and in the implementation a graphical tool is provided depicted in Fig. 4(a).

From a resolution model to a language description. The mapping between the features in the VAM and the Neverlang slices is defined in the CVL variability realization model (VRM). The VRM takes as input a RM which effectively contains the selected set of features. An *object existence variation point* (Sect. 2.2) is used to include or reject a slice. An *opaque variation point* (OVP) is a black box variation point whose behaviour is defined with an action language. In our CVL implementation, we currently support OVPs defined in Groovy3, in Javascript or

in Kermeta [16]. Using the OVP we define a dedicated derivation operator. This operator implements the semantics to generate a Neverlang language descriptor from a set of slices. The semantics of the CVL derivation process is extended to allow for ordering of the execution of variation points. The dedicated derivator operator has a lower precedence than *object existence variation point* thereby ensuring that all slices would be included prior to the generation of the language descriptor.

From **`language`** *descriptor to fully-functional language.* `nlgc` is a compiler provided by Neverlang, which translates the script generated by the VRM. `nlgc` creates the language by combining the pre-compiled pool of slices into a fully-functional language implementation, that is ready to be invoked at a command prompt with an input source file. The `nlg` tool can be invoked with the language name to start a minimal non-interactive interpreter that executes a program from a file input. Likewise, the `nlgi` tool starts an interactive interpreter that executes user-input programs shown in Fig. 4(b). An additional process or step is required in order to implement the variability model. However the benefit is that we have an explicit model of the relations between features of the set of languages. These features are also explicitly mapped to the slices in the VRM. These models can be exploited when the pool of slices need to be modified or new slices need to be introduced. In addition the resolution model provides a usable interface for the domain expert, who can immediately benefit by selecting a set of features and generating a desired fully-functional language. Which he/she can immediately test and use interactively or in batch mode, thereby allowing the domain expert to become completely *autonomous*.

6 Case Study: Family of Statemachines

Statemachines represented by statechart diagrams are typically used to model behavior. Over the years different implementations of statecharts have emerged, ranging from UML statechart diagrams, Harel's statechart and their object-oriented versions (implemented in Rhapsody). These implementations exhibit syntactic and semantic variations. In Crane *et al.* [7] a categorization is provided, highlighting the effects of such variations and the challenges in transforming from one implementation to another. Consider for example the pseudostate *fork* which would split an incoming transition into two or more transitions. In the case of classic statecharts simultaneous triggers/events can be handled; thus, in the *fork* implementation in the classic statechart the incoming and outgoing transitions support a trigger, shown in Listing 5(a) as `evt1`, `evt2` and `evt3`. In UML or Rhapsody, when an event arrives, the machine must complete the processing of such an event prior to accepting a new event known as *run-to-completion* (RTC) events. Using the statechart fork implementation in UML or Rhapsody statecharts would result in an ill-formed statechart, as they do not handle simultaneous events. In Listing 5(b) the UML implementation is shown: in this case we have removed the triggers on the outgoing transitions, which makes it compliant with UML as outgoing fork transitions may contain labels or actions. However this implemen-

```
statechart Classic {
  State: S1; State: S2; State: S3;
  State<Fork> : F1;
  Transition: T1 <S1,F> Trigger[evt1];
  ForkTransition: T2 <F,S2> Trigger[evt2] Effect[act1];
  ForkTransition: T3 <F,S3> Trigger[evt3] Effect[act2];
}
```

(a) Harel statechart

```
statechart UML {
  State: S1; State: S2; State: S3;
  State<Fork> : F1;
  Transition: T1 <S1,F> Trigger[e];
  ForkTransition: T2 <F,S2>  Effect[act1];
  ForkTransition: T3 <F,S3>  Effect[act2];
}
```

(b) UML statechart

```
statechart Rhapsody {
  State: S1; State: S2; State: S3;
  State<Fork> : F1;
  Transition: T1 <S1,F> Trigger[e];
  ForkTransition: T2 <F,S2>;
  ForkTransition: T3 <F,S3>;
}
```

(c) Rhapsody statechart

Listing 5: Textual DSL notation for the three kinds of Statecharts.

Implementation	Neverlang Slices						
	Statechart	State	Transition	ForkState	ForkTriggerEffect	ForkEffect	ForkNoActions
Classic statechart	✓	✓	✓	✓	✓		
UML statechart	✓	✓	✓	✓		✓	
Rhapsody statechart	✓	✓	✓	✓			✓

Table 1. Statechart implementations in relation to the slices

tation would still remain ill-formed for Rhapsody, as the fork is simply a split which is shown in Listing 5(c). In the next section a pool of slices in Neverlang is defined to support the fork implementations in the different statechart variants. The CVL derivation engine and the Neverlang implementation can be downloaded from their websites[5].

Step ① Implementation of the language components. In Neverlang, such a set of statecharts in Neverlang is defined as a set of slices. In the implementation, the statemachine supports simple states, transitions and the pseudostate fork. For the sake of brevity we merely show the syntax of the slices which would support the variations in the different fork implementation. The slice `ForkState` represents the fork pseudostate, which is supported by the module `ForkState`. The slice includes the syntax for `State<Fork>`, and includes the keyword `ForkTransitions`, that introduces the outgoing transition in a fork. Finally the nonterminal `ForkActions` represents the possible actions that can be used in the fork transitions. Depending on the implementation the syntax for the `ForkActions` would vary, as shown in Listing 6. Similarly, states and transitions are implemented as slices. Using these slices we can implement each variation.

Classic State Chart In this case we need to combine a set of slices that supports a simple statechart with a fork state, supporting simultaneous triggers.

[5] `people.irisa.fr/Suresh.Pillay/vm-neverlang` and `neverlang.di.unimi.it` respectively.

```
module ForkState {
    reference syntax {
        StateDef <- Fork;
        Fork <- "State<Fork>" ":" Identifier;
        TransitionDef <- "ForkTransition" "<" Identifier "," Identifier ">" "(" ForkActions ")";
}}
slice ForkState {
    concrete syntax from ForkState
    module ForkState with role evaluation
}

module ForkTriggerEffect {
    reference syntax { ForkActions <- Trigger "," Effect; }
}
slice ClassicForkActions {
    concrete syntax from SimultaneousTriggers
    module ForkTriggerEffect with role evaluation
}

module ForkEffect {
    reference syntax { ForkActions <- Effect; }
}
slice UMLForkActions {
    concrete syntax from RTCEffects
    module ForkEffect with role evaluation
}

module ForkNoActions {
    reference syntax { ForkActions <- ""; }
}
slice RhapsodyForkActions {
    concrete syntax from RTCEffects
    module ForkNoActions with role evaluation
}
```

Listing 6: Slices and modules to support variations in Fork implementations.

In this case we would combine the slices for the simple statemachines together with the slices `ForkState` and `SimultaneousTriggers`.

UML In the case of UML we would use `ForkState` and `RTCEffects` as this kind of graph supports RTC with labels or effects.

Rhapsody Finally, in the case of Rhapsody we can simply use the simple statemachine together with `ForkState` as there is no need for fork triggers or effects.

A summary of the possible choices is represented in Table 1. Using such a set of slices we can support the different language variants, and the right combination of slices is automatically generated according to the domain expert choices.

Step ② Dependency Graph. Figure 5 shows the dependency graph extracted from our pool of slices. Included in the pool is the states, transitions and the pseudostate fork slices that would support the different implementations (classic, UML and Rhapsody). The language developer and the domain expert can clearly see that they have slices for representing states and transitions. The transition supports three options `Trigger`, `Guard` and `Effect`. Using this set of slices, it is also possible to represent a feature to support the pseudostate fork. In addition, the different fork implementations reflect that the variability model should cater for the variations to support classic and UML statechart.

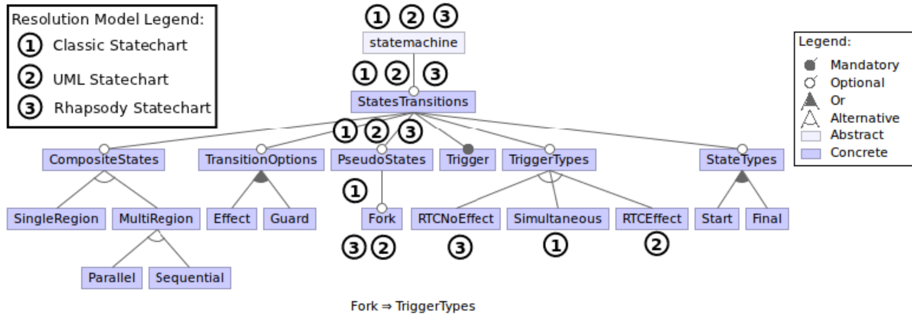


Fig. 6. Variability Model for a Family of Statemachines

Partitioning the slices in features correctly requires domain knowledge. The major difference between the three implementations, with respect to the fork pseudostate, is a result of how events are handled, either simultaneously or RTC. From the dependency graph, it is only possible to infer that each implementation supports the different `ForkActions`. However only leveraging on the knowledge of the domain expert, it is possible to decide to model this against the type of event system adopted by the given implementation. The dependency graph provides some guidance towards reaching a VM, however it still requires human intervention.

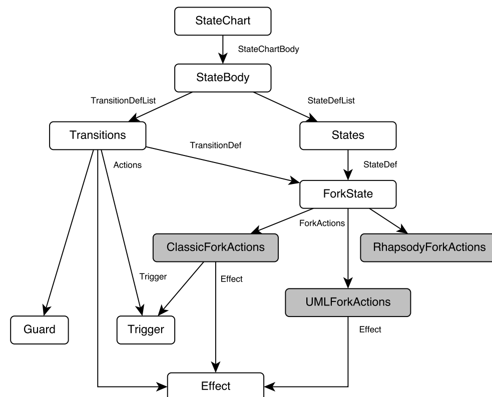


Fig. 5. Dependency graph for a family of statemachines

Step ③ Variability Model. Figure 6 shows the VAM for a family of statemachines, and shown in the legend it can be seen that different relations (e.g., *or*, *alternative*) can be modeled in such a structure. The focus is on the pseudostate fork part of the variability model. It is possible to choose the feature *fork*, which has a dependency on feature *TriggerTypes*. Feature *TriggerTypes* imposes an alternative between the *ForkActions* being either simultaneous triggers or RTC triggers with or without effects. Knowing how triggers/events are handled provides sufficient information to implement the fork correctly. The VRM allows us to map the features to the slices. These variation points are defined as *Object Existence* variation points in the VRM, which links the *feature* to the slice or slices which would implement the requirements of such a *feature* in the language. In Table 2 the mapping between the features and the slices supporting the fork variations is listed.

Step ④ Resolution of the variability by selecting a set of features. CVL provides a resolution model which is used to select or reject a feature. In Fig. 6 we

```
language Classic {
  slices
    Program States Transitions
            ...
    ForkState ClassicForkActions
  roles syntax < evaluation
}
```

(a) Resolution 1

```
language UMLSC {
  slices
    Program States Transitions
            ...
    ForkState UMLForkActions
  roles syntax < evaluation
}
```

(b) Resolution 2

```
language Rhapsody {
   slices
     Program States Transitions
             ...
     ForkState RhapsodyForkActions
  roles syntax < evaluation
}
```

(c) Resolution 3

Listing 7: Neverlang **`language`** descriptor for (a) classic, (b) UML, (c) Rhapsody statechart

show three sets of possible feature selections. The numbers 1, 2 or 3 represent the configurations for the different implementations. Using such a set of selections a desired statechart can be defined, 1 represents a classic statechart, 2 a UML statechart and 3 the Rhapsody statechart.

Step ⑤ and ⑥ Derivation of the composition directives and generation of the language implementation. The derivation process extracts the slices corresponding to the features selected in the resolution model, and applying the *opaque variation point* (see Sect. 5) the Neverlang **`language`** descriptor is generated. In Listing 7(a), 7(b) and 7(c) the **`language`** descriptors for resolution model 1, 2 and 3 is shown. The Neverlang compiler then composes together the slices that the **`language`** descriptor lists, and the result is an executable interpreter for the language variation that the user had requested.

7 Related Work

This section discusses related work on language design and implementation, and variability modelling approaches.

Feature	Slice	Feature	Slice
Statemachine	`Program`	Fork	`ForkState`
StatesTransitions	`States,Transition`	Simultaneous	`ClassicForkActions`
Guard	`Guard`	RTCEffect	`UMLForkActions`
Effect	`Effect`	RTCNoEffect	`RhapsodyForkActions`

Table 2. Mapping Features to Neverlang Slices

As we noticed in Section 3, the approach that we presented is general and can be applied even to other frameworks that support modular language implementation. Several authors explored the problem of modular language design (e.g., [10,13,23,24]). For example, LISA [13] and Silver [23] integrate specific formal grammar-based language specifications supporting the automatic generation of various language-based tools (e.g., compiler or analysis tools). One practical obstacle to their adoption is a perceived difficulty to adopt specific grammar-based language specifications, and the relative gap with the development environments used daily by software engineers. JastAdd [10] combines traditional use of higher order attribute grammars with object-orientation and simple aspect-orientation (static introduction) to get better modularity mechanism. To develop our work, we chose Neverlang [2,3], a language development framework that focuses on modularity, and reuse of pre-compiled language modules for the JVM. In Neverlang, language components are compiled into regular JVM classes that can be instantiated and inspected using a public API, to retrieve rich, structured information. This is a departure from the classic source-based analysis found in other tools, and makes Neverlang's core easier to plug into a higher-level workflow, such as the one we described.

Many formalisms were proposed in the past decade for variability modeling. For an exhaustive overview, we refer the readers to the literature reviews that gathered variability modeling approaches [5, 14, 20, 21]. All formalisms for variability modeling could be used following the approach we introduce in this paper. In our case, we use the choice diagram proposed by CVL, very similar to an attributed feature diagram with cardinalities.

Several works [8,12,22,28] have highlighted the benefits of coupling language development and variability approaches. Czarnecki [8] has shown that DSL implementation can be improved by employing feature description languages. Van Deursen *et al.* has proved the usefulness of their text-based Feature Description Language (FDL) using DSL design as a case study. In the work by Haugen *et al.* [12], the authors show how the features of DSLs such as the ARI language (an expression language similar to the one described in Sect. 4), the Train Control Language (TCL) and even UML can be modeled to design possible variations using CVL. White *et al.* [28] have demonstrated how feature modeling can be used to improve reusability of features among distinct, but related languages.

More recently, some work has applied variability management to language implementation. MontiCore [17] modularizes a language by extension. Extension is achieved by inheritance and language embedding. In [4] a variability model is used to manage language variants. In our case we focus on the reuse of existing components, not only varying a base language. Langems [27] uses role-based metamodeling in order to support modularization of languages. The roles play the role of interfaces and bind to concrete classes for the implementation. The concrete syntax is bound to the abstract syntax. A more restrictive version of EMFText is used to try to avoid ambiguities when the grammar is composed following the abstract syntax. However it is left to the language developer to avoid such conflicts. In [18] a family of languages is decomposed in terms of

their features. The grammar is constructed using SDF and the semantics is implemented using re-writing rules in Stratego. However, in this work the focus is on the language developer, who implements the language components without any assistance from a domain expert. Their approach is bottom up, but they do not start from a set of pre-defined component, but rather they componentize an already existing language and develop the variability model to support it. Therefore, the relations between language components are imposed by the developers as they implement them, while in our approach we rely on the existing dependencies of the language components to direct the implementation of the VM using an intermediate artifact (the *dependency graph*, Sect. 4).

8 Conclusions and Future Work

Applying variability modeling techniques to language development bridges the gap between the language developer and the domain expert. The variability model not only represents the features of the domain and their relations, but also the relation between the features and the language components. In our approach, the dependency graph provides a useful artifact to direct the construction of the variability model. This graph helps the domain expert to recognize possible language variants, and assists the language developer in finding possible shortcomings in the implementation of language components. A dedicated derivation operator is provided to allow the domain expert to automatically generate a language implementation, supported by an interactive interpreter, without further assistance from the language developer. In future work we intend to provide a set of operators to fully-automate the implementation of a variability model from a pool of existing language components.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.
2. W. Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proc. of SC'12*, LNCS 7306, pp. 162–177, Prague, Czech Republic, June 2012.
3. W. Cazzola and E. Vacchi. Neverlang 2: Componentised Language Development for the JVM. In *Proc. of SC'13*, Budapest, Hungary, June 2013. Springer.
4. M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within Modeling Language Definitions. In *Proc. of MoDELS'09*, LNCS 5795, pp. 670–684, Oct. 2009.
5. L. Chen and M. Ali Babar. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Journal of Information and Software Technology*, 53(4):344–362, Apr. 2011.
6. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Aug. 2001.
7. M. L. Crane and J. Dingel. UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *Proc. of MoDELS'05*, LNCS 3713, pp. 97–112, Montego Bay, Jamaica, Oct. 2005. Springer.

8. K. Czarnecki. Overview of Generative Software Development. In *Proc. of UPP'04*, LNCS 3566, pp. 326–341. Springer, 2004.
9. S. Efftinge and M. Völter. oAW xText: A Framework for Textual DSLs. In *Proc. of the EclipseCon Summit Europe 2006 (ESE'06)*, Esslingen, Germany, Nov. 2006.
10. T. Ekman and G. Hedin. The JastAdd System — Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, Dec. 2007.
11. F. Fleurey, Ø. Haugen, B. Møller-Pedersen, A. Svendsen, and X. Zhang. Standardizing Variability — Challenges and Solutions. In *Proc. of SDL'11*, LNCS 7083, pp. 233–246, Toulouse, France, July 2011. Springer.
12. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proc. of SPLC'08*, pp. 139–148, Limerick, Ireland, Sept. 2008. IEEE.
13. P. R. Henriques, M. J. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic Generation of Language-Based Tools Using the LISA System. *IEE Proc. — Software*, 152(2):54–69, Apr. 2005.
14. A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A Preliminary Review on the Application of Feature Diagrams in Practice. In *Proc. of VaMoS'10*, pp. 53–59, Linz, Austria, Jan. 2010. Universität Duisburg-Essen.
15. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proc. of ICSE'11*, pp. 471–480, Hawaii, May 2011.
16. J.-M. Jézéquel, O. Barais, and F. Fleurey. Model Driven Language Engineering with Kermeta. In *Proc. of GTTSE'09*, LNCS 6491, pp. 201–221, Braga, Portugal, 2009.
17. H. Krahn, B. Rumpe, and S. Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, Sept. 2010.
18. J. Liebig, R. Daniel, and S. Apel. Feature-Oriented Language Families: a Case Study. In *Proc. of VaMoS'13*, Pisa, Italy, Jan. 2013. ACM.
19. M. Mernik and V. Žumer. Incremental Programming Language Development. *Computer Languages, Systems and Structures*, 31(1):1–16, Apr. 2005.
20. K. Pohl and A. Metzger. Variability Management in Software Product Line Engineering. In *Proc. of ICSE'06*, pp. 1049–1050, Shanghai, China, May 2006. ACM.
21. R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey. *Journal of Information and Software Technology*, 52(3):324–346, Mar. 2010.
22. A. van Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technolog*, 10(1):1–17, 2002.
23. E. van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming*, 75(1-2):39–54, Jan. 2010.
24. E. van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Proc. of CC'02*, LNCS 2304, pp. 128–142, Grenoble, France, Apr. 2002. Springer.
25. M. Völter and V. Pech. Language Modularity with the MPS Language Workbench. In *Proc. of ICSE'12*, pp. 1449–1450, Zürich, Switzerland, June 2012. IEEE.
26. P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.
27. C. Wende, N. Thieme, and S. Zschaler. A Role-Based Approach towards Modular Language Engineering. In *Proc. of SLE'09*, LNCS 5969, pp. 254–273, Oct. 2009.
28. J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt. Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE Software*, 26(4):47–53, July 2009.