Automating Variability Model Inference for Component-Based Language Implementations

Edoardo Vacchi, Walter Cazzola, Benoit Combemale, Mathieu Acher

▶ **To cite this version:**

Edoardo Vacchi, Walter Cazzola, Benoit Combemale, Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. SPLC'14 - 18th International Software Product Line Conference, Sep 2014, Florence, Italy. hal-01023864

HAL Id: hal-01023864
https://inria.hal.science/hal-01023864

Submitted on 15 Jul 2014

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.

Automating Variability Model Inference for Component-Based Language Implementations

Edoardo Vacchi, Walter Cazzola
Computer Science Department
Università degli Studi di Milano, Italy
lastname@di.unimi.it

Benoit Combemale, Mathieu Acher
University of Rennes 1
IRISA / Inria, France
firstname.lastname@irisa.fr

ABSTRACT

Recently, domain-specific language development has become again a topic of interest, as a means to help designing solutions to domain-specific problems. Componentized language frameworks, coupled with variability modeling, have the potential to bring language development to the masses, by simplifying the configuration of a new language from an existing set of reusable components. However, designing variability models for this purpose requires not only a good understanding of these frameworks and the way components interact, but also an adequate familiarity with the problem domain.

In this paper we propose an approach to automatically infer a relevant variability model from a collection of already implemented language components, given a structured, but general representation of the domain. We describe techniques to assist users in achieving a better understanding of the relationships between language components, and find out which languages can be derived from them with respect to the given domain.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.3.4 [**Programming Languages**]: Processors—*Compilers,Interpreters*

Keywords

Variability Models, SW Product Lines, DSL Implementation.

1. INTRODUCTION

Domain-specific languages (*DSLs*) are programming languages that target specific problem areas using terms and concepts that pertain to those domains. DSLs enable the domain experts to express solutions to their problems by using concepts related to their expertise.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPLC'14 September 15-19, 2014, Florence, Italy.
Copyright 2014 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

In the last few years, the practice of developing problem-tailored languages has been rediscovered [19], as a means to develop domain-specific solutions in an easier way. It is widely recognized [20, 23] that DSLs provide a better way to solve a domain specific problem (where better is in term of time-to-market, readability and maintainability). In this perspective, several small languages have been developed but this practice is not diffuse as expected since DSL development is an activity that requires time (often the development must be done from scratch) and skills about language engineering not always within everyone reach. Many modern language development frameworks are improving reuse and sharing language components among different implementations (e.g., [24, 7, 28]) to foster DSL development. Earlier work [32, 30] has shown that it is possible to construct a *family of languages*, i.e., a set of languages that can be built from different configurations of the same set of components.

Still, even though many language frameworks support component reuse and sharing, they often do not account for *variability and dependency management*. Variation points (features), dependencies between them and their mapping onto concrete artifacts are not made explicit or documented. A manual identification of variation points —and constraints among them— by the developer runs the risk of *over-* or *under*-estimating the capabilities of the DSL. It is easy for a developer to forget a technical constraint or to add an unnecessary variation point. On the one hand, an over-estimation leads to unsafe variants of a DSL, i.e., some variants rely on missing foundations. On the other hand, an under-approximation does not exploit the full range of features offered by a DSL.

In this work we describe an automated approach that applies variability management to componentized language development given a representation of the domain. We show how to *discover* and *present* the relations that occur in a given set of language components in an explicit, understandable, accurate way: a feature model that end users may use to perform choices and derive automatically a safe conforming language implementation. Developers of a DSL can also benefit from having a central and explicit feature model.

The contributions of this work are i) an approach to automatically derive a tree structure out of the language components using domain concepts found in a semantic network and hierarchical clustering algorithms; ii) techniques to automatically and interactively refine the feature model for enhancing its readability and enforcing its configuration set; iii) an evaluation of the proposal on two DSLs. We report on our experience and show that a substantial part of the

feature model (including feature hierarchy and constraints) necessitates domain knowledge.

The rest of this paper is structured as follows. In Sect. 2 we provide some background with respect to 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 using a running example (a family of state machine languages). In Sect. 5 we evaluate the approach on two case studies, the first based on the running example, the other represents a family of programming languages implementing the Linda [15] coordination model. In Sect. 6 and 7 we describe some related work and draw our conclusions.

2. BACKGROUND AND MOTIVATION

In this section we give an overview of the background that is required to understand the following of this paper.

2.1 Component-based Language Framework

A component-based language framework is a language implementation framework that emphasizes the separation of the concepts of a language as pluggable and composable units. Each unit usually represents a syntactic feature of the language (a keyword, or a construct) along with the implementation of its semantics. Many component-based language frameworks have been proposed over the years (e.g., [21, 34, 24]).

In this work we employed Neverlang [7, 8, 33]. In Neverlang, a reusable language component is called a **`slice`**. Each slice includes a syntax definition, that is a portion of a grammar of the language in BNF and a definition of several **`roles`**, that is, the implementation of a compilation phase, with respect to that part of the syntax. All the roles together represent the *semantics* of the construct. A language can be therefore seen as the result of the composition of several **`slice`**s. Grammar portions are sets of *production rules* (or, in short, *productions*) of the form $A \to \omega$, where A is called a *nonterminal* and ω is a string of *terminals* and nonterminals. A terminal can be roughly equated to a keyword or a literal of the language.

Language Components and Dependency Graph. When generating a language, the union of all the sets of productions contained in the slices should produce a "meaningful" grammar, that is, its generated language should not be empty. In concrete terms, consider a slice s, defining the production $A \to B$. Then there must be some slice s_A that *refers* nonterminal A in the right part of one of its productions (e.g. $X \to A$), and that, ultimately, there is a chain of references that leads to the starting symbol of the grammar. Similarly, we expect another slice s_B to include some rule of the form $B \to \beta$ such that finally $A \Rightarrow^* w$, with w being a syntactically-correct program. From these simple observations, it has been shown in a previous work [32] that production rules contained in a language component can be used to infer a set of *required* nonterminal definitions (the *require*-set) and a set of *provided* nonterminal definitions (the *provide*-set). Required nonterminals are all those that occur in the right-hand part of a production, while provided nonterminals are all those that occur in the left-hand part. Then, given slice s with production $A \to B$, we can say that s is *providing* a rule for nonterminal A, but it *requires* B; that is, another slice in the language should provide B.

Now, let us have a set 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 set 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 an example, see Fig. 6).

2.2 Problem Statement

Given a set of components languages (slices), we can automatically derive a dependency graph. Yet we are far from obtaining a comprehensive feature model:

- Features do not necessarily correspond to assets (components languages): the mapping between "slices" and features is not necessarily one-to-one. Typically abstract features can be added to reinforce the structure of the model.
- The dependency graph is not a tree: a node in the graph may have many possible parents whereas only one parent should be chosen.
- More dependencies —beyond implication constraints expressed in the dependency graph— are likely to occur such as excludes, disjunction, or bi-implications constraints. The dependencies can be reified directly in the feature diagram (e.g., with *alternative* groups) or as cross-tree constraints.

Overall automated support is needed to help (1) structuring the model and (2) enforcing its configuration set.

Notice that, in the rest of the paper we will assume a feature model to be a tree-like structure that represents relationships between features. Figure 9 represents a feature model for a family of state machine languages: relationships between a parent feature and its child features can be *optional* (represented with an empty circle, e.g. `Trigger`), *mandatory* (black-filled circle, e.g. `TransitionOption`), *or* (filled triangle e.g. `Guard` or `Effect`) and *alternative* (empty triangle, e.g., `SingleTriggerFork` vs `MultiTriggerFork`).

3. OVERVIEW OF THE APPROACH

Our goal is to synthesize a feature model from a set of components languages (slices). The basic idea is that to build the most relevant feature model for a given domain, from a set of language components such as slices, we can exploit part of the information that we find in these components, coupled with an encoding of the domain knowledge in the form of a semantic network. All the slices *provide* nonterminals, which are themselves words. We put a *meaning* to words and establish relations with *other words* that belong to the same domain, thus mapping a slice to a set of words.

It is then possible to apply an *agglomerative hierarchical clustering algorithm* to the slices associated to each set of words; by making an appropriate choice for the similarity measure, the *dendogram* (a binary tree) that is obtained as a result of the clustering procedure can be used as a first, rough approximation of the feature model. Additional kinds of relationships are used to further organize the features and a series of edits refine the intermediate result so that we can obtain a final feature model. The process is as follows:

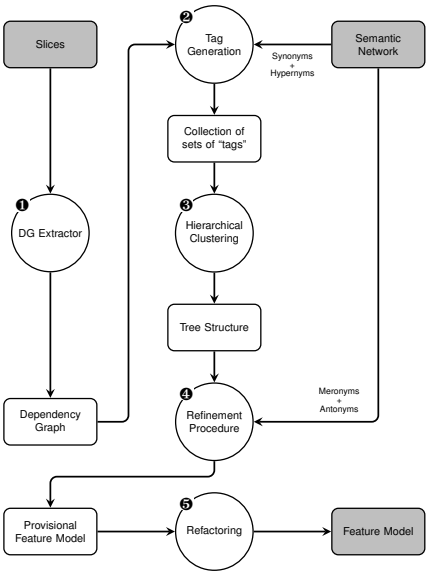

Fig. 1: Overview of the approach

1. At the first step, we extract the *dependency graph* from the set of slices (see Sect. 2);
2. At the second step, domain concepts from a *semantic network* (developed by a domain expert) are attached to the dependency graph with the aim to provide some semantic information that will be exploited to further structure the graph.
3. We feed the decorated dependency graph to a hierarchical clustering algorithm with an appropriate distance metric: the result is a *tree structure* which will serve as the basis for the feature model;
4. The tree structure is refined using:
   - An heuristic to infer (1) implications over a disjunction of features, and (2) alternative choices from the implemented components;
   - The *antonym* relation of the semantic network that can be used to find other alternative choices in the variability model;
   - The *meronym* relation describes mandatory relations between components.
5. In case, *synthesis techniques* can be applied to the *provisional model* to get a *final feature model*.

The result is a feature model that represents the family of languages that can be implemented in the given domain. The approach benefits from the skills of language and domain experts individually without forcing any of them to acquire the skills of the other. Moreover, the semantic network is independent of the language implementation and therefore reusable in different contexts (e.g., in case of multiple-domains) or different component language frameworks.

4. SUPPORTING THE APPROACH

In this section we detail and illustrate the automated techniques that support the approach on a running example: the family of state machine languages.

State machines represent the behavior of a system, described as a collection of a finite number of states. Several forms of state machines exist, Crane *et al.* [11] provide a categorization of the different variants. The full list of the language components that have been implemented with their

```
statechart Example :
  State: SW initial;
  State: SA;
  State: SB;
  State: SF final;
  ForkState: FS {
   Fork: Transition { tf };
   LeftState: SA RightState: SB
  };
  JoinState: JS {
    Join: Transition { tj };
    LeftState: SA RightState: SB
  }
Transition(SW,FS);Transition(JS,SF);
```

Listing 1: An example state machine written using one of the languages in the family.

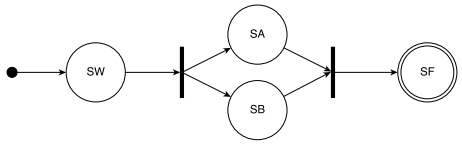

Fig. 2: The state machine described by Listing 1

description can be found in Sect. 5.1.

Listing 1 shows a program written using a language of the family and Fig. 2 shows the graphical representation of the same state machine.

4.1 Tag Generation

We already have *logical information* (implies constraints) between slices, coming from the dependency graph. The implications can be exploited to organize slices into a hierarchy, but are not sufficient: slices are merely symbols. Intuitively user intervention is needed to further refine the meaning of each slice and to hierarchically organize slices. The *semantics* is what is still missing, i.e., the relations that occur between the domain concept that each slice represents.

For establishing relationships, domain experts elaborate a domain-specific semantic network with *synonym* (syn), *antonym* (ant), *hypernym*[1] (hyper) and *meronym*[2] (mero) relations. Slices are lexically mapped onto domain concepts: each slice is associated to a set of terms, or *tags* that describe what it represents conceptually. These sets of tags are automatically generated from an *initial* set of words. Let be W a set of words. Then, if slice $s \in S$ (where S is an arbitrary set of slices), *provides* nonterminal X, we can define the initial set $T_0(s) = \{w_X, w_s\}$ to the slice, where $w_X \in W$ is that word $w_X \equiv X$, and $w_s \in W$ is the name of the slice. For instance, consider slice `OuterCompositeStates` in Fig. 6; because the slice *provides* nonterminal `StateDef`, as we can see from the labeled inbound edge, the initial set of tags for this slice will be:

$$T_0(\texttt{OuterCompositeStates}) = \{\texttt{StateDef}, \texttt{OuterCompositeStates}\}$$

Starting from the collection $T_0(s)$, we can find a superset $T(s) \supseteq T_0(s)$ of related words, by the help of a *semantic network* $N = \langle W, R \rangle$, where R is the set of binary relations between words in W : $R = \{\mathsf{syn}, \mathsf{ant}, \mathsf{hyper}, \mathsf{mero}\}$.

For each slice, we want to find the set $T(s)$ of words that describe that slice, therefore we can consider all the relations in $R' \setminus \{\mathsf{ant}\}$[3]. We drop ant because for every pair $(w, w') \in R_k$, with $R_k \in R'$, w' *describes* in some sense what the concept w *is* whereas the relation ant describes what the concept w *is not*. Then for each slice s there is a maximal

[1] $(A, B) \in \mathsf{hyper}$ if every B is a kind of A; e.g., *animal* is a hypernym of *dog*.
[2] $(A, B) \in \mathsf{mero}$ if every B is a part of A; e.g., *page* is a meronym of *book*.
[3] Antonyms will be used later in the process (Sect. 4.5).

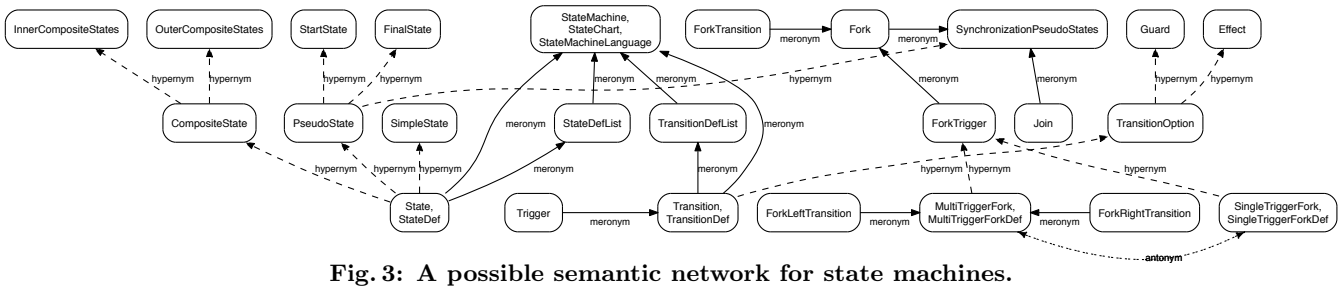

Fig. 3: A possible semantic network for state machines.

set $T(s) \subseteq W$ s.t.:

1. $T_0(s) \subseteq T(s)$
2. $w \in T(s) \implies \exists r \in R', \exists w' \in T(s) : (w, w') \in r$

For instance, $T(\texttt{OuterCompositeStates})$, using the semantic network in Fig. 3, would be:

$$T(\texttt{OuterCompositeStates}) = \{\texttt{StateDef}, \texttt{OuterCompositeStates}, \texttt{CompositeState}, \dots\}$$

4.2 Hierarchical Clustering

Once tags have been generated for each language component (slice), a relevant tree structure —that will serve as the basis for the feature model— should be derived. The idea is to clusterize our language components (the slices) by applying an agglomerative hierarchical clustering algorithm [31]. The algorithm generates a binary tree (called *dendogram*).

Specifically we reasoned by analogy with documental collections where each document can be seen as a set of words. Each word is associated with a frequency f, i.e., the number of occurrences of the word in the document. Our set of words $T(s)$ for slice s can be seen as a document where each word only occurs once. The clustering algorithm works by comparing clusters using a *similarity measure*. A reasonable measure for similarity between slices is the *Jaccard similarity* [31]:

$$J(s_1, s_2) = \frac{|T(s_1) \cap T(s_2)|}{|T(s_1) \cup T(s_2)|} \tag{1}$$

For instance the Jaccard similarity between $s = \texttt{InnerCompositeStates}$ and $s' = \texttt{OuterCompositeStates}$ with $T(s) = \{\texttt{InnerCompositeStates}, \texttt{CompositeState}, \texttt{StateDef}, \texttt{StateMachineLanguage}\}$ and $T(s') = \{\texttt{OuterCompositeStates}, \texttt{CompositeState}, \texttt{StateDef}, \texttt{StateMachineLanguage}\}$ would be $J(s, s') = .6$, and the distance can be defined as $d(s, s') = 1 - J(s, s') = 0.4$, that is the *complement* of the similarity measure. Within this framework, we can recursively define a cluster as:

1. the singleton set $\{T(s)\}$, with s being a slice
2. the set $\{c_1, c_2\}$ where c_1 and c_2 are clusters

The output of this process is a dendogram where all the leaves are clusters on one element of the form $\{T(s)\}$, that are therefore easily mapped to single slices. Each cluster contains the slices that are closer to each other, with respect to the Jaccard similarity measure. In some sense, then, each cluster contains the slices that are closer to each other *semantically*: in fact, the more two sets $T(s)$, $T(s')$ overlap, the closer the measure will be to 1. Figure 4 shows the result of the hierarchical clustering on a small subset of slices for the state machine language example.

4.3 Refinement Procedure

In this first approximation, the tree (Fig. 4) exhibits many nodes labeled in the same way and it is poorly structured. Now we describe how to merge nodes and compute labels.

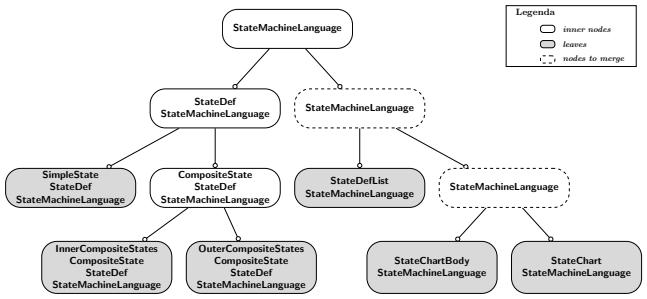

Fig. 4: (Part of) Dendogram for the State Machines.

Merging nodes. To reduce the number of choices, we merge nodes according to the chosen distance measure d. If the distance between a pair of nodes is zero, then they must be merged. Once the nodes have been merged, they can be labeled with the tags contained in the clusters. The result of the clustering procedure is a binary tree $H = \langle C, E \rangle$, where C is a set of clusters and the set of edges is $E = \{(c, c') \mid c' \in c\}$. In particular, there is a subset $C_S \subset C$ that is the collection of 1-element clusters, i.e., $C_S = \{\{T(s)\} \mid s \in S\}$; C_S is the set of the tree *leaves*. For each cluster c we define:

$$\tau(c) = \begin{cases} T(s) & c \in C_S \\ \tau(c_1) \cap \tau(c_2) & c = \{c_1, c_2\} \end{cases} \tag{2}$$

Intuitively, the τ function flattens the word sets found in each cluster, and computes their intersection. E.g., consider a cluster $\bar{c} = \{\{\texttt{StateChart}, \texttt{StateMachineLanguage}\}, \{\texttt{StateChartBody}, \texttt{StateMachineLanguage}\}\}$, then $\tau(\bar{c}) = \{\texttt{StateMachineLanguage}\}$ (cf. Fig. 4). The τ function and the Jaccard similarity are used to find parent/child pairs that can be merged. For each parent/child pair $(c, c') \in E$ we compute the similarity value:

$$J\left(\tau(c), \tau(c')\right)$$

where J is still the Jaccard similarity. When the similarity value is 1 (the distance is 0), then parent and child can be merged, i.e., children of c' may become children of c, and node c' may be removed from the tree.

Labeling nodes. The result of the merging procedure is a non-binary tree where nodes are still unlabeled. Using again the τ function we can now define a labeling strategy ℓ, *i.e.*, the labeling where parent and child labels do not overlap:

$$\ell(c) = \begin{cases} \ell(c) & c \text{ is root} \\ \tau(c) \setminus \ell(c') & (c', c) \in E. \end{cases} \tag{3}$$

The result of the entire procedure applied to Fig. 4 can be seen in Fig. 5 (Red, dashed edges are implies constraints that can be added by a user later on, see Sect. 4.5)

4.4 Heuristics for Mining Constraints

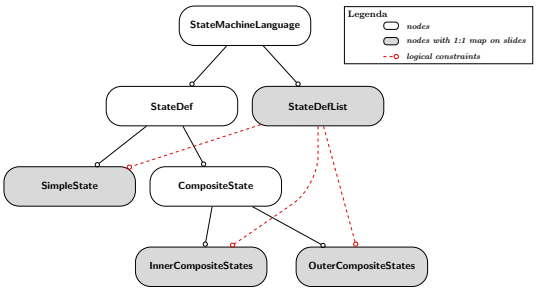

Fig. 5: Structure of the tree with respect to the features that represent states.

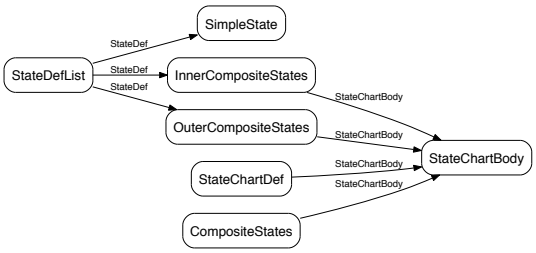

Fig. 6: The dependency graph for a subset of the state machine language family

The dependency graph $G = \langle S, D \rangle$ represents implication relations that we can extract from the concrete artifacts. Figure 6 shows the dependency graph for a subset of the slices that we are considering for the state machine example. There are two additional opportunities for mining other kinds of constraints (beyond binary implications).

Implication and disjunctions. For each vertex s in the dependency graph G there exist a pair $(s, s') \in D$ if and only if there is an s' that satisfies s (Sect. 2.1), where $s, s' \in S$ are slices. In particular, as we saw previously, s' satisfies s when there is some production $B \to \beta$ in s' and s contains some production $A \to \omega B \omega'$. In this case, we say that s' *provides* B and that s *requires* A. The *satisfies* relation between slices s, s' can be therefore interpreted as the logic constraint:

$$s \to s' \tag{4}$$

However, consider the case when there are multiple slices s'_i satisfying s, for $i = 0, 1, \ldots, n$. For instance, each slice might contain a production of the form $B \to \beta_i$, where $\beta_0 \neq \beta_1 \neq \cdots \neq \beta_n$. Although one might then be tempted to write the set of formulas

$$s \to s'_0, \quad s \to s'_1, \quad \ldots, \quad s \to s'_n$$

this would be incorrect. In fact, in a grammar, rules of the form $B \to \beta_i$ represent *choices* between possible rewritings, rather than constraints that shall hold *at the same time*.[4] Consequently, in this case we can infer the constraint:

$$s \to (s'_0 \vee s'_1 \vee \cdots \vee s'_n) \tag{5}$$

For instance, in Fig. 6, the collection of equally-labeled outbound edges can be expressed using the formula:

$$\texttt{StateDefList} \to (\texttt{SimpleState} \vee \texttt{InnerCompositeStates} \vee \texttt{OuterCompositeStates}). \tag{6}$$

[4]Recall that each production expresses a rewriting of the symbol on the left with all the symbols on the right; the logic constraints above would mean that B must be rewritten to *every* β_i *at the same time*, which clearly does not make sense.

As a general rule, if there is one and only one slice s' that satisfies s, then the logic constraint in Eq. (4) encodes the mandatory requirement that, when s is in the language, then also s' shall be included; otherwise, if there are n slices s'_i that satisfy s, then the logic constraint Eq. (5) encodes the requirement that, when s is in the language, *at least one* slice s'_i shall be included. By reasoning in the same way for each slice in set S and for each pair in set D, we obtain a collection of *crosscutting constraints* that we can pair with the tree that we have as a result of the clustering procedure.

Conflicting components. Another kind of constraints can be inferred by looking at conflicts between components. So far we did not consider logical *exclusion* between features; we may say that two language features are in conflict if introducing both of them leads to an incorrect language implementation. In particular, two *language components* that define different semantics for the same keyword cannot coexist at the same time in the same language implementation; in this case, the conflict can be detected at the level of the language framework. In the case of Neverlang, we consider that two slices are in conflict when they define the same syntax with different semantics (in Sect. 5, we report on such a conflict).

4.5 Further Refactorings

The leaves of the tree that we obtain from the previous procedure represent the components that constitute our language. However, so far each feature has to be considered optional. In the following, we will show how to infer further relations between nodes of the tree. Some relations will be retrieved from the slices, using the dependency graph: therefore, most of them will directly relate to the leaves of the tree; but it is possible to promote these relations to the internal nodes of the tree. Other relations will be inferred from the semantic network.

Conflicting concepts. Another kind of logical exclusion (or conflict) can arise between domain *concepts*. In this case the conflict cannot be detected by the language framework, because conflicts between concepts are domain-specific. Conflicting concepts, however, are *antonyms* in the semantic network —as provided by a domain expert. In both cases we can encode the conflict as a logic formula. Now, let be s and s' two features in conflict; then we can write:

$$s \to \neg s' \tag{7}$$

which is effectively stating that when one feature is present in our language, the other shall not be included at the same time. An example of possible conflict between concepts is between `SingleTriggerFork` and `MultiTriggerFork`, which are *antonyms* to each other (see Fig. 4). In practice, some languages for state machine definition actually admit multiple event trigger after a fork and some may not (e.g., UML vs. Rhapsody [11]). Therefore it is left to the developer to decide whether or not `SingleTriggerFork` and `MultiTriggerFork` are antonyms and mutually exclusive. Since the names `SingleTriggerFork`, `MultiTriggerFork` are trivially mapped onto the slices with the same name, we can easily also add a crosscutting constraint similar to (9), encoding that it is not possible to generate a meaningful state machine language implementation that includes both features.

Other constraints. A *meronym* encodes the *part of* relation. For instance, we can say that *wheel* is a meronym of *car*. In particular, in this work, we are assuming that when

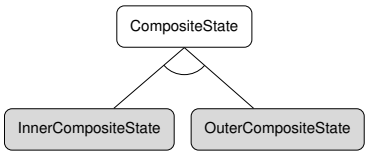

Fig. 7: `InnerCompositeStates` → ¬`OuterCompositeState`

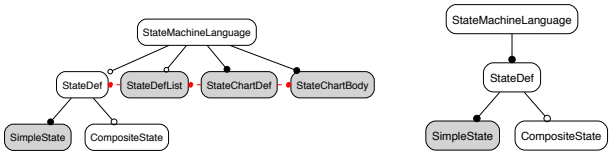

Fig. 8: Collapsing mandatory features.

$(w, w') \in$ mero, then w is *part of* w', and concept w may exist in our language if and only if w' is also present. Therefore, for all pairs $(w, w') \in$ mero we must add the constraint

$$s \leftrightarrow s' \tag{8}$$

for all pairs (s, s') such that $w \in T(s)$ and $w' \in T(s')$. For instance, in our example, one such case is

`StateChartDef` ↔ `StateMachineLanguage`.

In fact, the `StateChartDef` slice defines the outer container for a state machine program (see Listing 1). Therefore, it is *a part of* a `StateMachineLanguage` and there cannot be a `StateMachineLanguage` without the outer container of its body. In other words, meronyms can be used to infer *mandatory features* in the variability model (see for instance Fig. 8).

Synthesis of variability information. The result of the previous phase is a provisional feature diagram: a tree with a possibly large collection of logic formulas. The process is likely to be incremental: a domain expert can refine the set of cross-tree constraints. All of these formulas say something about the relations between language components, and many of these formulas will crosscut between different subtrees. However, many formulas can be simplified, promoted to internal nodes of the tree and encoded directly in the structure of the variability model using heuristics and simple rewriting strategies. For instance, consider Fig. 5, and the logic constraint in Eq. (9). The synthesis process reconstructs (i.e., *refactors*) the feature diagram and does make visible the variability information in the tree. As shown in Fig. 7 there is now an Xor-group in the subtree rooted at `CompositeState`. In fact, each time a constraint is added or revised, the new variability information can be automatically synthesized in the feature diagram. We apply state-of-the-art synthesis techniques [4, 1] that guarantee the feature diagram is *sound* and *maximal* at each step of the editing process.

5. CASE STUDIES

We evaluated our approach using two case studies. The first is an extended version of the running example (a family of state machine languages). The second case study is a family of simple, imperative, general purpose languages mixed with the Linda coordination model [15]. In both cases we automatically extracted the dependency graph from the slices, we transformed it to a feature model by using the semantic network, and inferred some further constraint to improve the final result. At the end, the obtained feature model permits to select which language belonging out of those pertaining to the described families we desire to get

`StateChartDef`	Outer container of the state machine body
`StateChartBody`	Body of the state machine
`StateDefList`	List of states
`SimpleState`	Syntax for simple state
`StartState`	Syntax for pseudostate start
`FinalState`	Syntax for pseudostate final
`InnerCompositeStates`	Specific definitions for Inner semantics
`OuterCompositeStates`	Specific definitions for Outer semantics
`MultiTriggerForkDef`	Syntax for Fork with Multi Trigger
`SingleTriggerForkDef`	Syntax for Fork with Single Trigger
`Transition`	Definition of a transition
`TransitionDefList`	List of transitions
`TransitionAction`	Body of a transition
`Trigger`	Trigger of a transition
`Guard`	Guard of a transition
`Effect`	Effect of a transition
`Join`	Join pseudostate implementation
`Fork`	Fork pseudostate implementation
`ForkTransition`	Fork Transition in the Single Trigger case
`ForkLeftTransition`	Left Transition in the Multi Trigger case
`ForkRightTransition`	Right Transition in the Multi Trigger case

Table 1: Slices for the SM language family

and its implementation is automatically achieved via Neverlang and the composition of the selected slices. The goal of this section is twofold. First, we report on our practical experience and further illustrate our approach (see Sect. 5.1 and 5.2). Second, we aim to assess which parts of a variability model can be automatically extracted and which parts require domain knowledge —see Sect. 5.3.

5.1 Family of State Machines

In the previous sections, the family of state machine (SM) languages has been used as our running example, focusing on a smaller subset of the cases to ease the explanations. The case study implements all of the slices found in Table 1. We will now complete the description of this case study. In Fig. 9) we show the full variability model; gray nodes are directly mapped onto the slices (Table 1), while white nodes either mapped are onto multiple slices or have no direct correspondence to a slice.

In Fig. 9, `SimpleState` is a *mandatory* child of `StateDef`, because in our semantic network (Fig. 3) `SimpleState` is a *meronym* for `StateDef`. From the dependency graph (a portion not shown here for lack of space) the slices `TransitionBody`, `Transition`, and `TransitionDefList` are all dependent on each other; in particular, we can infer the following constraints:

`TransitionDefList` → `Transition`

`Transition` → `TransitionBody`

`TransitionDef` and `TransitionDefList` are *meronyms* of `StateMachineLanguage` and `Transition` is a synonym of `TransitionDef`: therefore `Transition` and `TransitionDefList` are *mandatory*. As seen in Sect. 4.5, the nodes can be folded into their parent `TransitionDef` (cf. Fig. 8). A similar pattern can be detected in the subtree rooted in `MultiTriggerFork`: all of its children can be folded inside this node: in fact, `ForkLeftTransition` and `ForkRightTransition` are meronyms for `MultiTriggerFork` that it is a synonym of `MultiTriggerForkDef`.The following constraints from the dependency graph:

`SingleTriggerFork` → `ForkTransition`

`MultiTriggerFork` → `ForkTransition`

can be promoted to the parent (Sect. 4.5) as

`ForkTrigger` → `ForkTransition`.

and being `ForkTransition` and `ForkTrigger` meronyms for `Fork`:

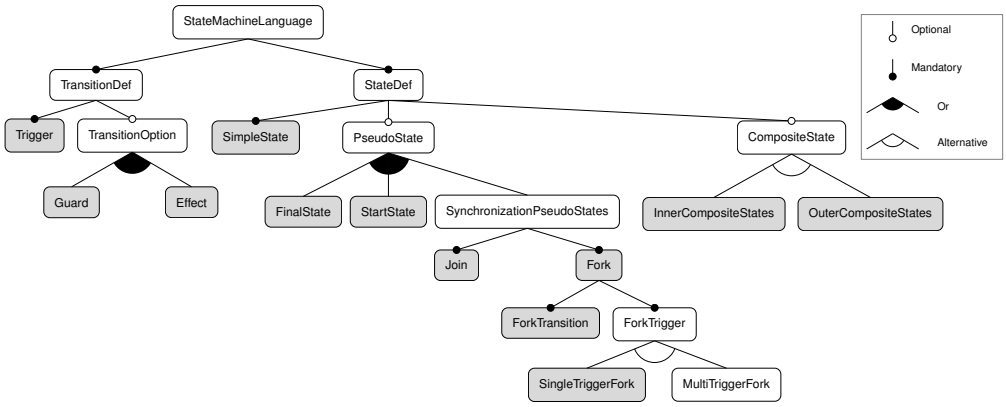

Fig. 9: Final generated variability model. In grey, features that are mapped 1:1 to a slice.

$$\texttt{ForkTransition} \leftrightarrow \texttt{Fork}, \quad \texttt{ForkTrigger} \leftrightarrow \texttt{Fork}.$$

All of these fork-related nodes are children of the same parent `Fork`; it follows that `ForkTransition` and `ForkTrigger` are *mandatory*, and that whenever `ForkTrigger` is chosen, one of its children should be chosen as well; so `SingleTriggerFork` and `MultiTriggerFork` are in an *alternative* relation (they are in conflict: see Sect. 4.5). Finally, another constraint can be found in the dependency graph:

$$\texttt{CompositeState} \rightarrow \texttt{StateChartBody}.$$

However, since `StateChartBody` has been already folded into the root `StateMachineLanguage`, this formula is redundant, and it can be removed. It follows that in this particular case we could remove all of the constraints and encode them directly in the structure of the tree.

Two components implement the behavior of the compiler in correspondence with the same keyword. It is the case of `InnerCompositeStates` and `OuterCompositeStates`. As it is not possible to include both components in the same language implementation, they are in conflict. Such situations can be automatically inferred and the constraint can be added:

$$\texttt{InnerCompositeState} \rightarrow \neg\texttt{OuterCompositeState} \qquad (9)$$

5.2 Imperative Languages with Linda

Gelernter *et al.* [15] introduced Linda, a coordination model that implemented inter-process communication using a shared memory concept called *tuple space*. This model implements a very limited set of concepts (only six primitives: **`in`**, **`inp`**, **`out`**, **`rd`**, **`rdp`** and **`eval`**) that can be embedded in a host programming language. We further validate our approach by applying the same process used for the state machine language family to the union of a collection of slices for a simple imperative language and a set of slices that implement the Linda primitives (see Table 2 for the implemented slices). Linda primitives have been implemented as pairs of slices with a threaded backend (local execution) or an RMI backend (distributed execution): in this case, since only their semantics change, a language that includes the slices for Linda may be executed either locally or in a distributed context, without requiring a rewrite.

For instance, Listing 2 shows a program written in a language of the family represented by the variability model in Fig. 10. The language includes the features `Expression`, `VariableDeclaration`, `PrintStatement`, `Block`, and then it is possible to choose a pair of slices between `LindaRMI` + `RMIProcess` and `LindaThread` + `ThreadProcess`. Of course, it is not possible

```
process ProcessFoo {
    var messageToBar = "hello from Process Foo!"; var intValue = 100;
    out(messageToBar, intValue);
    var messageFromBar = "";
    in(?messageFromBar);
    print "Foo> String:  "+messageFromBar;
}
process ProcessBar {
    var messageFromFoo = ""; var intFromFoo =  -1;
    in(?messageFromFoo, ?intFromFoo)
    print "Bar> String:  "+messageFromFoo+" and int: "+intFromFoo;
    var messageToFoo = "hello from Process Bar!";
    out(messageToFoo);
}
```

Listing 2: Processes in a language from the *Linda* family.

`RMIProcess`	Outer container for a process (RMI-based)
`ThreadProcess`	Outer container for a process (Thread-based)
`Block`	List of statements between curly braces {}
`IfStatement`	Branch construct `if`
`ForStatement`	Loop construct `for`
`WhileStatement`	Loop construct `while`
`DoWhileStatement`	Loop construct `do-while`
`VariableDeclaration`	Syntax for variable declaration
`Expression`	Expressions, numeric and string type defs
`LindaRMI`	RMI-based Linda primitives
`LindaThread`	Thread-based Linda primitives

Table 2: Slices for the Linda language family

to mix and match those, because of the constraints shown in the variability model.

This example might be seen as not *domain-specific*, as it defines parts of a general-purpose imperative language, but it couples them with primitives from the Linda language, and it can therefore be considered to some extent a family of DSLs. In this context, reasoning on the componentization of a general-purpose imperative language is interesting since many components result optional (see Fig. 10). This is really not surprising, given the generality of the involved constructs, and it makes perfect sense: for instance, a simple imperative language may be restricted to a non-turing complete subset (e.g., no looping constructs, as in the provided example), and still serve some purpose. As you can see in the resulting variability model, most of the components are in fact optional or in an *or* relation. For lack of space, we have chosen not to show the semantic network used to generate the variability model but we will discuss how we obtained the result. For this example it was necessary to take a simple semantic network that put in relation general concepts for programming languages (e.g., loops, branches, control flow,

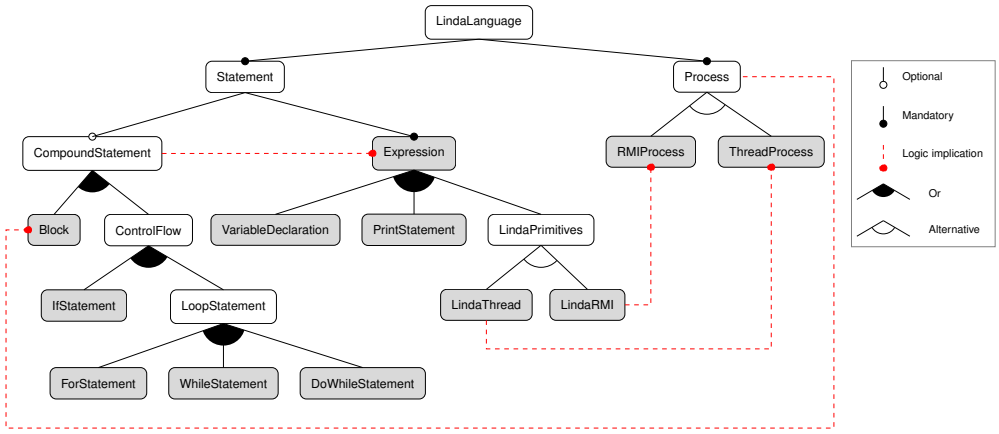

Fig. 10: Linda variability model. Logic implications are in red. In gray, features that map to one slice.

etc.) and enriching it with concepts that are related to the coordination domain whose Linda belongs.

In Fig. 11 we show a detail of the provisional variability model that we obtained after applying the clustering algorithm, and merging the nodes. At this point, all of the child of `SimpleStatement` are optional, and therefore, in an *or* relation. `Expression` is a child of `SimpleStatement`. The edges of the dependency graph have been superimposed in the figure in red or are spelled out as logic formulas: as you can see, there are many interdependencies between `Expression` and other components. In the final variability model (Fig. 10) we could fold `Expression` into its parent `SimpleStatement`. In general, this is not necessarily possible: for instance, `ControlFlow`'s sibling `Block` depends on `SimpleStatement` and *not* necessarily on `Expression`, and a language might include a statement that performs some kind of action that does not require an expression (e.g., printing a newline). However, because in this case *every* sibling of `Expression` depends on it, and because there are no other siblings that *do not* depend on it, it makes sense to fold `Expression` into its parent. In fact, since `ControlFlow` depends on `Expression`, then the dependency would be promoted to `SimpleStatement`. However, `ControlFlow`'s sibling `Block` depends on `SimpleStatement` and *not* necessarily on `Expression`, because, in general, there might be a statement that performs some kind of action that does not require an expression. Still, because in this case *every* sibling of `Expression` depends on it, and because there are no other siblings that *do not* depend on it, even including `Block` would transitively cause `Expression` to be included (as a dependency of another child of `SimpleStatement`); therefore, it makes sense to fold `Expression` into the parent.

Although one might be tempted to try and perform other refactorings, the model shown in Fig. 10 cannot be simplified further. For instance, because of the 1:1 mapping between the `LindaRMI` slice, which implements the primitives in the RMI case, and `RMIProcess` (and the same for the thread-based implementation of the primitives and of the `process` definition), one might be tempted to fold them into one feature. However, this would not be correct, since the implication is only one-sided: it is possible, and it makes sense to define a language where processes are executed in a thread or distributed using RMI and *do not* include the Linda primitives in this language (for instance, one might use other libraries to perform message passing). This means that even language that *do not* include Linda constructs belong to this family.

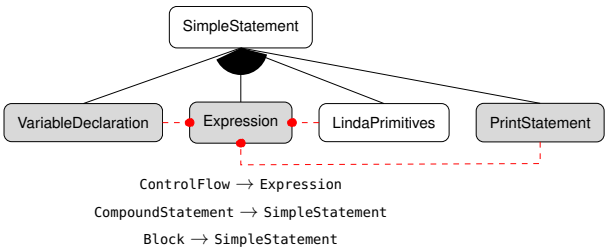

Fig. 11: A detail of the VM for `SimpleStatement`.

5.3 Automation and Domain Knowledge

Table 3 summarizes the results for the two case studies. For each case study (SM and Linda), we first report on the number of antonyms, synonyms, meronyms, and hypernyms. The information is included in the semantic work and specified by a domain expert. We also report on the number of nodes and edges in the dependency graph —the inference of the dependency graph is fully automated. *1:1 ft/slice* is the number of features that correspond to an actual component language (slice). Such features have been depicted in gray throughout the paper. The other features represent *abstract* concepts and they have been automatically inferred from the semantic network. Finally we reported on the total number of exclude relationships (being represented as cross-tree constraints or alternative groups), mandatory relationships, and or-groups in the final feature model of each case study.

The results show that the dependency graph of the existing language components need to be refined with the expertize from the domain expert (i.e., by using the semantic network) to get a feasible feature model. First additional features should be added to structure the feature models. Abstract features, not originally present in the dependency graph, represent 43% (resp. 37%) of the final feature model in SM (resp. Linda). Second, heuristics for mining configuration constraints (or-groups and excludes) supplement the feature models. Third, additional logical relationships (e.g., exclusion, mandatory relations) have been inferred from the semantic network.

6. RELATED WORK

Many authors have addressed the problem of recovering a feature model from various kinds of artifacts. She *et al.* [30] showed how to reverse engineer a feature model starting from *feature descriptions* (written in natural language) and

	SM	Linda
#antonyms	1	2
#synonyms	0	2
#meronyms	13	10
#hypernyms	13	14
#DG's nodes	18	13
#DG's edges	21	32
#1:1 ft/slice	12 (out of 21)	12 (out of 19)
#excludes	2	2
#mandatory	8	3
#Or-groups	2	4

Table 3: Results of the case studies

static analysis of source code. Davril *et al.* [12] presented a fully automated approach for constructing feature models from publicly available product descriptions (e.g., as found in SoftPedia and CNET). Alves *et al.* [3] and Niu *et al.* [26] use clustering techniques to infer a tree structure. Ferrari *et al.* [13] considered natural language documents. Weston *et al.* [35] extract feature models from the requirements description in natural language.

A key difference with our work is the presence of textual artifacts to mine and organize features. In our context, we can only rely on slice names —there is no feature description— and on the natural dependencies between language components —the dependency graph. It explains why we need domain knowledge in the form of a semantic network, providing extra information to refine the provisional feature model that can be inferred from the language components.

FAMILIAR [1] provides an environment to synthesize feature models from a propositional formulas. An interactive support (through ranking lists, clusters, and logical heuristics) for choosing a sound and meaningful hierarchy is part of the environment [5, 6]. Generic ontologies (like WordNet or Wikipedia) are exploited as well as synthesis techniques [4]. In our context, there are three notable differences: (1) the dependency graph is a rough over-approximation of the configuration set (2) the complete list of features is not *a priori* known (3) feature names are quite technical and specific. Therefore the application of synthesis techniques [4, 1, 5] is not immediate and requires some user effort.

Another related subject is constraint mining. In [2], architectural and expert knowledge as well as plugins dependencies are combined to obtain an exploitable and maintainable feature model. Ryssel *et al.* [29] developed methods based on formal concept analysis and analyzed incidence matrices containing matching relations. Nadi *et al.* [25] developed a comprehensive infrastructure to automatically extract configuration constraints from C code. Their empirical study showed that many of the constraints require expert knowledge or more specific analysis. Our experience in a very different context concurs with the findings. A substantial amount of constraints cannot be inferred only from the analysis of languages components, despite the development of specific heuristics for mining constraints (see Sect. 4.4).

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 [27, 18, 10]. The support for identifying constraints and organizing features is likely to be relevant as well for other variability formalisms,

Some work has applied variability management to language implementation. Although we used Neverlang [8, 7], other modular language implementation frameworks can be employed to implement a similar kind of approach (e.g., [34, 17]). Cengarle *et al.* [9] use MontiCore [21] to describe variations of a base language. Haugen *et al.* [16] have used CVL to model possible DSL variations. White *et al.* [36] use feature modeling to improve reusability of features among a language family. In Liebig *et al.* [22] a family of languages is decomposed in terms of their features. The authors do not start from a set of pre-defined components, but rather they componentize an already existing language and develop the variability model to support it. Therefore, relations between language components are imposed by the developers as they implement them. In our approach we discover the relations between the components using information that we extract directly from the implemented language components. Our objectives are thus quite different: we want to help a user finding implicit or explicit relations between *existing* components. The result makes it possible for end-users to configure their own DSL.

7. CONCLUSIONS

We presented an approach to automatically infer a variability model from a set of language components, given a description of a domain as a semantic network. The resulting variability model represents a family of languages that can be implemented using those components, with respect to the given domain description. We evaluated our approach against two case studies: the state machine and the imperative+linda family of languages. Such an evaluation showed that (1) the automatic extraction of constraints out of language components is feasible but it is only a starting point of the inference of a variability model; (2) the initial variability model can be automatically refined by using some domain knowledge (expressed as a semantic network).

We developed tools and automated techniques to support the process. Semantic network (as a description of a domain) as well as clustering techniques are used to hierarchically organize features of the variability model. The variability model has the merit of being readable, well-structured, and consistent with the *technical* and the *domain* constraints. The resulting variability model can be exploited to configure a family of languages and automatically generate a language implementation.

We implemented a prototype that uses the Neverlang [8, 7, 33] framework for the implementation of the language components and the common variability language (CVL) [14], as the domain-independent language for specifying and resolving variability. Given a desired combination of features, we are able to generate a language implementation as the composition of the set of Neverlang slices. The automating support developed in this paper now allows developers and users of DSLs to shift to a variability approach. As future work, we plan to investigate how inferred variability models can be maintained in parallel with the evolution of the DSL.

Acknowledgements. This work has been partially supported by the MIUR project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society, by the ANR INS Project GEMOC (ANR-12-INSE-0011), and the bilateral collaboration VaryMDE between INRIA and Thales Research & Technology.

8. REFERENCES

[1] M. Acher, B. Baudry, P. Heymans, A. Cleve, and J.-L.

Hainaut. Support for Reverse Engineering and Maintaining Feature Models. In P. Collet and K. Schmid, editors, *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'13)*, Pisa, Italy, Jan. 2013. ACM.

[2] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and Evolution of Architectural Variability Models in Plugin-based Systems. *Software and Systems Modeling*, July 2013.

[3] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In K. Pohl and B. Geppert, editors, *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 67–76, Limerick, Ireland, Sept. 2008. IEEE.

[4] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient Synthesis of Feature Models. In C. Scwanninger and D. Benavides, editors, *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, pages 97–106, Salvador, Brazil, Sept. 2012.

[5] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr. Breathing Ontological Knowledge Into Feature Model Management. Technical report, INRIA, Rennes, France, Oct. 2013.

[6] G. Bécan, S. Ben Nasr, M. Acher, and B. Baudry. WebFML: Synthesizing Feature Models Everywhere. In P. Heymans and J. Rubin, editors, *Proceedings of 18th International Software Product Line Conference (SPLC'14)*, Florence, Italy, Sept. 2014.

[7] W. Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In T. Gschwind, F. De Paoli, V. Gruhn, and M. Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[8] W. Cazzola and E. Vacchi. Neverlang 2: Componentised Language Development for the JVM. In W. Binder, E. Bodden, and W. Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[9] M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within Modeling Language Definitions. In A. Schürr and B. Selic, editors, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, LNCS 5795, pages 670–684, Denver, CO, USA, Oct. 2009. Springer.

[10] L. Chen and M. A. 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.

[11] M. L. Crane and J. Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In L. Briand and C. Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, LNCS 3713, pages 97–112. Springer, 2005.

[12] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature Model Extraction from Large Collections of Informal Product Descriptions. In L. Baresi and M. Mezini, editors, *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 290–300, Saint Petersburg, Russia, Aug. 2013. ACM.

[13] A. Ferrari, G. O. Spagnolo, and F. Dell'Orletta. Mining Commonalities and Variabilities from Natural Language Documents. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, pages 116–120, Tokyo, Japan, Sept. 2013. ACM.

[14] F. Fleurey, Ø. Haugen, B. Møller-Pedersen, A. Svendsen, and X. Zhang. Standardizing Variability — Challenges and Solutions. In I. Ober and I. Ober, editors, *Proceedings of the 15th International Conference on Interating System and Software Modeling (SDL'11)*, LNCS 7083, pages 233–246, Tolouse, France, 2011. Springer.

[15] D. Gelernter. Generative Communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, Jan. 1985.

[16] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In K. Pohl and B. Geppert, editors, *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 139–148, Limerick, Ireland, Sept. 2008. IEEE.

[17] 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 Proceedings — Software*, 152(2):54–69, Apr. 2005.

[18] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A Preliminary Review on the Application of Feature Diagrams in Practice. In D. Benavides, D. S. Batory, and P. Grünbacher, editors, *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pages 53–59, Linz, Austria, Jan. 2010.

[19] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In H. Gall and N. Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 471–480, Waikiki, Honolulu, Hawaii, May 2011. IEEE.

[20] G. Kövesdán, M. Asztalos, and L. Lengyel. A Classification of Domain-Specific Language Intents. *International Journal of Modeling and Optimization*, 4(1):67–73, Feb. 2014.

[21] 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.

[22] J. Liebig, R. Daniel, and S. Apel. Feature-Oriented Language Families: A Case Study. In P. Collet and K. Schmid, editors, *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Pisa, Italy, Jan. 2013. ACM.

[23] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[24] M. Mernik and V. Žumer. Incremental Programming Language Development. *Computer Languages, Systems and Structures*, 31(1):1–16, Apr. 2005.

[25] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analysis and Empirical Results. In L. Briand and A. van der Hoek, edi-

tors, *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 140–151, Hyderabad, India, May 2014. IEEE.

[26] N. Niu and S. Easterbrook. On-Demand Cluster Analysis for Product Line Functional Requirements. In K. Pohl and B. Geppert, editors, *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 87–96, Limerick, Ireland, Sept. 2008. IEEE.

[27] K. Pohl and A. Metzger. Variability Management in Software Product Line Engineering. In L. J. Osterwell, H. D. Rombach, and M. L. Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 1049–1050, Shanghai, China, May 2006. ACM.

[28] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*, pages 127–136, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[29] U. Ryssel, J. Ploennings, and K. Kabitzsch. Extraction of Feature Models from Formal Contexts. In F. Heidenreich and M. Resenmüller, editors, *Proceedings of the 3rd Wirkshop on Feature-Oriented Software Development (FOSD'11)*, pages 1–8, Münich, Germany, Aug. 2011.

[30] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse Engineering Feature Models. In H. Gall and N. Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 461–470, Waikiki, Honolulu, Hawaii, May 2011. IEEE.

[31] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, Reading, MA, USA, Mar. 2006.

[32] E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability Support in Domain-Specific Language Devevlopment. In M. Erwig, R. F. Paige, and E. van Wyk, editors, *Proceedings of 6th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes on Computer Science 8225, pages 76–95, Indianapolis, USA, 27th-28th of Oct. 2013. Springer.

[33] E. Vacchi, D. M. Olivares, A. Shaqiri, and W. Cazzola. Neverlang 2: A Framework for Modular Language Implementation. In *Proceedings of the 13th International Conference on Modularity (Modularity'14)*, pages 23–26, Lugano, Switzerland, 22nd-25th of Apr. 2014. ACM.

[34] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, LNCS 2304, pages 128–142, Grenoble, France, Apr. 2002. Springer.

[35] N. Weston, R. Chitchyan, and A. Rashid. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In J. mc Gregor and D. Muthig, editors, *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, pages 211–220, San Francisco, CA, USA, Aug. 2009. ACM.

[36] J. White, J. H. Hill, J. Gray, S. Tambe, A. Gokhale, and D. C. Schmidt. Improving Domain-specific Language Reuse with Software Product-line Configuration Techniques. *IEEE Software*, 26(4):47–53, July-Aug. 2009.