Thomas Kühn, Walter Cazzola, Diego Mathias Olivares

Choosy and picky: configuration of language product lines

Erstveröffentlichung in / First published in:

International Conference on Software Product Lines. Nashville, 20.–24.07.2015. ACM Digital Library, S. 71–80. ISBN 978-1-4503-3613-0.

DOI: https://doi.org/10.1145/2791060.2791092

Choosy and Picky: Configuration of Language Product Lines

Thomas Kühn
Software Technology Group
Technische Universität Dresden
thomas.kuehn3@tu-dresden.de

Walter Cazzola
Computer Science Department
Università degli Studi di Milano
cazzola@di.unimi.it

Diego Mathias Olivares
Computer Science Department
Università degli Studi di Milano
olivares@di.unimi.it

ABSTRACT

Although most programming languages naturally share several language features, they are typically implemented as a monolithic product. Language features cannot be plugged and unplugged from a language and reused in another language. Some modular approaches to language construction do exist but composing language features requires a deep understanding of its implementation hampering their use. The choose and pick approach from software product lines provides an easy way to compose a language out of a set of language features. However, current approaches to language product lines are not sufficient enough to cope with the complexity and evolution of real world programming languages. In this work, we propose a general light-weight bottom-up approach to automatically extract a feature model from a set of tagged language components. We applied this approach to the Neverlang language development framework and developed the AiDE tool to guide language developers towards a valid language composition. The approach has been evaluated on a decomposed version of Javascript to highlight the benefits of such a language product line.

CCS Concepts

•**Software and its engineering** → *Frameworks;* **Translator writing systems and compiler generators;** *Software product lines;*

Keywords

Language Product Lines, Language Composition.

1. INTRODUCTION

Over the past few decades there has been a staggering amount of new programming and domain-specific languages featuring new constructs for particular application domains, e.g.: concurrency, querying, and hardware design. Although most of these languages naturally share several language features[1], their compilers/interpreters are usually build from scratch as a monolithic product. Hence, language developers can neither reuse language features nor whole subsets of another language when implementing their own language. Moreover, it is difficult to directly vary the number of features supported by a monolithic language, as their implementation is buried within the compiler/interpreter. To overcome this, several frameworks for modular language construction have been proposed, e.g., LISA [29], Silver/Copper [40], Spoofax [24], and Neverlang [8, 10, 36]. These allow for reusing portions of previously defined language features when creating new languages. On the downside, the language developer needs to have a deep understanding of the features' implementation, dependencies, and side-effects to employ them in a sound way. So far, features of mainstream languages can hardly be reused, removed, and extended.

This calls for a new way of thinking about programming languages, whose compiler/interpreter is not a monolithic product but the result of the composition of many language features. Adding, modifying or discarding features from a language in turn will simply act on the features part of the process and the compiler will be just a by-product of the composition process. Consequently, programming languages can be treated as a *software product line* (SPL), where features correspond to language constructs mapped to their syntactical and semantical implementation. Using techniques from SPL, language developers can now choose and pick the desired language features to create the language best suited for his/her needs. As a result, the programming language becomes a family of programming languages created by a *language product line* (LPL).

Although tools for feature modeling and product configuration, such as FeatureHouse [6] and FeatureIDE [34], are well-established for typical SPLs, they are not sufficient enough to cope with families of real world programming languages. This is due to the fact that real world languages are very complex and tend to grow over time. Thus, typical top-down feature modeling that start from a feature model and afterwards creates a feature mapping to concrete artifacts fail miserably, because new language features might be introduced in every version of the language and might invalidate big portions of the feature mapping. To make things worse, general-purpose programming languages like C, Java, or Javascript tend to provide a massive amount of built-in language features. This, in turn, makes the generation and

DOI: http://dx.doi.org/10.1145/2791060.2791092

[1]A *language feature* is, in our view, a language construct as the for loop or a language concept without any syntactic correspondent as scope and coercion.

```
Expr    ::= Add
Add     ::= Mul     | Add '+' Mul   | Add '-' Mul
Mul     ::= Unary   | Mul '*' Unary | Mul '/' Unary
Unary   ::= Primary |'+' Unary      | '-' Unary
Primary ::= Literal | '(' Expr ')'
Literal ::= [0-9]+
```

Listing 1: Grammar of the expression language

maintenance of such a family of programming languages infeasible. While the tooling can still handle the number of features, the biggest problem arises from the strong interdependence of language features. Because choosing one language feature without picking the dependent language features, would result in an open syntax definition or erroneous semantics. However, enforcing the feature selection would drastically reduce the number of language variants also including those conceived to be useful for educational purposes [9]. Consequently, a better tool for language configuration must guide its users providing means to resolve open syntax definitions and missing requirements without requiring in-depth knowledge about the features' implementations. In sum, current SPL approaches cannot fully support the creation and provision of language product lines.

To overcome their shortcomings, this work proposes a general light-weight approach to automatically extract a feature model from a set of tagged language constructs. Additionally, we present AiDE, a tool to guide language developers towards a valid language configuration by employing the Neverlang language development framework. We show that the presented approach and tooling is suitable to create, maintain, and extend a real world programming language as a LPL. For this purpose, we implemented a modular and reusable version of Javascript, namely Neverlang.JS; configured several language subsets for teaching programming to students; and created 14 additional language features that can be seamlessly integrated into Neverlang.JS.

This paper is structured accordingly. Sect. 2 briefly introduces language product lines by covering language decomposition, dependency extraction, and language configuration. Sect. 3 describes our feature model generation approach. Afterwards, Sect. 4 shows its application using the Neverlang approach to language composition and introduces the language configuration tool: AiDE. The presented approach is evaluated, in Sect. 5, by describing the implementation, configuration, and extension of Neverlang.JS. The paper is concluded by reviewing related work (Sect. 6) and summarizing the presented results and future prospects (Sect. 7).

2. LANGUAGE PRODUCT LINES

The development of families of programming and domain-specific languages has gained popularity among researchers and practitioners, e.g., [26, 20, 30]. Following the ideas of SPL, a LPL facilitates the process of language development, which can be *customized* by selecting individual *features*. Similar, to SPLs a language could be designed to specifically suit a certain use case or application domain. For instance, authors [35, 15, 38] have shown that the many variants of state machine languages could be modeled as one single family of programming languages. Nonetheless, this is also true for *general-purpose programming languages*, from which *dialects* may be defined for DSL purposes. On one side, specialized versions of full-fledged programming languages can be employed in case of security purposes (*e.g.,* Java Card [14]) or teaching programming [17]. Language extension, on the other side, can be useful to embed new language features into an existing programming language, such as type-checked SQL queries [18]. Following this vision of a product line for programming languages, this section describes the artifacts and models required to establish such a product line.

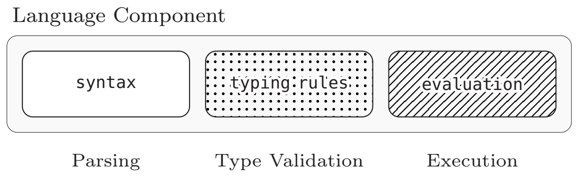


Figure 1: Language component and its parts

2.1 Running Example

Before we delve into the peculiarities of LPLs, it is useful to consider a small example: the *expression language*. Listing 1, shows the typical text book grammar of expressions in EBNF notation [2]. For brevity, the example is limited to natural numbers and the basic mathematical operators for addition, subtraction, multiplication, division, and brackets.[2] In particular, the grammar directly reflect the precedence of the operators, as it is usually necessary for parser generators of the LR-family [11]. Despite the fact that typically the full expression language is reused or extended and for the sake of argument, the language is further decomposed to elaborate on notions, such as language feature, feature dependencies, and language configuration.

2.2 Language Decomposition

To *model* the variability of a language family, their language features must be organized in a feature model and mapped onto concrete artifacts. These artifacts must be reusable and composable implementations of a partial language definition encompassing all necessary definitions for this language feature. A language feature is implemented by a *language component*. SPL approaches suggest to either collect and organize the features beforehand or retrieve the features by analyzing multiple variants of the same language. Unfortunately, neither approach is suitable for general purpose programming languages, because the number of language features tend to grow over time, the number of language variants is generally limited to previous versions, and the complexity of the monolithic compilers renders their analysis almost impossible.

This raises two questions: *How to effectively decompose a languages compiler/interpreter into reusable language components?* and *How to extract the dependencies between multiple language components?* To approach the former question, developers of language components have to choose the desired level of granularity. The expression language, for instance, can be developed as a single component limiting the flexibility and reusability, whereas a further decomposition increases the complexity of language configuration. Despite of the latter, our experience with language decomposition [36] indicates that each language feature, like a *variable declaration* or an *addition*, should be encapsulated in a

[2]Luckily, the intended semantics of this language is clear and thus needs no further explanation.

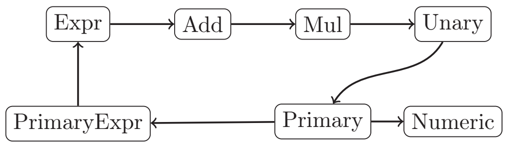


Figure 2: Language feature dependency graph

language component together with all corresponding syntactical and semantical descriptions. Figure 1 shows three typical aspects of a language feature encapsulated in a language component. First, it declares an open (partial) grammar to parse that construct. Second, it provides rules to validate the correct use of this construct. Finally, it defines the semantics of the construct by translation to code or direct evaluation. Although these are the general aspects of a language component, there are many more possible aspects to consider. The language feature *scoping*, for instance, does not correspond to a syntactic construct, but can be implemented in a crosscutting language component [8, 39, 36].

Consequently, the expression language is decomposed into seven language components: `Expr`, `Add`, `Mul`, `Unary`, `Primary`, `PrimaryExpr`, `Numeric`. Each grammar rule (Listing 1) corresponds to a language feature and thus to a language component. The only exception is the `Primary` rule that is further divided into two components to deal with the distinction between a *literal* and a *bracketed expression*.[3] Basically, each language component contains the respective grammar rule as its syntax with a specification of its evaluation.

In conclusion, our approach to language decomposition is to create a language component for each syntactic construct encapsulating both its syntax and semantics. While this answers the first question, a way to extract the dependencies between multiple language components is still missing.

2.3 Dependency Extraction

In [38, 37] the concept of a *dependency graph* was introduced for language components. In general, it describes the dependencies between multiple language components, as a tuple $DG = \langle S, D \rangle$, where S is the set of language components, and $D = \{(s, s') \mid s' \text{ satisfies } s\}$ the dependency relation. A language component s' *satisfies* s iff s' *provides* the implementation of a language feature that s *requires*.

While these dependencies can be inferred in a number of ways, we found that the language components' syntactical definitions provide enough information for constructing the corresponding dependency graph [38, 37, 36]. In detail, for each production rule of a component's syntax definition all nonterminals on the left-hand side of that rule are considered *provided*, whereas all nonterminals on the right-hand side are *required*. Consider for instance the `Unary` language component that, according to its production rule, requires the `Primary` and `Unary` nonterminal and provides the `Unary` nonterminal. Figure 2 shows the resulting dependency graph for the decomposed expression language. Although this graph shows only one kind of dependency, it implicitly implies two other kinds of dependencies:

- *unique* (mandatory) dependencies specify that a component is needed to satisfy a grammar rule.
- *alternate* (inclusive-or) dependencies occur when a nonterminal symbol of the grammar can be produced in more than a single way, therefore allowing a degree of choice upon the corresponding components.

In both cases these kinds of dependencies can be spotted by the in-degree of the graph nodes.

The dependency graph is a suitable starting point for creating a feature model, as showed in [38, 37]. However, some additional information must be provided to construct [38] or mine [37] a feature model from a set of language components. This is due to the complexity of the syntax of general-purpose languages that implicitly has many connected components, i.e., several language components that circularly depend on others. Just consider, the dependency graph of the decomposed expression language in Fig. 2. It basically resembles a circular chain of mandatory features, and thus, it is not suitable to generate a feature model. Nevertheless, Sect. 3 shows a simple approach to efficiently generate a feature model from a set of language components.

2.4 Language Configuration

Henceforth, let's assume that the feature model for the expression language family is given and faithfully reflects all the dependencies between language components.[4] Using standard tools for feature modeling, e.g., FeatureIDE [5], choosing any feature from the available features would result in the selection of all other features, as all of them are circularly dependent on each other. As a result, there would be only one variant of our expression language to choose from. Clearly, this is not an error within these tools, as they strictly follow the definition of the feature model by automatically resolving mandatory feature requirements.

One option is to let the user pick the best resolution strategy for required features and to ensure that the composite syntax of a language product is complete, i.e., without any *open* nonterminal that is not provided by any language component. The user can either select the mandatory feature or *rewire* the missing feature to another activated feature. This is done by adding a new production rule for the *open* nonterminal mapping it to another provided nonterminal. Additionally, rewiring can also be used to connect previously unrelated concerns by satisfying dangling features. Considering the family of expression languages, this allows for the creation of several other language variants, for instance, an expression language that only supports addition and subtraction of positive numbers. This variant is obtained by simply rewiring the missing `Mul` nonterminal to the selected `Primary` nonterminal. Concerning syntax, these options are sufficient to construct a reliable and usable grammar from the selected language components.

However, this component versatility can also lead to compatibility problems between language components, as rewiring nonterminals implies that the involved language components work upon equally defined, or at least compatible, attributes. Standard application frameworks tackle this problem by applying *design patterns* to maintain interface consistency through the whole framework. Similarly, it is useful to define a common set of attributes for all language components and encapsulate their internal state employing object-oriented inheritance and polymorphism. Eventually, a complete hierarchy of type families facilitates feature compatibility over all language components.

[3]Please note, that the actual expression language of `Neverlang.JS` is more complex, to accommodate for corner cases.

[4]Impatient readers can find the feature model in Fig. 3.

Table 1: Tags for the expression language

Nonterminal	Tags
Expr	expression
Add	expression, numbers, sum, sub
Mul	expression, numbers, mul, div
Unary	expression, unary, numbers, sign
Primary	expression, primary, literal
PrimaryExpr	expression, primary, parenthesis
Numeric	expression, primary, literal, numbers

In summary, letting the user being picky about resolution strategy allows for a more flexibly configuration of his/her desired language. Under the assumption that the user has a basic understanding of production rules and the components compatibility, since ambiguous resolutions may result in ill-conceived or broken language products.

2.5 Requirements

In our opinion, a viable and flexible LPL for a family of programming or domain-specific languages can only be established if the following five requirements are fulfilled:

R1 The language family must be decomposed wrt. to its language features.

R2 Each feature is encapsulated (including syntax and semantics) within a language component.

R3 A dependency graph capturing the constraints between language components can be constructed.

R4 A language development framework is able to compose arbitrary language components.

R5 A configuration tool for language products is used that supports multiple dependency resolution strategies.

These five requirements form a minimal set of prerequesits that must be fulfilled in order to facilitate a viable LPL.

3. FEATURE MODEL GENERATION

Our previous approach to automated feature model extraction [37] heavily relies on the presence of a *semantic network* that must be provided and maintained by a domain expert. Although this semantic network features *meronyms*, *hypernyms*, *synonyms*, and *antonyms*; only the first two semantic relations proved useful for extracting an initial feature model. Furthermore, the initial model must be manually adjusted. Consequently, both the maintenance of the semantic network and the manual adjustment of the feature model became a bottleneck for the development of more complex LPLs. Learning from previous mistakes, our new method for feature model extraction is lightweight and fully automated.

3.1 Tagging Language Components

Language components, as explained, deal with several facets of the language feature they implement. They provide the syntactic and semantic definition for the language feature and also have to disclose their *interface* (required and provided nonterminals) towards the other components and a classification for the role the language feature has within the whole language. Such a classification is based on *tag*s associated to the feature provided by the component (basically the provided nonterminals); a tag is a label that describes the nature of the language feature. Each feature is correctly described if *all* of its provided nonterminals are tagged.

Algorithm 1: GenerateFeatureTree(p:Node)

```
begin
    $T^* := \{t \mid c \in \text{childs}(p) \wedge t \in \text{tags}(c)\}$;
    while $\exists t \in T^* : |\{c \in childs(p) \mid t \in tags(c)\}| > 1$ do
        select $t' \in T^*$, such that $|\{c \in childs(p) \mid t' \in tags(c)\}|$
        is maximal;
        create new node $n$ with $tags(n) := \{t'\}$;
        $childs(p) := childs(p) \cup \{n\}$;
        for $o \in \{c \in childs(p) \mid t' \in tags(c)\}$ do
            $tags(o) := tags(o) \setminus \{t'\}$;
            if $tags(o) = \emptyset$ then
                move components of $o$ to $n$;
            else
                $childs(n) := childs(n) \cup \{o\}$;
            end
            $childs(p) := childs(p) \setminus \{o\}$;
        end
    end
    for $c \in childs(p)$ do
        VMTreeBuild($c$);
    end
end
```

Reprising the running example, Tab. 1 shows the tags associated to the various nonterminals provided by the language components of the expression language. The Unary nonterminal, for instance, is tagged with the labels: *expression*, *unary*, *numbers*, and *sign*; because each denotes a specific nature of the *unary expression*. Note that no assumptions are made on the used tags; the language developers (as domain experts) should rely on a common set of tags or naming conventions. Fortunately, language developers already use such a common terminology.

3.2 Deriving the Feature Tree

These tags are sufficient to automatically generate an initial feature tree from the language components by employing Algorithm 1. The initial feature tree is obtained by calling the algorithm on a dummy tree where all language components are child nodes of a single tag-less root node. The most recurring tag t' is extracted from 1st level children of each node and a new child node n labeled by $\{t'\}$ is created. Each time all the siblings containing t' are moved below n and t' is removed from their tags. When the current node is stable, i.e., no tag appears more than once among its children, the algorithm is applied recursively to each of its children. Basically, this algorithm creates a hierarchy by selecting the most common tags and introducing new branches for them. Fig. 3 shows the feature tree obtained by applying the algorithm to the tagged language components for the expression language (Tab. 1). The tags used by the algorithm are still listed underneath each feature.

Although the algorithm is fairly simple, it has several notable properties. First and foremost, the algorithm is guaranteed to produce a tree, because it is initially called with a tree and at each step of its execution child nodes will be either moved to a new child, removed, or retained. Hence, no node can become the child of two different parents at the same time. Furthermore, this also ensures that it never creates overlapping features. Second, as it generates an elementary feature model, all nodes are optional with respect to their parent (drawn as edges with white circle). This is due to the fact, that all child–parent relationships in the tree can be considered as *meronyms*. Last but not least, it can handle crosscutting constraints by distributing them among

Final edited form was published in "SPLC '15: 2015 International Conference on Software Product Lines. Nashville 2015", S. 71–80. ISBN: 978-1-4503-3613-0
https://doi.org/10.1145/2791060.2791092

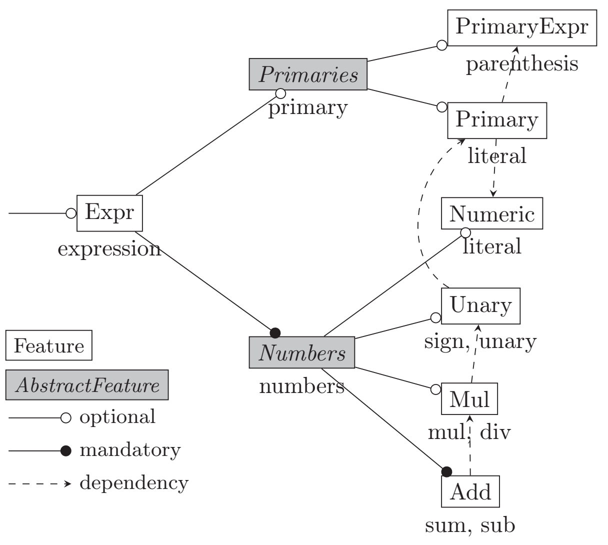


Figure 3: Feature model for the expression language

different branches of the initial feature tree. For instance, the language feature for *variables* has subfeatures related to declarations (*variable declaration*), statements (*variable assignment*), and expressions (*variable resolution*) that will be placed into the most appropriate branch. On the downside, the initial feature tree does not fully reflect the syntactical dependencies of the set of language components.

3.3 Adding Dependencies

Hence, the initial feature tree must be enriched with information retrieved from the dependency graph. Consequently, each dependency is added as a new directed edge to the feature tree. This affects the model in three alternative ways:

- If an edge does not exist yet, it is added and its dependency type is specified as either required or alternative.
- Otherwise, if a hierarchical relationship already exists, it is updated abiding the feature model specification. In detail, a unique dependency converts the corresponding feature to a *mandatory* feature.
- In case of multiple alternate dependencies the affected relations are replaced by an *inclusive-or*.

Finally, after including the dependencies to the initial feature tree, the result is a valid feature model respecting all the dependencies of the language family. Fig. 3 shows the feature model generated for our expression language family.

3.4 Summary

The presented method can automatically generate a feature model from an arbitrary set of tagged language components. The quality of the generated feature model solely depends on the quality of the used tagging scheme. Besides that, the presented method is generally applicable to LPLs regardless of the employed language development framework, as long as they fulfill the requirements R1 to R4 (Sect. 2.5) and provide a classification for the language components.

4. NEVERLANG BASED LPLS

This section exploits the Neverlang language development framework [10, 8, 36] to create reusable language components, to extract their features and dependencies, and dynamically configure language variants.

```
module neverlang.js.AddExpr {
  reference syntax {
    provides { AddExpr: expression, numbers, sum, sub; }
    requires { MulExpr; }
    [ADD_0] AddExpr ← MulExpr;
    [ADD_1] AddExpr ← AddExpr "+" MulExpr;
    [ADD_2] AddExpr ← AddExpr "-" MulExpr;
  }
  role(evaluation) {
    [ADD_0] @{ $ADD_0[0].value = $ADD_0[1].value; }
    [ADD_1] .{ /*...*/ }.
    [ADD_2] .{ /*...*/ }.
  }
}
slice neverlang.js.AddExprSlice {
  concrete syntax from neverlang.js.AddExpr
  module neverlang.js.AddExpr with role evaluation
}
language neverlang.js.TinyLang {
  slices neverlang.js.AddExprSlice neverlang.js.UnaryExprSlice
         /* ... */
  roles syntax < evaluation
  rename { MulExpr → UnaryExpr; }
}
```

Listing 2: **Neverlang's `slice` and `language` constructs.**

4.1 Neverlang in a Nutshell

Neverlang promotes code reuse and sharing by making language units first-class concepts. 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. The base unit is the **`module`** (Listing 2). A module may contain a **`syntax`** definition and/or semantic **`role`**s. A role defines actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique. Syntax definitions are portions of BNF grammars represented as sets of *grammar rules* or *productions*. Semantic actions are defined as code snippets that refer to production rules and the contained nonterminals with the prefixed labels. Syntax definitions and semantic roles are tied together using **`slices`**. For instance, module `neverlang.js.AddExpr` in Listing 2 declares a reference syntax for addition and subtraction, and actions are attached to each of the three productions by referring to their label. The slice `neverlang.js.AddExprSlice` declares to use this concrete syntax in our language together with that corresponding semantics. Finally, the **`language`** descriptor (Listing 2), indicates which slices are required to be composed together to generate the compiler for the language. The **`language`** descriptor is the cornerstone of the whole mechanism and allows for easily composing, restricting and extending a programming language.[5] More importantly, each **`module`** can additionally declare the set of nonterminals it **`requires`** and **`provides`**; each accompanied by a number of tags.

Note that since Neverlang compiles all its artifacts into JVM compatible classes, the language development environment can be used on any JVM compliant target platform.

4.2 Developing Language Product Lines

Henceforth, we employ Neverlang to develop a language product line. Fortunately, a **`module`** corresponds to a language component, as it can be used to encapsulate the syntactical and semantical aspects of a language feature in a

[5]Neverlang details can be read in [36].

reusable way (R2). In addition to that, for each **module** a **slice** has to be created accordingly. Thus, after decomposing the language family (R1), each language feature is implemented as **module** accompanied by a **slice**. Moreover, each nonterminal provided by a module must be tagged to enable the feature model generation. The implementation of the Add language component from Sect. 2.2, for instance, is shown in Listing 2. Its syntax is defined by three BNF productions, equivalent to the EBNF rule in Fig. 1, and its semantics is implemented within the *evaluation* **role**. After implementing each language component, an arbitrary language variant can be created (R4) by stating the selected **slices** to compose within the **language** descriptor and specifying the rewired nonterminals within the **rename** section. The descriptor, in Listing 2, creates a variant of the expression language without products and divisions by excluding the MulExprSlice and rewiring the MulExpr nonterminal. Furthermore, Neverlang provides special features to access the required and provided nonterminals of a **module** together with their tags, and to retrieve the dependency graph created by the Neverlang compiler (R3) [38, 37]. As such, Neverlang is a suitable framework for the implementation of LPLs. Nonetheless, manually writing **language** descriptors is intractable to configure variants of general-purpose programming languages. Thus, a tool is needed that guides users towards the creation of valid language variants automatically generating the corresponding **language** descriptors.

4.3 The AIDE Tool

AiDE is an interactive configuration tool especially tailored to these needs. It implements the presented method to synthesize the *feature model* of a given language family out of language components developed with Neverlang. Through its graphical user interface, depicted in Fig. 4, the user can explore the feature model, choose features, create a language variant, and test it. Because feature models of LPLs tend to be huge, the tool initially shows the first level of the tree, however, allowing it to be expanded on demand. A feature is selected or deselected by clicking on it, this selects all its parents or deselects all its children, respectively. Selected features are highlighted in green. To not confuse the user, unique and alternative dependencies of selected features are highlighted in red and yellow, respectively. Besides, optional features are linked by white edges, whereas mandatory features by black edges. Moreover, while the user configures a language variant (or product), AiDE keeps track of all unresolved dependencies, i.e., all open nonterminals of the selected language components and lists them on the left-hand side allowing the user to rename (or rewire) them, i.e., binding them to another provided nonterminal. Thus, users can easily resolve dependencies during the component picking. Another important feature of AiDE is its ability to dynamically update the language variant during its configuration. Whenever a valid configuration, i.e., one without unresolved dependencies, exists, the user can update the internal language variant and test it using the integrated command line interface of Neverlang. This, permits users to verify the consistency and test the behavior of the language variant under construction. Internally, AiDE updates the **language** descriptor maintained by the underlying Neverlang language development framework. When the language satisfies the expectations, a stable copy of the development environment is prepared and ready to be dispatched to any JVM compliant workstation. In sum, AiDE is able to guide users towards the generation of consistent language variants by supporting multiple dependency resolution strategies (R5) and continuous generation of the language's compiler/interpreter.

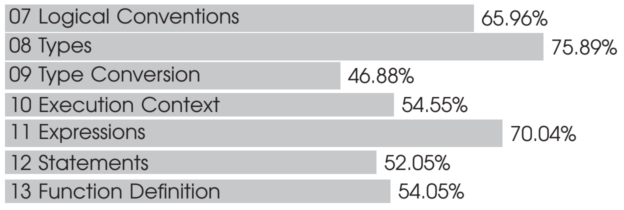


Figure 5: Javascript coverage of Neverlang.JS

5. DEMONSTRATION CASE STUDY

This section demonstrates the feasibility of the presented approach and tooling by developing a LPL for Javascript and report on its applicability for the generation of both language specializations and extensions [28].

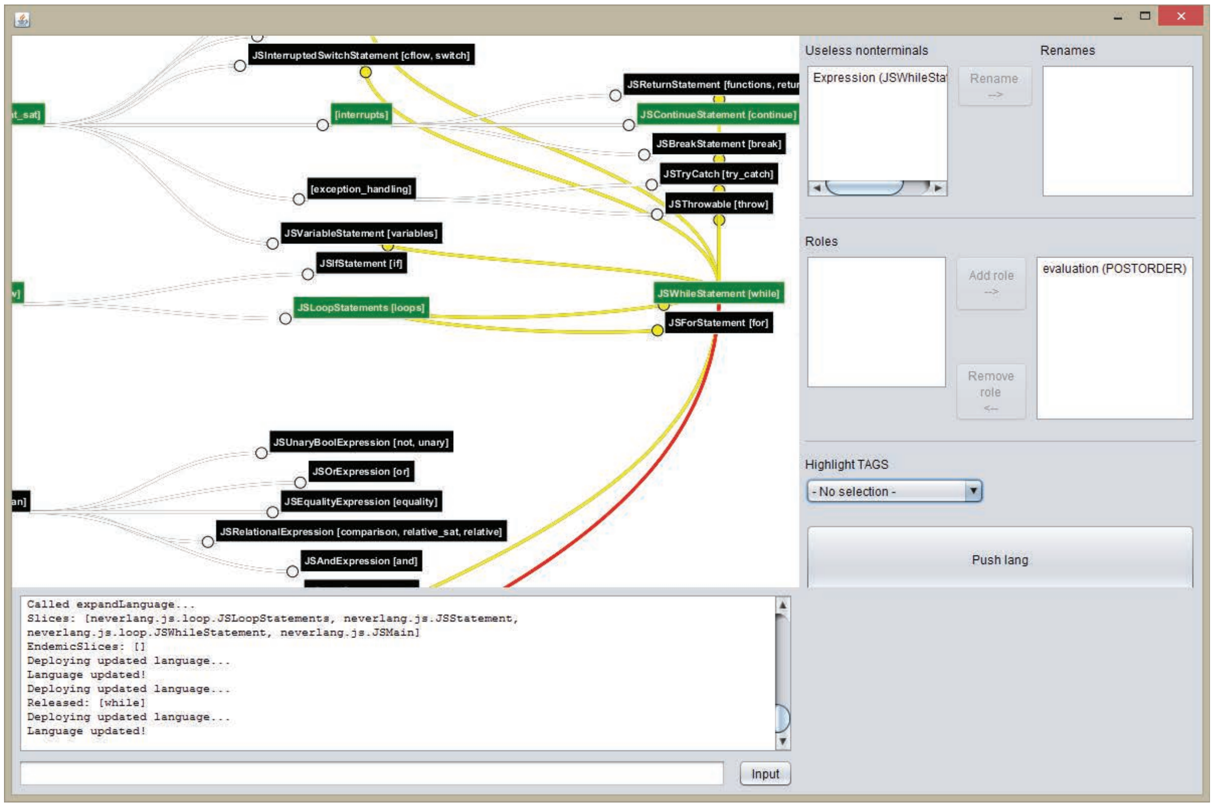
5.1 Neverlang.JS

Over the course of roughly two months we decomposed Javascript into its language features and implemented each as a Neverlang **module** (and **slice**). Especially, each nonterminal provided by a **module** was tagged to fully describe its nature. Moreover, Neverlang.JS was developed to be a Javascript interpreter conform to the ECMAScript 3 Language Specification (ECMA-262) and consists of 73 **slice**s that accumulate to 3043 lines of code and 228 production rules [36]. Tab. 2 shows the detailed number of slices, rules, and lines of code (LOC) for each implemented language feature. In addition to that, 64 Java classes were implemented to support some of Javascript's various built-in objects. In general, the feature complete variant of Neverlang.JS covers about 70% of the corresponding language specification, according to the Sputnik test suite.[6] Fig. 5 shows an excerpt of the language coverage report indicating good results regarding the *logical conventions*, *types*, and *expressions*. Please note, that the test suite assumes that *all* of the built-in libraries have been implemented, which is merely a technicality; this is currently out of the scope of Neverlang.JS and does not compromise the soundness of the experiments. Besides that, Neverlang.JS fully supports the semantics of Javascript including constructs as the *prototype chain*, *exception handling*, and *closures*. In particular, our implementated language provides a default *value* attribute of type JSType for each nonterminal. This type is a superclass to all possible Javascript types and allows modules to implicitly retrieve, convert, and store values in a transparent fashion. This ensures the compatibility of all language components including components that were not intended to be combined. In summary, Neverlang.JS is a fully tagged, decomposed version of the general-purpose programming language Javascript, and thus a viable LPL.

To generate, explore, and configure the feature model of Neverlang.JS, AiDE was employed. In particular, the resulting feature tree has a maximum depth of 6 and contains 92 nodes, wheres 19 are abstract features and 73 language features. Due to space restrictions, only a small part of the

[6]https://code.google.com/p/sputniktests/

Final edited form was published in "SPLC '15: 2015 International Conference on Software Product Lines. Nashville 2015", S. 71–80. ISBN: 978-1-4503-3613-0
https://doi.org/10.1145/2791060.2791092



```
Called expandLanguage...
Slices: [neverlang.js.loop.JSLoopStatements, neverlang.js.JSStatement,
neverlang.js.loop.JSWhileStatement, neverlang.js.JSMain]
EndemicSlices: []
Deploying updated language...
Language updated!
Deploying updated language...
Released: [while]
Deploying updated language...
Language updated!
```

Figure 4: Screenshot of the AiDE application representing the feature model of Neverlang.JS.

feature model is shown in Fig. 4, however a fully expanded version of the feature model is available online.[7] Using AiDE we constructed many variants of Neverlang.JS including both restricted and extended variants, discussed henceforth.

5.2 Feature Restriction

The configuration of products is the general use case of SPLs and LPLs alike. However, for language products this usually coincides with restricting a feature complete language variant. In case of Neverlang.JS, a feature complete variant is comparable to Javascript, whereas a variant with fewer features will create a specialization of Javascript [28]. As indicated previously, these language specializations can be used for gradually teaching programming languages. To test this idea, we employed both Neverlang.JS and AiDE in [9] to generate 13 variants of Javascript according to the learning stages. These variants were aligned to the individual learning stages that successively introduce language features ranging from *numeric expressions* up to *constructors* and the *prototype model*. Fig. 6 shows the successive learning stages and language features introduced in each stage of the experimental course described in [9]. Our experience during the preparation of this course convinced us that AiDE is able to guide teachers towards the creation of language specializations in accordance to the desired structure of the course. In particular, its support of alternative dependency resolution strategies and continuous testing of the language variant under construction provides simple means to create working language specializations of the Neverlang.JS LPL.

[7]http://neverlang.di.unimi.it/aide/njs_graph.png

Table 3: List of Javascript Extensions, from [36]

Extension Name	LOCs
Function Type Annotations	225
Catch Guards	80
Class-Based Single Inheritance	314
Dictionary Comprehension	79
Destructuring Assignment	73
Tuple Literal	91
List concat operator	91
Lambda Expressions	76
Named Arguments in Functions	78
List Sum Operator (Vector Sum)	41
Pipe Forward Operator	92
Immutable References	31
List Comprehension	81
Syntax for Pattern Matching	191

5.3 Feature Extension

While feature restriction might be the usual case to create a language product, feature extension represents a more interesting use case. As argued in Sect. 2, new language features can be easily added to an LPL by simple adding new language components and regenerating the language's feature model. To evaluate, whether this also holds for Neverlang.JS, 14 PhD students were assigned to implement new language features [36]. Each language feature introduces a new language construct ranging from simple extensions, like *immutable references*, to very complex extensions, like *class-based single inheritance*. Tab. 3 list all the developed lan-

Final edited form was published in "SPLC '15: 2015 International Conference on Software Product Lines. Nashville 2015", S. 71–80. ISBN: 978-1-4503-3613-0
https://doi.org/10.1145/2791060.2791092

Table 2: Size of the Neverlang.JS implementation per language feature, from [36]

Bundle	Slices	LOC	Rules
Core			
Language core	11	277	24
Expressions			
Arithmetic	3	128	9
Boolean	3	92	5
Relational	2	137	10
Conditional	1	32	2
Bitwise	5	216	17
Typing (typeof, instanceof)	2	65	2
Function call	2	113	9
Construct call	1	56	3
Types			
String	1	21	2
Number	1	24	3
Boolean	1	23	3
RegExp	1	23	2
Object	4	189	13
Array	3	131	9
Function (definition)	2	100	11
This resolution	1	17	1

Bundle	Slices	LOC	Rules
Statements			
Block Statement	1	32	4
Cflow			
If Statement	1	45	3
Switch Statement	1	102	12
(Loop Statements)	*1*	*19*	*1*
While Statement	1	50	2
For loop	1	57	7
For-each loop	1	113	10
(NoIn expressions integration)	*11*	*305*	*26*
Interrupt: break	1	22	2
Interrupt: continue	1	22	2
Interrupt: return	1	30	3
Exception throwing + handlin	2	122	8
Variables			
Variable assignment	5	226	21
Variable resolution	1	24	2
Endemic Slices			
Symbol Table		230	

Italicized features depend on other features: *loop statements* require at least one actual loop implementation (*e.g.*, `while`, `for`, etc.), *No-In expressions* are part of the ECMAScript spec and depend on the definition of *for-each*.

Total Slices	**73**
Total LOC	**3043**
Total Rules	**228**

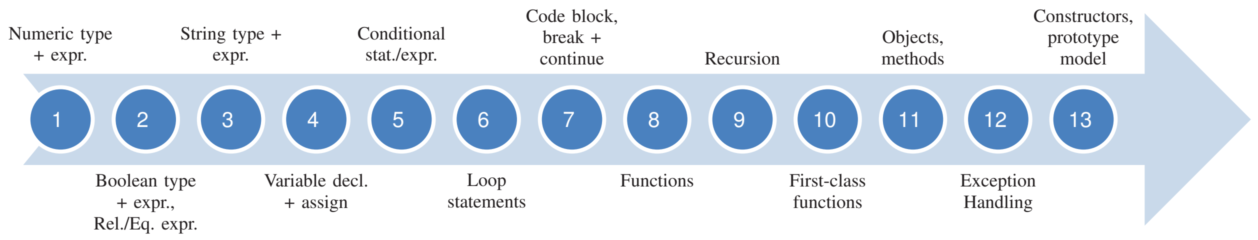


Figure 6: Language variants of Neverlang.JS used for teaching, from [9]

guage extensions together with the size of its implementation in Neverlang. Notably, most language features were implemented in less then one hundred lines of code. We tested all language extensions first individually and afterwards all possible combinations. As a result, most extensions were compatible and only few had conflicts due to similar syntactic definitions. This indicates that language families simplify the creation of language extensions to a point were language features can be implemented independently. Above all, as AiDE is able to automatically generate a corresponding feature model for Neverlang LPLs, given that each extension is consistently tagged, there is no need to manually refine the feature model for each additional language feature. The **slice** and **module** implementing the new features must simply be added to the set of files loaded by AiDE. Moreover, as AiDE allows users to test language variants during configuration, incompatibilities between new language components can be immediately detected, as the Neverlang framework will raise an error in that case. In sum, our experience with Neverlang.JS shows that both the presented feature model generation approach and the language configuration tool, AiDE, is suitable for LPLs of general-purpose programming languages.

6. RELATED WORK

A number of authors have tried to address the problem of extracting a feature model from various kinds of artifacts. She *et al.* [33] showed how to reverse engineer a feature model starting from natural language *feature descriptions* and static analysis of source code. Davril *et al.* [16] 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.* [31] use clustering techniques to infer a tree structure. Ferrari *et al.* [19] considered natural language documents. Weston *et al.* [42] extract feature models from the requirements description in natural language. All these works require *extra* information to be associated with the concrete implementations of the features. Information not always available, and, as in the case of the domain of programming languages, often overrated with respect to its usefulness. In this work, we move slightly away from our previous work [37], in that, instead of relying on a *semantic network*, we mine metadata already present in the language features. As compared to natural language feature descriptions, tagging makes it simpler to mechanically compare feature implementations, and, as compared to semantic networks, it requires less work by the domain experts. The dependency graph, on the other hand, is still exploited to superimpose natural relations between the language features onto the generated feature tree.

FAMILIAR [1] provides an environment to synthesize feature models from propositional formulae. An interactive support (through ranking lists, clusters, and logical heuristics) for choosing a sound and meaningful hierarchy is part of the environment [7]. 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, 7] is not immediate and requires some user effort.

Many formalisms were proposed in the past decade for variability modeling; we refer to the work of [32, 23, 13] for a survey on known solutions.

Some work has applied variability management to language implementation. Although we used Neverlang [10, 8], other modular language implementation frameworks can be employed to implement a similar approach (cf., [40, 22]). Cengarle *et al.* [12] use MontiCore [25] to describe variations of a base language. Haugen *et al.* [21] have used CVL to model possible DSL variations. White *et al.* [43] use feature modeling to improve reusability of features among a language family. In Liebig *et al.* [27] and Wende *et al.* [41] a family of languages is decomposed in terms of their features using Spoofax [24] and LanGems [41], respectively. The main differences to these proposals are that they either factorize the language and then manually map the features onto a feature model or they start from pre-implemented languages and design the feature model that brings their features together. In any case, the feature model is not inferred in an automated or at least semi-automated way from a set of pre-implemented features. In our approach the relations between the components are inferred using metadata (tags) that can be directly extracted from the implemented language components.

7. CONCLUSION AND FUTURE WORK

Both modular language development and feature-oriented software software development are current research topics. We have shown that by combining them, the process to produce a language variant out of a modularly implemented language can be simplified by using product line concepts (*language product line*). The paper describes how from a set of properly classified language components a feature model can be automatically generated. The feature model can be easily inspected and language features can be picked to form the desired variant of the language; missing dependencies can be resolved in two alternative ways. Language extensions and specializations are supported, as well.

The language product line approach is independent of the adopted development framework. The imposed requirements have been discussed. To demonstrate the feasibility of the language product line approach, it has been described in the case of Neverlang; a support tool (called AiDE) for choosing and picking the desired language components and form the final language product has been developed.

Finally, we have developed a language product line for a modular implementation of Javascript (Neverlang.JS). The complexity of the language is big enough to provide a high number of language components and a significant number of language variants; this facilitates an interesting playground for testing language specializations and extensions. Some real applications of these cases have been proposed, as well.

Only one feature of typical SPL approaches is currently not supported: *their ability to estimate the number of valid variants*. Although the usual approach of transforming the feature model to a logical formula and estimating the number of satisfying interpretations is usually an overestimation in the case of SPLs, it most certainly will be an underestimation for the number of variants in an LPL. This is because a missing dependency can also be satisfied by rewiring an open nonterminal. Hence, the number of variants generated in this way can exceed the estimated number of variants by an order of magnitude. In a worst-case scenario, the estimation must consider any possible rewiring from one nonterminal to another; this amounts to considering at most 2^n variants, where n is the number of nonterminals. Consequently, we need a more accurate estimation technique that not only take the generated feature model into account, but also the number of rewirings that fulfill dependencies and connect compatible language components. As it is impractical to create all possible combinations to evaluate the compatibility between language components, we currently investigate to exploit tags to limit the number of valid rewirings. However, further research is required to develop a reliable estimation mechanism suitable for LPLs.

Acknowledgements

This work has been partially supported by the MIUR project "CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society" as well as the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

8. REFERENCES

[1] M. Acher, B. Baudry, P. Heymans, A. Cleve, and J.-L. Hainaut. Support for Reverse Engineering and Maintaining Feature Models. In *VaMoS'13*, Jan. 2013. ACM.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[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 *SPLC'08*, pp. 67–76, Sept. 2008.

[4] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient Synthesis of Feature Models. In *Proc. of SPLC'12*, pp. 97–106, Salvador, Brazil, Sept. 2012.

[5] S. Apel and C. Kästner. An Overview of Feature-Oriented SW Development. *J. of Obj. Tech.*, 8(5):49–84, 2009.

[6] S. Apel, C. Kästner, and C. Lengauer. Language-Independent, Automated Software Composition. In *ICSE'09*, pp. 221–231, Vancouver, Canada, May 2009.

[7] G. Bécan, S. Ben Nasr, M. Acher, and B. Baudry. WebFML: Synthesizing Feature Models Everywhere. In *Proc. of SPLC'14*, Florence, Italy, Sept. 2014.

[8] W. Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *SC'12*, LNCS 7306, pp. 162–177, Prague, Czech Republic, June 2012.

[9] W. Cazzola and D. M. Olivares. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Trans. on Emerging Topics in Computing*, 4(1), Jan. 2016.

[10] W. Cazzola and E. Vacchi. Neverlang 2: Componentised Language Development for the JVM. In *SC'13*, LNCS 8088, pp. 17–32, Budapest, Hungary, June 2013.

[11] W. Cazzola and E. Vacchi. On the Incremental Growth and Shrinkage of LR Goto-Graphs. *ACTA Informatica*, 51(7):419–447, Oct. 2014.

[12] M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within Modeling Language Definitions. In *MoDELS'09*, LNCS 5795, pp. 670–684, Oct. 2009. Springer.

[13] L. Chen and M. Babar. A Systematic Review of Evaluation of Variability Management Approaches in SW Product Lines. *J. Inf. & SW Tech.*, 53(4):344–362, 2011.

[14] Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, Reading, MA, USA, 2000.

[15] M. L. Crane and J. Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *MoDELS'05*, LNCS 3713, pp. 97–112. 2005.

[16] 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 *ESEC/FSE'13*, pp. 290–300, Aug. 2013.

[17] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language Composition Untangled. In *LDTA'12*, Mar. 2012.

[18] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-Based Syntactic Language extensibility. In *OOPSLA'11*, pp. 391–406, Oct. 2011. ACM.

[19] A. Ferrari, G. O. Spagnolo, and F. Dell'Orletta. Mining Commonalities and Variabilities from Natural Language Documents. In *SPLC'13*, pp. 116–120, Sept. 2013.

[20] D. Ghosh. DSL for the Uninitiated. *Commun. ACM*, 54(7):44–50, July 2011.

[21] 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.

[22] 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.

[23] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A Preliminary Review on the Application of Feature Diagrams in Practice. In *VaMoS'10*, pp. 53–59, Jan. 2010.

[24] L. C. L. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. *OOPSLA'10*, pp. 444–463, Oct. 2010.

[25] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *Int.l J. on SW Tools for Technology Transfer*, 12(5):353–372, Sept. 2010.

[26] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *SLE'14*, LNCS 8706, pp. 141–160, Västerås, Sweden, Sept. 2014. Springer.

[27] J. Liebig, R. Daniel, and S. Apel. Feature-Oriented Language Families: A Case Study. In *VaMoS'13*, Jan. 2013.

[28] 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.

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

[30] K. Ng, M. Warren, P. Golde, and A. Hejlberg. The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis. White paper, Microsoft, Oct. 2011.

[31] N. Niu and S. Easterbrook. On-Demand Cluster Analysis for Product Line Functional Requirements. In *SPLC'08*, pp. 87–96, Sept. 2008. IEEE.

[32] K. Pohl and A. Metzger. Variability Management in Software Product Line Engineering. In *ICSE'06*, pp. 1049–1050, Shanghai, China, May 2006. ACM.

[33] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse Engineering Feature Models. In *ICSE'11*, pp. 461–470, May 2011. IEEE.

[34] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Sc. of Comp. Progr.*, 79(1):70–85, Jan. 2014.

[35] L. Tratt. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Trans. Prog. Lang. Syst.*, 30(6):31:1–31:40, Oct. 2008.

[36] E. Vacchi and W. Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 2015.

[37] E. Vacchi, W. Cazzola, B. Combemale, and M. Acher. Automating Variability Model Inference for Component-Based Language Implementations. In *SPLC'14*, pp. 167–176, Florence, Italy, Sept. 2014.

[38] E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability Support in Domain-Specific Language Development. *SLE'13*, LNCS 8225, pp. 76–95, Oct. 2013.

[39] E. Vacchi, D. M. Olivares, A. Shaqiri, and W. Cazzola. Neverlang 2: A Framework for Modular Language Implementation. In *Proc. of Modularity'14*, pp. 23–26, Lugano, Switzerland, Apr. 2014. ACM.

[40] 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.

[41] C. Wende, N. Thieme, and S. Zschaler. A Role-Based Approach towards Modular Language Engineering. In *SLE'09*, LNCS 5969, pp. 254–273, Denver, Oct. 2009.

[42] N. Weston, R. Chitchyan, and A. Rashid. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In *SPLC'09*, pp. 211–220, Aug. 2009. ACM.

[43] 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.