# Feature-Oriented Language Families: A Case Study

Jörg Liebig
University of Passau
Germany

Rolf Daniel
msg systems
Germany

Sven Apel
University of Passau
Germany

## ABSTRACT

Software-language engineering is gaining momentum in research and practice, but it faces many challenges regarding language evolution, reuse, and variation. We propose *language families*, a feature-oriented approach to language engineering inspired by product lines and program families. The goal is to systematically manage the development and evolution of variants and versions of a software language in terms of the *language features* it provides. We offer a tool chain for the development and management of composable language features and language families. By means of a case study on the Web-programming language Mobl, we evaluate the practicality of our approach and discuss our experiences, open issues, and perspectives.

## Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.3.2 [**Programming Languages**]: Language Classifications—*Extensible languages*

## General Terms

Languages

## Keywords

Language families, language evolution, feature-oriented programming

## 1. INTRODUCTION

Software languages are widely used in many areas of software engineering. Beside general-purpose programming languages, software languages comprise domain-specific languages, modeling and specification languages, pattern languages, application programming interfaces, and ontologies. The emerging paradigm of software-language engineering is concerned with the systematic development, use, and maintenance of these languages; this includes design, implementation, testing, and evolution of software languages.

Software languages are rarely designed and implemented in a single step. Instead, software languages are subject to incremental development and evolution, giving rise to different variants and versions of the language [39]. Even general-purpose languages evolve over time (e.g., Java 5 = Java 4 + generics) and pose major challenges on software-language engineers (e.g., tool builders).

Inspired by product-line and program-family approaches [1, 15, 17, 33], we propose the concept of a language family to systematically manage the development and evolution of variants and versions of software languages. A *language family* is a set of languages that share a base of common language features, but that also differ in certain language features. The goal of the language-family engineering is to facilitate reuse, variability, and automation: (1) language features are reused among the individual languages of the family; (2) different languages satisfy different requirements of different stakeholders by providing corresponding features; and (3) a language is specified and derived automatically based on a user's feature selection.

A language family targets a particular domain, such as data modeling, Web programming, query processing, or circuit design. The features of the language family represent abstractions of the target domain. For example, a language for query processing may provide features for supporting spatial and streaming queries, but in many application scenarios these two language features are not necessary, and—although not used in a particular scenario—they may even have negative effects, such as performance penalties, accidental complexity, and misbehavior.

Our proposal of language families aims at supporting the practical development and evolution of software languages. We propose to decompose a language definition along the language features it provides. This is in the spirit of modular language (de)composition, but it goes beyond that in that we employ non-classic modularization mechanisms (i.e., crosscutting decomposition using mixins), and we systematically manage variability among the family members, which is inspired by recent work on software product lines [4, 5, 10, 17]. For example, a query language could be decomposed into a base language and features for stream processing, spatial queries for sensor networks, and view management. Based on a user's feature selection, a simplified language variant—along with corresponding tools (e.g., compiler and editor)—can be generated on demand, excluding unnecessary language features that may result in disadvantages such as distraction of programmer's attention, performance reduction, and error introduction. On the contrary, nowadays languages have grown to a level of complexity that makes them hard to maintain and evolve, with the result of the mentioned disadvantages [39]. For example, in the database community it has been shown that "one size fits all" is not feasible in practice for SQL [40]. Instead, the trend goes toward specialized SQL dialects to downsize and simplify the corresponding language tools (e.g., analysis tools), to give the programmer a chance


*VaMoS '13,* January 23 - 25 2013, Pisa , Italy

to understand the language's features and their interactions, and to exploit optimization potential [36].

In addition to recent work on language (de)composition [13, 21, 24, 27], which concentrates on syntax, we consider all aspects of a language, including syntax, static and dynamic semantics, analysis and optimization, as well as documentation. Furthermore, we focus on technical aspects rather than on formal semantics. We provide means (in the form of a tool chain) to decompose a language definition along its features, and compose the pieces consistently based on a user's feature selection. We base our approach on recent advancements in software-language engineering. As a technological basis, we integrate and extend a number of existing languages and tools including SDF [44] and Stratego [12] for language specification, Spoofax for generating language tools [26], and FeatureHouse for composing language features (i.e., the corresponding documents and specifications) [4]. In a case study, we used this tool chain to refactor the existing language Mobl for mobile Web applications [23] into features and to compose them in different combinations to create different language variants. Here, we reflect on our experience with feature (de)composition of Mobl and discuss open issues and perspectives.

In summary, we make the following contributions:

- We introduce the concept of a language family, which is centered around the concept of composable language features.
- We provide an approach and a tool chain for the development and management of language families.
- By means of a case study on the Web-programming language Mobl, we evaluate the practicality of our approach and discuss our experience, open issues, and perspectives.

## 2. LANGUAGE FAMILIES

After introducing a running example, we introduce the concept of a language family and discuss issues such as variability and consistency.

*Running Example.* As our running example, we use a small language for expressions [34]. For now, it supports only boolean values and a conditional construct. We call the language $\text{EXPR}_{\mathbb{B}}$. In Figure 1, we provide a specification of its syntax and semantics in terms of grammar rules (left), typing rules (middle), and evaluation rules (right). The syntax is specified in Backus-Naur-Form. It comprises three syntactic forms (boolean constants and conditional statement) and distinguishes between terms and values. The type system consists of three typing rules (one for each syntactic form) that assign types to terms (in our case, we have only type Bool). Similarly, there is one evaluation rule per syntactic form, of which the last one is merely a congruence rule. Further details are outside the scope of our discussion and we refer the reader elsewhere [34].

Much like the syntax and semantics specifications, we can define rules for syntax highlighting and optimization, and we could include corresponding proofs of soundness and completeness of the type system, documentation, test cases, etc.

*Language Features.* The idea of a language family is not to specify a single language, but a *whole set* of related languages. The different languages of a family are related by the features they provide. Typically, a language shares features with other languages, but has also some unique ones. A key idea is to make the language features explicit by factoring them out of the language specification in terms of composable units, called henceforth *language feature units*. The expression language we have discussed so far can be seen as one unit. On top of it, we can define further units that extend and refine existing specification documents incrementally (cf. Fig. 3; left).

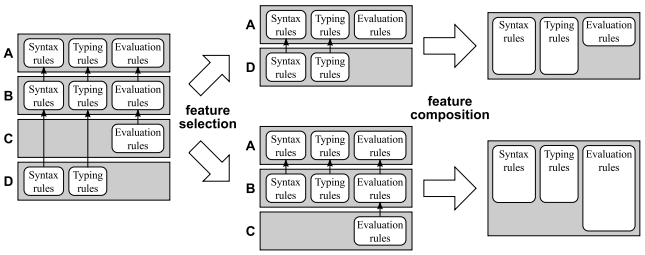


**Figure 3: Composing language specifications on demand; the example consists of the four features A–D, of which two combinations are selected and composed**

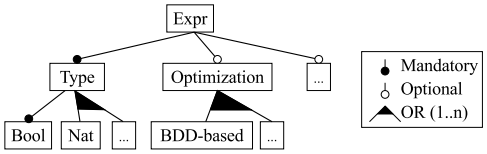


**Figure 4: A feature model of the family of expression languages**

As a concrete example, suppose we want to extend $\text{EXPR}_{\mathbb{B}}$ by a feature for supporting arithmetic expressions (i.e., natural numbers and corresponding operations). To this end, we introduce a new language feature unit, called $\text{EXPR}_{\mathbb{N}}$, that encapsulates the definitions concerned with arithmetic expressions and that builds on the definitions of $\text{EXPR}_{\mathbb{B}}$. In Figure 2, we show $\text{EXPR}_{\mathbb{N}}$'s syntax, typing, and evaluation rules. The dots ('...') within the new rules added by $\text{EXPR}_{\mathbb{N}}$ denote the locations where the rules of $\text{EXPR}_{\mathbb{B}}$ are extended. Essentially, $\text{EXPR}_{\mathbb{N}}$ introduces a number of new syntactic forms (constant numbers, successor, predecessor, and zero test). Furthermore, it introduces the new type Nat as well as straightforward typing and evaluation rules for the new syntactic forms. Details of the rules are available elsewhere [34].

*Families of Languages.* Once we have multiple language feature units, a user can compose them in different combinations, in our simple case, (1) boolean expressions *only* and (2) boolean *and* arithmetic expressions. In our example, we use superimposition as a composition operator [4, 5], which merges the rules of two documents recursively by name. While superimposition has proved useful for the extension of grammars, other composition operators such as weaving [11] or inheritance [27] are possible.

The user can compose language features in different combinations, as illustrated in Figure 3. However, in practice, not all combinations may be valid and represent desirable languages, so we need to constrain the set of possible feature combinations properly. We use feature models for this task. In Figure 4, we show a possible feature model for our expression language family. Using feature models, a language engineer can model complex relations between language feature units [17].

*Dimensions of Variability.* A key observation is that a family of languages induces two dimensions of variability [9]. First, we have the language features that a user can combine on demand (e.g., booleans and natural numbers or stream processing and spatial queries). Second, we have the different kinds of specification documents relevant for different kinds of language-processing tools (e.g., grammar rules and typing rules). To distinguish the choices of tools to be generated from the language features, we call them *tool*

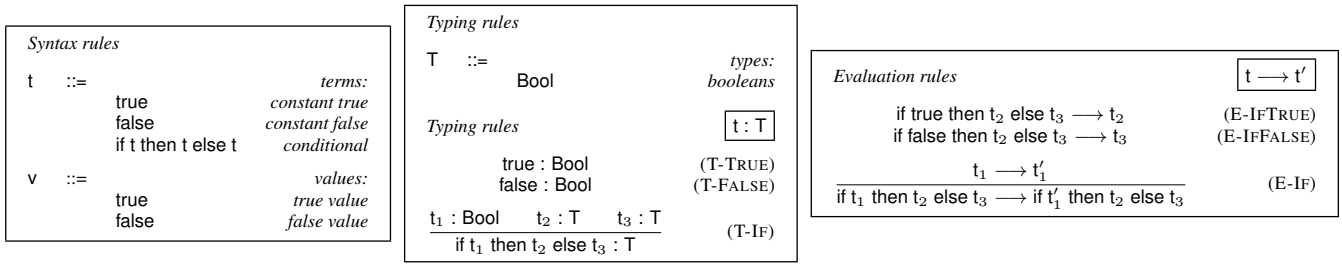


**Figure 1: Specification syntax and semantics $\textsc{Expr}_{\mathbb{B}}$**

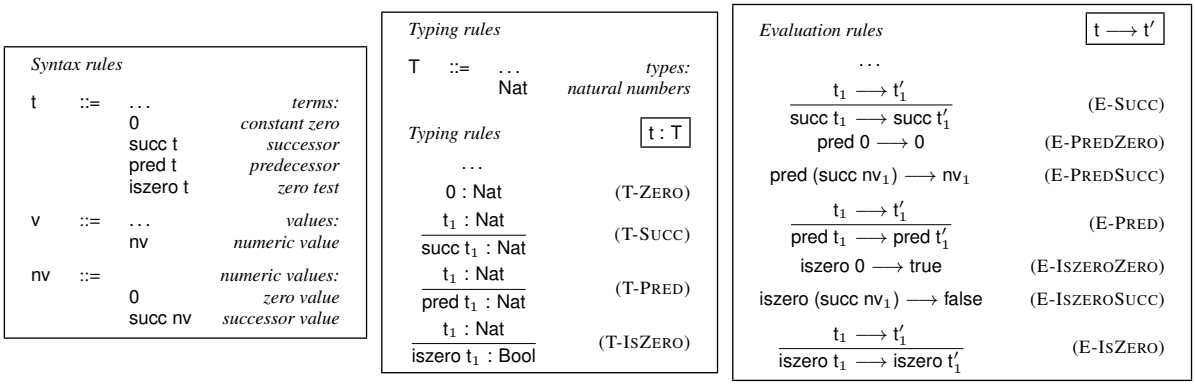


**Figure 2: Syntax and semantics specification of $\textsc{Expr}_{\mathbb{N}}$, which extends $\textsc{Expr}_{\mathbb{B}}$**

| | *Syntax* | *Typing* | *Evaluation* | *Optimization* | *Proofs* | *Docs* | ... |
|---|---|---|---|---|---|---|---|
| Bool | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ... |
| Nat | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ... |
| String | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ... |
| Lambdas | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ... |
| Variables | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ... |
| ... | ... | ... | ... | ... | ... | ... | ... |

**Table 1: Two dimensions of variability in the family of expression languages**

*features*. We illustrate both dimensions for the expression-language family in Table 1.

Note that, when adding a feature to one dimension, it is likely that we have to change and extend the features (i.e., their representation) of the other dimension. For example, when adding support for strings to the expression language, we have to add corresponding syntax, typing, and evaluation rules, we have to update the soundness proof, we have to extend the documentation, and so on. Or, when adding support for syntax highlighting, we have to extend the representations of all language features in order to define proper syntax-highlighting rules, for conditionals, lambdas, functions, etc.

The two-dimensional structure of Table 1 reveals that there are many interactions between language and tool features, which makes it even more important to make features and variability explicit. The situation is very similar to issues discussed in product-line architecture and programming languages. For example, Batory et al. [9] observed that features in product lines can be assigned to dimensions that form multi-dimensional program cubes; folding cubes dimension-wise corresponds to feature composition. The famous *expression problem*[1] illustrates a similar problem: Given a set of recursive data types (e.g., different kinds of terms and expressions) and a number of operations defined over them (e.g., evaluation and pretty printing), it is challenging to extend the program both by new data types and operations in a modular way [43].

Software-language engineers should be aware of the fact that languages have different dimensions of variability, which leads, in the worst case, to an explosion of interactions between language and tool features [25]. Our proposal of language families makes the dimensions explicit and provides guidelines for (de)composition. For example, if we want to generate a parser and type checker for the expression language with booleans and natural numbers, we can see from Table 1 that we have four units to compose:

Bool#Syntax, Nat#Syntax, Bool#Typing, Nat#Typing

where A#B denotes the part of a language feature A that is concerned with tool feature B.[2] Given a selection $l \subseteq \mathbb{L}$ of language features and a selection $s \subseteq \mathbb{S}$ of specification types, the desired language variant is composed via $\bullet_{i=1}^{|l|} \left( \bullet_{j=1}^{|s|} (l_i \# s_j) \right)$, where $\bullet$ denotes composition of feature units.

Decomposing a language specification along features and managing dimensions of variability, allows a user/tool to compose language variants effectively and consistently (cf. Fig. 5). Looking at Table 1, we gain a maximum of reuse of feature units and variety of corresponding languages. The mapping of dimensions and features to units enables the automatic derivation of languages and corresponding tools based on a user's feature selection.

# 3. TOOL CHAIN

We developed a tool chain for language-family engineering. The tool chain rests on a number of existing tools that we adapted and integrated for language families. First, there is a number of tools for the specification of software languages and the generation of cor-

[1]The expression problem was named by Phil Wadler in 1998 but has been known for many years [16, 28, 35].

[2]The notation is based on work of Batory et al. [8].

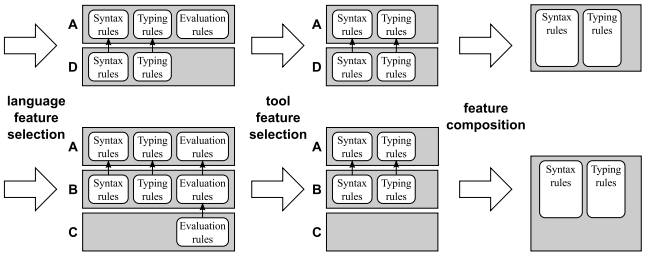


**Figure 5: Language families gives rise to two dimensions of variability: (1). selecting language features and (2). selecting tool features**

responding language tools, and, second, there are tools for the representation, management, and composition of language features.

*Language Specification.* As discussed in Section 2, we specify a language in terms of its syntax and semantics rules. For defining the syntax, we use SDF [44], an expressive and concise syntax definition formalism and tool. We chose SDF because it has a number of desirable properties: SDF integrates lexical and context-free syntax in a single formalism. It is a declarative language and supports arbitrary context-free grammars. That is, a syntax definition can be used for different purposes, including the generation of parsers, pretty printers, and data type definitions. SDF syntax definitions can be modular, enabling their reuse in different syntax definitions.

For the definition of the semantics of a software language, we use Stratego [12]. Stratego is a domain-specific language for program transformation. It is based on programmable rewriting strategies, which makes Stratego useful for traversing, analyzing, and transforming syntax trees. In particular, we use it to define typing rules and code-generation rules for software languages, which boils down to traversing and transforming abstract-syntax trees.

To generate language tools from the syntax and semantics definitions, we use Spoofax [26], a language workbench for the development of textual software languages. In particular, we use Spoofax to generate parsers based on SDF grammars and type checkers, and code generators and interpreters based on declarative rules sets written in Stratego. Furthermore, Spoofax can be used to generate Eclipse-based editor tools based on declarative rules, including syntax highlighting and code completion as well as on-line reference resolution.

*Language Features.* Using our tool chain, we specify a language by providing SDF grammar and Stratego typing and code-generation rules. For the development of a language family, we have to (de)compose the documents that contain the rules along the language features that the family is supposed to provide. (From right to left in Figure 3.)

For representing and composing language features, we use FeatureHouse [4], which is a tool chain for software composition based on superimposition. We extended FeatureHouse with support for SDF and Stratego, so that language feature units consisting of SDF and Stratego documents can be composed based on a user's feature selection. The extension includes writing parsers and pretty printers for SDF and Stratego documents, as well as providing composition rules for SDF and Stratego documents. In Figure 6, we show a Stratego document defining basic typing rules (top, feature Base) for our Mobl case study and a refinement of that document that adds typing rules for HTML (bottom, feature HTML). SDF and Stratego documents are represented in FeatureHouse using a tree-like data structure with terminal and non-terminal nodes. Based on the language semantics, we classified for a relevant (according to language composition) subset of AST nodes of SDF and Stratego, all terminal and non-terminal nodes and specified how they can be composed. For example, rules in Stratego documents represent a non-terminal node and therefore an existing rule-set can be extended with new rules. In our example (cf. Fig. 6), the two constraint-error rules of feature HTML extend the rule-set of feature Base. Table 2 summarizes the entire classification of SDF and Stratego syntactic elements into FeatureHouse's terminal and non-terminal nodes.

| | terminal nodes | non-terminal nodes |
|---|---|---|
| SDF | module name, module declaration, production, priority, restriction | disambiguations, export declaration, hidden declaration, import declaration, module, productions |
| Stratego | definition, module name, overlay, rule definition, signature declaration (sorts, constructors), strategy definition (ID) | declaration (rules, strategies, signature, signatures, overlays), module, strategy definition (external) |

**Table 2: Classification of SDF and Stratego syntactic elements into terminal and non-terminal**

Feature Base

```
1  module check
2  imports include/MoBL lookup type rename desugar mobl pp /* ... */
3
4  rules
5   find-duplicate :
6     [el|k] -> <find-duplicate(|el)> k
7   constraint-warning :
8     Module(qid,_) -> (qid,$[Module name does not match file path.])
9     where not(<eq>($[[<qid-to-path>qid].mobl],<CompilingFilename>))
10  /* ... */
```

Feature HTML

```
1  module check
2
3  rules
4   constraint-error :
5     Html(tag,_,_,closeTag) -> (closeTag,$[Wrong closing tag])
6     where not(<eq> (tag,closeTag))
7   constraint-error :
8     NamedHtml(_,tag,_,_,closeTag) -> (closeTag,$[Wrong closing tag])
9     where not(<eq> (tag,closeTag))
10  /* ... */
```

**Figure 6: Excerpts of two language feature units of Mobl; feature HTML extends feature Base**

Note that, of course, not all combinations of language features may be valid (e.g., composing two valid grammars may result in an invalid grammar). If not ruled out by a corresponding feature model, invalid combinations of language feature can be detected using safe-composition techniques [2, 18, 42]. This issue is outside the scope of this paper.

*Dimensions of Variability.* In Figure 7, we illustrate the interplay between the tools for language specification and tool generation, and for the development and composition of language features. A key point is that we base our approach on declarative rule-based specifications that will allow us to develop and evolve language families easier than implementing language tools using general-purpose languages such as Java. For example, it is easier to extend a given type system by adding new typing rules and refining existing ones than by extending and modifying the corresponding Java implementation.

The two dimensions of variability are quite explicit in Figure 7. From top to bottom, we have different variants of a language specification that provide different sets of language features. From left to right, we have the choice of generating different tools from parsers

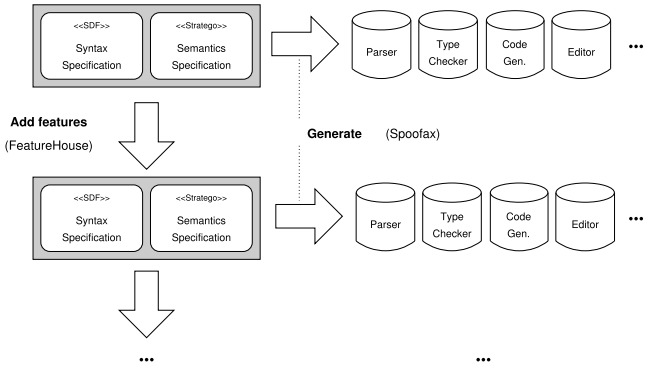


**Figure 7: Using Spoofax to generate language tools, and FeatureHouse and FeatureIDE to compose language specifications**

to editors—the tool features. For example, if we want to generate a parser for our expression language with support for booleans only, we just invoke Spoofax on the SDF grammar that contains only the grammar rules for booleans. If we want additionally a type checker, we invoke Spoofax also on the Stratego typing rules to generate a corresponding type checker. If we want support for natural numbers, we include the corresponding grammar and typing rules using FeatureHouse (i.e., top-down, by means of superimposition) and invoke Spoofax as done before. Following the earlier composition equation, all three examples can be expressed as compositions of specification documents and refinements thereof:

Bool#Syntax

Bool#Syntax • Bool#Typing

Bool#Syntax • Nat#Syntax • Bool#Typing • Nat#Typing

Depending on the document type, the composition operator may differ. Some compositions may involve the extension of existing documents (e.g., adding new syntax rules), others may connect existing components and tools (e.g., invoking a type checker on the syntax tree produced by a parser).

# 4. THE MOBL CASE STUDY

To demonstrate the practicality of our approach, to gain experience with language families, and to reveal open issues, we conducted an initial case study on the basis of a real software language called Mobl. In this section, we introduce Mobl, we describe how we refactored it into language feature units, we explain how we derived different language and tool variants, and we discuss the merits of language families based on the resulting Mobl language family.

*The Mobl Language.* Mobl is a free and open-source language for programming mobile Web applications [23]. It integrates all aspects of a mobile Web application into a single language, including data modeling, user interfaces, application logic, styling, and Web services. Mobl programs are translated to 100% client-side HTML5-based applications, there is no dependency on a specific server-side technology, and the applications are principally off-line capable. Mobl is statically typed and employs type inference to simplify coding. Furthermore, it is deployed with a powerful editor that offers syntax highlighting, content completion, and as-you-type error detection. Mobl has been used to develop a number of real-world applications (e.g., e-learning, podcatching, and time management).[3]

[3] `http://docs.mobl-lang.org/showcase`

Mobl has been developed with Spoofax. That is, its syntax and semantics as well as editor services and so on are defined by means of SDF and Stratego. The corresponding tools are generated with Spoofax. Hence, it is a perfect candidate for our case study.

*Feature Decomposition.* Overall, the Mobl sources consist of 17 SDF documents (912 non-blank lines) that specify Mobl's syntax, 38 Stratego documents (5270 non-blank lines) that define its semantics, and 8 further documents (85 non-blank lines) that define the editor's appearance.

For the purpose of the case study and based on domain knowledge, we have selected 4 features along which we decomposed the specification documents of Mobl: HTML, Service, Async, and OrderBy. As a result, we have a language feature unit for the base Mobl language and 4 additional language feature units that can be added in different combinations (HTML is mandatory).

The process of feature decomposition was as follows. For each feature, we selected one or more rules that obviously belong to the feature (e.g., rule `"async" "" Statement* "" -> Statement cons("Async")` for feature Async). Then, we removed the rules and looked at other specification documents for dependencies and references to them. For each rule we found, we decided whether it belongs to the feature in question (in this case, we removed it as well) or to another feature. For example, for feature Async, we found, among others, references to type checking and pretty printing, which represents a dependency between a language and a tool feature. Beside the syntactic search process, we manually inspected all remaining rules of Mobl to catch also rules that belong to the feature in question, but that are without references to other rules of the feature. Then, we moved all removed rules to a distinct language feature unit. During the decomposition process, we applied minor refactorings, such as splitting of long grammar rules into several shorter ones, to removed and remaining rules to ensure that they can be composed seamlessly. Overall, we had to apply refactorings to the Mobl specification infrequently. We did not observe any interactions between the identified features, such as the change of behavior of a feature in the presence of another one. While feature interactions may occur in larger languages, it is possible to encapsulate them using derivatives [29].

During the decomposition process, we observed that language features crosscut multiple specification documents. In Figure 8, we show the crosscutting nature of Mobl's language feature units. Our observation confirms the necessity of expressive composition mechanisms such as superimposition.

*Deriving Mobl Variants.* Mobl's feature model is simple. Except the basic language specification and the mandatory feature HTML, all features are optional and independent. This results in 8 valid language variants. A language variant is derived by selecting the desired features and passing them to FeatureHouse. On the basis of the feature selection, it generates the corresponding specification documents (in SDF, Stratego, and so on). Based on the generated specification documents, we can generate selectively corresponding language tools such as a parser using Spoofax. We identified, 3 tool features which give rise to 3 tool sets.

In Table 3, we provide information on the individual Mobl variants we generated and, implicitly, the degree of reuse among the language-family members. Using our approach, it is simple to extend the language. One simply adds a new language feature unit and refines the existing specification documents non-invasively—the same holds for adding new tool features.

For illustration, we show in Figure 9 a Mobl variant in action. This variant does not support feature Async, hence the generated

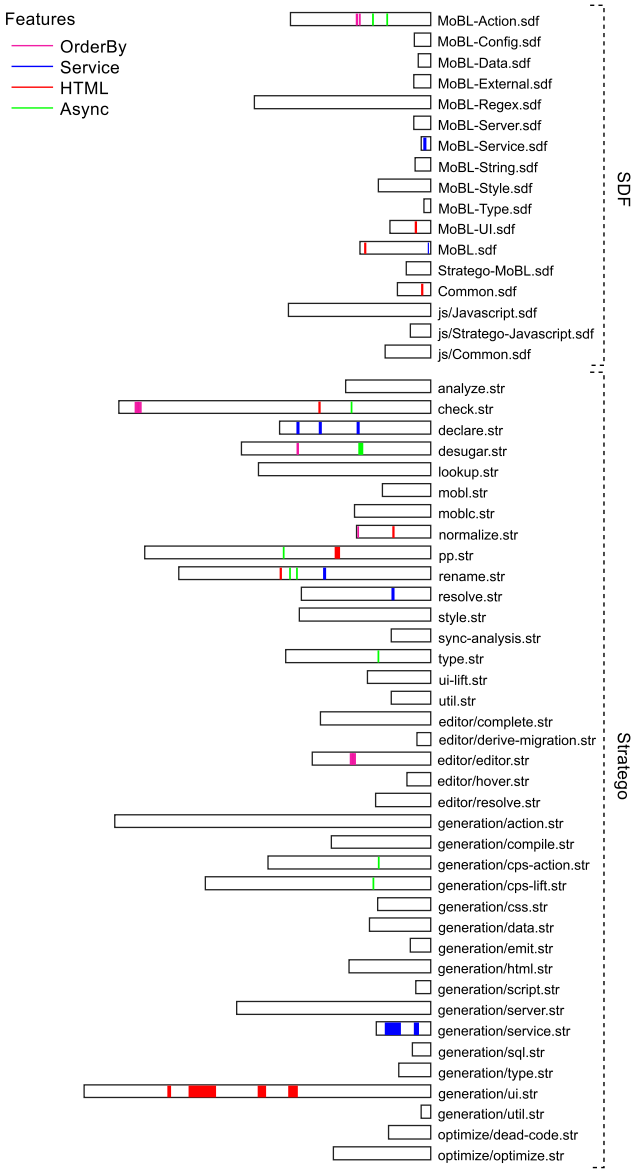


**Figure 8: Crosscutting nature of language features in Mobl**

```
58 // UI
59 screen root() {
60   header("Twitter trends")
61   var trends = async(Twitter.trends())
62   whenLoaded(trends) {
63     group {
64       list(topic in trends) {
65         item(onclick={ search(topic.name); }) {
66           label(topic.name)
67         }
68       }
69     }
70   }
71 }
```

**Figure 9: A Mobl variant in action: This variant does not provide feature Async, thus it does not understand keyword `async` and reports an error**

Mobl editor reports an error because keyword `async` is not known. This error and the numbers of Table 3 show that feature Async has really been removed from the corresponding Mobl variant.

| Language features | | | | | Tool features | | | LOC | Binary |
|---|---|---|---|---|---|---|---|---|---|
| Base | HTML | Async | Service | OrderBy | SDF | Stratego | Edit | | |
| ✓ | ✓ | | | | | | | 860 | 17 836 379 |
| ✓ | ✓ | ✓ | | | ✓ | | | 872 | 17 848 996 |
| ✓ | ✓ | | ✓ | | ✓ | | | 872 | 17 854 504 |
| ✓ | ✓ | | | ✓ | ✓ | | | 864 | 17 841 281 |
| ✓ | ✓ | ✓ | ✓ | | ✓ | | | 874 | 17 861 796 |
| ✓ | ✓ | ✓ | | ✓ | ✓ | | | 866 | 17 853 857 |
| ✓ | ✓ | | ✓ | ✓ | ✓ | | | 876 | 17 859 171 |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | 878 | 17 870 584 |
| ✓ | ✓ | | | | ✓ | ✓ | | 5 837 | 24 490 287 |
| ✓ | ✓ | ✓ | | | ✓ | ✓ | | 5 895 | 24 520 002 |
| ✓ | ✓ | | ✓ | | ✓ | ✓ | | 5 911 | 24 548 812 |
| ✓ | ✓ | | | ✓ | ✓ | ✓ | | 5 885 | 24 533 243 |
| ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | | 5 969 | 24 577 114 |
| ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | | 5 943 | 24 550 214 |
| ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | | 5 959 | 24 593 531 |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | 6 017 | 24 626 201 |
| ✓ | ✓ | | | | ✓ | ✓ | ✓ | 5 922 | 24 474 441 |
| ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | 5 980 | 24 521 693 |
| ✓ | ✓ | | ✓ | | ✓ | ✓ | ✓ | 5 996 | 24 551 352 |
| ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | 5 950 | 24 545 311 |
| ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | 6 054 | 24 600 655 |
| ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | 6 028 | 24 597 107 |
| ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | 6 044 | 24 616 153 |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 6 102 | 24 657 767 |

**Table 3: Overview of the generated Mobl variants**

*Discussion and Perspectives.* Our case study demonstrates the practicality of our approach. We decomposed a real-world language into language feature units that can be reused in different language variants, which are automatically generated based on a user's feature selection. The language features are specified declaratively and composed via superimposition. This way, we are able to add new features easier than based on low-level implementations in general-purpose languages. The use of superimposition as composition operator allows us to encapsulate even crosscutting features, as it was the case for all features of Mobl we considered (cf. Fig. 8). Crosscutting appears to be the rule in software-language engineering—induced by multiple dimensions of variability (cf. Sec. 2).

During our experiments with Spoofax, we recognized that, actually, it uses a combination of generation *and* interpretation. Parts of the overall tool chain are generated, but some parts of the Stratego rules are interpreted by the generated tool chain at runtime. Unfortunately, we were not able to precisely quantify the amount of generation and interpretation and the implications for the binary size of the generated code. Still, our results are promising in that they demonstrate the principle benefits of language families.

Decomposing a language along the features it provides allows us to generate *tailored* language variants. The variants of Mobl do really provide only the features that have been selected during variant generation. Hence, tailoring has an effect on the complexity of the language and the size of the resulting language tools, as shown in Table 3. Of course, in our case study, the size of the language features, compared to the overall size of the language, is rather small—naturally, the influence of the tools features is higher in this respect. Anyway, the extraction of more substantial language features will likely have a larger effect on downsizing.

Once accepting the feature idea, language evolution boils down to feature inclusion. But, as with any other system, not all evolution tasks can be anticipated and may require refactorings of existing features. This issue is subject of ongoing research on product-line evolution [41].

## 5. RELATED WORK

*Stepwise and Feature-Oriented Language Development.* The idea of decomposing a language specification along its features has been first proposed by Batory et al. [10]. They developed and evolved languages by adding language features incrementally in the form of composable units. They applied their approach to grammars as well as to language-processing tools written in Java. In this spirit, Apel et al. developed a formal language in a feature-oriented way, including syntax, typing, and evaluation rules [2, 3, 7]. However, they did not develop corresponding tools and they did not apply their approach to a real language. Batory and Börger developed a model of the Java language incrementally (feature by feature), but the work was only at the conceptual level, without implementation or tool support. In a related attempt, Batory et al. applied the idea of decomposing language specifications to theorems about properties of a particular language. Based on a user's selection, the corresponding correctness proof is generated [7, 19].

Cazzola proposed the framework NEVERLANG, which provides means to encapsulate language and tool features and to generate specific language variants using feature composition [14]. Language and tool features are encoded directly in a programming language including the generator for language variants. By contrast, we use the language workbench Spoofax [26], which uses generic language specifications in the form of SDF/Stratego documents, which can be composed using our software composition tool FeatureHouse [4]. Furthermore, our approach integrates feature models to rule out invalid language variants.

*Extensible Compiler Frameworks.* A number of extensible compiler frameworks have been developed, for example, Polyglot [31], abc [6], and JastAdd [20]. The idea is to provide extension points in the framework to extend and refine a given language. Typically, compiler frameworks are written in general-purpose languages such as Java. They are inherently complex, hard to understand, and difficult to adapt to unanticipated extensions [30, 38]. Our approach aims at the development of language families based on declarative languages specifications and feature decompositions thereof. The underlying assumption is that declarative language descriptions are easier to understand and extend—an assumption that drives the development of parsers for many years (in terms of parser generation based on grammars).

*Language Composition.* A number of approaches aim at composition of languages that serve different purposes. The main focus was on embedding domain-specific languages into general-purpose languages or combining multiple domain-specific languages. Furthermore, the approaches concentrated on grammar combination and pure syntax embedding, for example [13, 21, 24, 27]. On the one hand, our approach goes beyond these approaches by considering all kinds of declarative language specifications and the subsequent generation of language tools. On the other hand, we do not consider the integration of completely independent languages (e.g., integrating SQL into Java), but families of related languages that differ in some language features (e.g., SQL with and without spatial queries), which poses less challenges.

*Extensible Type Systems.* Recently, researchers noted that type systems either grew too big or were too less restrictive. An early approach is used in the Glasgow Haskell compiler, which can be configured to support additional syntax and type rules via command-line flags. However, the support for the syntax and typing rules, although disabled, remains in the compiler and may degrade performance or introduce errors. A recent idea is to develop extensible type systems that form a framework for basic types and type checking, but that supports also plugging in new types and typing rules [22, 32]. In contrast to our approach, types are plugged in by the user on demand. There is no explicit family of type systems that is developed and evolved systematically, in sync with declarative specifications of the language's syntax and semantics.

*Intentional Programming.* Simonyi et al. proposed intentional programming, an approach to express the programmer's intention explicitly in the language [37]. It is based on powerful editors that allow developers to create easily domain-specific abstractions and to use them right ahead in their programs (see the JetBrains Meta Programming System for a recent implementation[4]). In fact, user-defined domain-specific abstractions can be composed in different combination, thus representing a language family. Our approach is similar in spirit but pursues a generative approach. There is no single powerful, programmable editor in which one has to load abstractions, but a set of language feature units that are composed by a generator based on a feature selection.

## 6. CONCLUSION

We have presented an approach to feature-oriented language engineering. The idea is to decompose a language along the features it provides, thus giving rise to a language family. We have assembled a tool chain for the development of language families, and we used it to refactor the given language Mobl into a language family. As a result, we can derive 24 variants of Mobl simply by specifying the language and tool features it shall provide. This has an immediate effect on the complexity and size of the language variants. Our approach is able to handle crosscutting features and facilitates an incremental style of language development.

In further work, we shall gather more experience with language families, for instance, by applying the approach to a subset of SQL and its versions. Furthermore, it is interesting to explore the influence of unused language features and feature interactions on performance, memory consumption, and defects of the programs in question.

## Acknowledgments

We are grateful to Don Batory and Ben Delaware for valuable comments on earlier drafts of this paper. Apel's work is supported by the German Research Foundation (DFG – AP 206/2 and AP 206/4).

## 7. REFERENCES

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *JOT*, 8(5):49–84, 2009.

[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *ASE*, 17(3):251–300, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *GPCE*, pages 101–112. ACM, 2008.

[4] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *ICSE*, pages 221–231. IEEE, 2009.

[4] `http://www.jetbrains.com/mps/`

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *SCP*, 75(11):1022–1047, 2010.

[6] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An Extensible AspectJ Compiler. In *AOSD*, pages 87–98. ACM, 2005.

[7] D. Batory and E. Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. *JUCS*, 14(12):2059–2082, 2008.

[8] D. Batory, P. Höfner, and J. Kim. Feature Interactions, Products, and Composition. In *GPCE*, pages 13–22. ACM, 2011.

[9] D. Batory, J. Liu, and J. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *ESEC/FSE*, pages 48–57. ACM, 2003.

[10] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[11] S. Boxleitner, S. Apel, and C. Kästner. Language-Independent Quantification and Weaving for Feature Composition. In *SoftComp*, pages 45–54. Springer, 2009.

[12] M. Bravenboer, K. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *SCP*, 72(1–2):52–70, 2008.

[13] M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *OOPSLA*, pages 365–383. ACM, 2004.

[14] W. Cazzola. Domain-Specific Languages in Few Steps - The Neverlang Approach. In *SoftComp*, pages 162–177. Springer, 2012.

[15] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[16] W. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Proc. of REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178. Springer, 1991.

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *FSE*, pages 243–252. ACM, 2009.

[19] B. Delaware, W. Cook, and D. Batory. Product Lines of Theorems. In *OOPSLA*, pages 595–608. ACM, 2011.

[20] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA*, pages 1–18. ACM, 2007.

[21] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *OOPSLA*, pages 391–406. ACM, 2011.

[22] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, Pluggable Types for a Dynamic Language. *Computer Languages, Systems and Structures*, 35(1):48–62, 2009.

[23] Z. Hemel and E. Visser. Declaratively Programming the Mobile Web with Mobl. In *OOPSLA*, pages 695–712. ACM, 2011.

[24] P. Hudak. Modular Domain Specific Languages and Tools. In *ICSR*, pages 134–142. IEEE, 1998.

[25] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *SPLC*, pages 181–190. SEI, 2009.

[26] L. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.

[27] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *STTT*, 12(5):353–372, 2010.

[28] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-Use. In *ECOOP*, pages 91–113. Springer, 1998.

[29] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *ICSE*, pages 112–121. ACM, 2006.

[30] S. Moser and O. Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, 1996.

[31] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An Extensible Compiler Framework for Java. In *CC*, volume 2622, pages 138–152. Springer, 2003.

[32] M. Papi, M. Ali, T. Correa, Jr., J. Perkins, and M. Ernst. Practical Pluggable Types for Java. In *ISSTA*, pages 201–212. ACM, 2008.

[33] D. Parnas. On the Design and Development of Program Families. *TSE*, 2(1):1–9, 1976.

[34] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[35] J. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 13–23. MIT Press, 1994.

[36] M. Rosenmüller, C. Kästner, N. Siegmund, S. Sunkle, S. Apel, T. Leich, and G. Saake. SQL à la Carte – Toward Tailor-made Data Management. pages 117–136, 2009.

[37] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *OOPSLA*, pages 451–464. ACM, 2006.

[38] S. Sparks, K. Benner, and C. Faris. Managing Object-Oriented Framework Reuse. *IEEE Computer*, 29(9):52–61, 1996.

[39] Guy Steele Jr. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[40] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, pages 2–11. IEEE, 2005.

[41] M. Svahnberg and J. Bosch. Evolution in Software Product Lines: Two Cases. *Journal of Software Maintenance*, 11(6):391–422, 1999.

[42] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.

[43] M. Torgersen. The Expression Problem Revisited. In *ECOOP*, pages 123–143. Springer, 2004.

[44] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *CC*, volume 2027, pages 365–370. Springer, 2001.