Neverlang and FeatureIDE Just Married

Integrated Language Product Line Development Environment

Luca Favalli
Università degli Studi di Milano
favalli@di.unimi.it

Thomas Kühn
Karlsruhe Institute of Technology
thomas.kuehn@kit.edu

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

ABSTRACT

Language development is inherently complex. With the support of a suitable language development environment most computer scientists could develop their own domain-specific language (DSL) with relative ease. Yet, when the DSL is the result of a configuration over a language product line (LPL)—a special software product line (SPL) of compilers/interpreters and corresponding IDE services—they fail to provide adequate support. An environment for LPL engineering should facilitate the underlying process involving three distinct roles: a language engineer developing the LPL, a language deployer configuring a language product, and a language user using the language product. Neither IDEs nor SPLE environments can cater all three roles and fully support the LPL engineering process with distributed, incremental development, configuration, and deployment of language variants. In this paper, we present an LPL engineering process for the distributed, incremental development of LPLs and an *integrated language product line development environment* supporting this process, catering the three roles, and ensuring the consistency among all artifacts of the LPL: language components implementing a language feature, the feature model, language configurations and the resulting language products. To create such an environment, we married the Neverlang language workbench and AiDE its LPL engineering environment with the FeatureIDE SPL engineering environment. While Neverlang supports the development of LPLs and deployment of language products, AiDE generates the feature model for the LPL under development, whereas FeatureIDE handles the feature configuration. We illustrate the applicability of the LPL engineering process and the suitability of our development environment for the three roles by showcasing its application for teaching programming with a growable language. In there, an LPL for Javascript was developed/refactored, 15 increasingly complex language products were configured/updated and finally deployed.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software product lines**.

KEYWORDS

Domain Specific Languages, Language Product Lines, Neverlang

SPLC '20, October 19–23, 2020, Montréal, QC, Canada

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414961

ACM Reference Format:
Luca Favalli, Thomas Kühn, and Walter Cazzola. 2020. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, Montréal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3382025.3414961

1 INTRODUCTION

Computer scientists generally agree that the development of a programming language is inherently complex. With the advent of language development environments [14]—dedicated *integrated development environments* (IDE) for software language engineering—most of them could now develop their own *domain-specific language* (DSL) or programming language with relative ease. Consequently, some computer scientists combined different language variants to families of both DSLs and programming languages, e.g., [15, 23, 24, 26, 31]. However, state-of-the-art language development environments reach the limits of their capabilities when tasked with incrementally developing reusable language components, configuring language variants by choosing and picking individual language features for a language variant,[1] or deploying IDEs for multiple different language variants. To address their limitations, researchers recently started investigating *language product lines* (LPL), i.e., a special software product line (SPL) of compilers/interpreters and embedded IDE services, as well as dedicated language development environments for LPL engineering, e.g., [21, 24, 32]. However, in contrast to these approaches, we argue that such an environment, should support an engineering process enacted by three distinct roles: (1) a language engineer developing the LPL, (2) a language deployer configuring and deploying language variants and (3) a language user using one (or more) language products in an IDE. Moreover, we emphasize that an effective LPL IDE must support incremental development and configuration as well as rapid deployment and testing of language variants. Currently, neither language development nor SPL engineering environments nor LPL development environments are able to support the LPL engineering process catering all three roles in one tightly integrated development environment.

In this paper, we outline the LPL engineering process and present an *integrated language product line development environment* for bottom-up LPLs able to support this process catering all roles in a single IDE. For brevity, we assume the bottom-up approach is meant whenever we refer to LPLs, LPL development environment and LPL engineering. By extension, we support distributed and incremental development, configuration, and deployment by maintaining the

[1] Following [30], *language features* are either language constructs, e.g., *for loop*, or language concepts (without concrete syntax), e.g., *scope* and *recursion*.

consistency between language components implementing a language feature, the feature model (FM), the language configurations and the deployed language products. Additionally, we allow deployers to cope with the *domino effect* when restricting a language, i.e., where strongly connected dependencies between language features prevent language specialization [20]. To create this LPL development environment, we married the Neverlang language workbench [30] and AiDE its LPL development environment [20] with the FeatureIDE [25] SPL engineering environment.Neverlang permits the development of LPLs and deployment of language products within the Eclipse IDE, AiDE generates the feature model for the LPL under development whereas FeatureIDE provisions the feature configuration. Moreover, we extended the *Feature Configuration Editor* to allow for declaring a language configuration and dealing with the *domino effect*. We demonstrate the applicability of both the process and our environment for LPL engineering by showcasing their application for gradually teaching JavaScript [5] by employing the LPL of JavaScript-based languages [30]—neverlang.JS. We incrementally recreated 15 JavaScript language variants of increasing complexity following the teaching case study whereas each variant was individually deployed and tested within equivalent Eclipse IDEs. Besides, we showcase how the deployer copes with the *domino effect* and illustrate the consistency preservation mechanisms in play when refactoring language components. In conclusion, this marriage brings the state-of-the-art in LPL engineering and SPL engineering together to support distributed, incremental development, configuration, and deployment of language families.

2 BACKGROUND

2.1 SPL Engineering with FeatureIDE

Product variability, product families and product lines are concepts not only applied to product development, but also to software development. SPL engineering strives to ease the creation of software variants by applying product line concepts to software development. An SPL is a family of software products whose commonalities and differences can be described in terms of their features. SPL engineering combines concepts from domain engineering for the design and implementation of software artifacts with concepts from application engineering to create products from selected features [2]. SPLs, along with features and their dependencies are often described in terms of a feature model (FM) [17].

FeatureIDE [25] is a SPL development environment that copes with all aspects of the development of SPLs. It supports the FM construction, the management of software artifacts, the configuration and product derivation. SPL engineering support in FeatureIDE encompasses: (1) the *Feature Model Editor* for the creation, visualization and tracing of FMs, concrete and abstract features, feature dependencies and cross-tree constraints; (2) the *Configuration Editor* for the creation, modification, and validation of feature configurations; and (3) various *Composers* for the derivation of product variants from a given a valid feature configuration. As an example, Figure 1 showcases a FM built with FeatureIDE.

FeatureIDE maintains consistency between the different views during all phases of the development process. For instance, any modification of the FM in the FM Editor is propagated to the Configuration Editor, while any other configuration is marked if it became

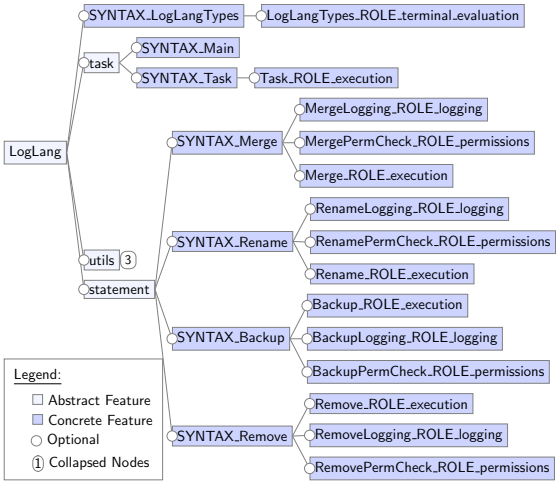

Figure 1: Example feature model for log rotating tools [6].

inconsistent, as a result. The Configuration Editor runs a SAT-solver to check the validity of feature configurations in relation to feature dependencies and cross-tree constraints. Finally, a composer generates the final product variant for a given configuration by selecting and composing the corresponding software artifacts.

2.2 Language Product Lines

The development of families of programming and DSLs has gained popularity among researchers and practitioners, e.g., [15, 22, 23, 26]. Following the ideas from SPLs, an LPL facilitates the process of language development, which can be *customized* by selecting individual *features*: similarly to any other software, a language can be designed for a specific use case or application domain. For instance, several works [10, 29, 32] showed that the variants of state machine languages can be modeled as a 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 [8]) or teaching [5, 12]. 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 [13].

LPLs can be created using either a *top-down* or a *bottom-up* approach [19]. In the former approach, the feature model is created first, then the features are mapped to the language artifacts and finally language variants are developed though a configuration process. In the latter, the language developer creates individual language artifacts, from which a feature model is generated and used to guide the configuration. Henceforth, we will focus on the *bottom-up* approach for LPL engineering.

Figure 1 showcases the FM generated for the family of LogLang variants. LogLang is a simple DSL that describes tasks for a log rotating tool similar to the logrotate Unix utility with a modular Neverlang implementation [6]. Given valid feature configuration the interpreter for a specific LogLang variant is generated. We denote a language variant to be *viable*, if its language recognizes/evaluates the selected language constructs with the expected semantics.

```
1  module Backup {
2    reference syntax {
3      provides { Backup: backup, statement; Cmd: statement; }
4      requires { String; }
5      Backup ⟵ "backup" String String;
6      Cmd    ⟵ Backup;
7      categories  : Keyword = { "backup" };
8      in-buckets  : $1 ⟵ { Files }, $2 ⟵ { Files };
9      out-buckets : $1 ⟶ { Files }, $2 ⟶ { Files };
10   }
11   role(execution) {
12     0 .{
13       String src  = $1 string, dest = $2 string;
14       $$FileOp.backup(src, dest);
15     }.
16   }
17 }
18 slice BackupSlice {
19   concrete syntax from Backup
20   module Backup with role execution
21   module BackupPermCheck with role permissions
22 }
24 language LogLang {
25   slices BackupSlice RemoveSlice RenameSlice
26          MergeSlice Task Main LogLangTypes
27   endemic slices FileOpEndemic PermEndemic
28   roles syntax < terminal-evaluation < permissions : execution
29 }
```

Listing 1: Syntax and semantics for the backup task.

2.3 Neverlang and AiDE in a Nutshell

Neverlang [4, 7, 30] is a framework for *component-based* development of programming languages. Language components, called *slices*, embody language features and are developed as separate units that can be independently compiled, tested, and distributed, enabling developers to share and reuse the same units across different language implementations. The basic development unit is a `module` (Listing 1). A module may contain a `syntax` definition and/or roles. A `role` defines semantic actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique [1]. Syntax definitions and semantic roles are tied together using **`slices`**.

Listing 1 illustrates the implementation of the `Backup` feature of the LogLang LPL. Here the module `Backup` declares a reference syntax for the backup task (lines 2-10). The reference syntax of a module also declares information for basic IDE services, such as syntax highlighting (line 7) and code-completion (lines 8-9). Semantic actions are attached to nonterminals of the productions (lines 12-15) by referring to their position in the grammar: numbering starts with 0 from the top left to the bottom right.[2] Thus, the `Backup` non-terminal on line 5 is referred to as 0 and the two `String` non-terminals on the right-hand side of the production as 1 and 2, respectively. Each `role` may declare different semantic actions for the same productions and will represent a different compilation phase. In contrast, the `BackupSlice` (lines 18-22) declares that it will combine *this* syntax (as the `concrete syntax`) in our language (line 19), with the semantics of two separate semantic actions from two different modules (lines 20-21). Finally, the `language` descriptor (lines 24-29) indicates which **`slices`** should be composed to generate the language interpreter (lines 25-26). Therefore, composition in Neverlang is twofold: (1) between modules, which yields slices, and (2) between slices, which yields a language implementation. The grammars are merged to generate the complete language parser. Semantic actions are performed with respect to the parse tree of the input program; roles are executed in the sequence and traversal specified in the **`roles`** clause (line 28) of the `language` descriptor, e.g., `permission` is executed after `parsing` and `terminal-evaluation`. Besides that, it can declare **`endemic slices`**, instances shared across multiple compilation phases (line 27).[3]

Neverlang supports LPL engineering thanks to AiDE [19, 20]. AiDE is a variability management tool tailored for the development of LPLs. It uses information provided by Neverlang modules (lines 3-4 of Listing 1) and implements an extension to the method presented in [20, 31, 32] to automatically synthesize the FM of a given language family [30]. Through its graphical user interface, the user can explore the FM, choose features, create a language variant, and test it. Because FM of LPLs tend to be large [19], AiDE initially shows the first level of the tree enabling its expansion. Moreover, AiDE tracks all unresolved dependencies—i.e., all open nonterminals in the current configuration—and provides the user with mechanisms, such as renaming, to bind them to other nonterminals already in the configuration, while the user configures a language variant. This simplifies the resolution of dependencies during the feature selection.

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. The AiDE current configuration can be deployed at any time, resulting in a Java archive containing the developed language and all dependencies. By changing the name of the deployed language, users can effectively develop a library of variants of the same language family. In sum, AiDE guides users towards the generation of consistent language variants by supporting multiple dependency resolution strategies and continuous generation of the language compiler/interpreter. Notably though, AiDE lacks automatic feature selection and suggestion as well as an integrated editor for language variants supporting syntax highlighting, code-completion, and debugging.

3 TOWARDS AN INTEGRATED LPL DEVELOPMENT ENVIRONMENT

First and foremost, an LPL development environment must support the LPL engineering process, and thus, individually support the three roles: language developer, language deployer, language users. Moreover, it must support several distinct views on the LPL under development and provide basic IDE services, e.g., syntax highlighting, error marking, code-completion. Thus, in this section, we outline the LPL engineering process, the roles involved, the different views used by each role, the underlying artifacts, and consistency relations between the artifacts.

[2] Neverlang also permits to label a production and refer nonterminals via an offset from such a label, e.g., `$BKP[0]` is the head of the `BKP` production.

[3] Please see [30] for further details.

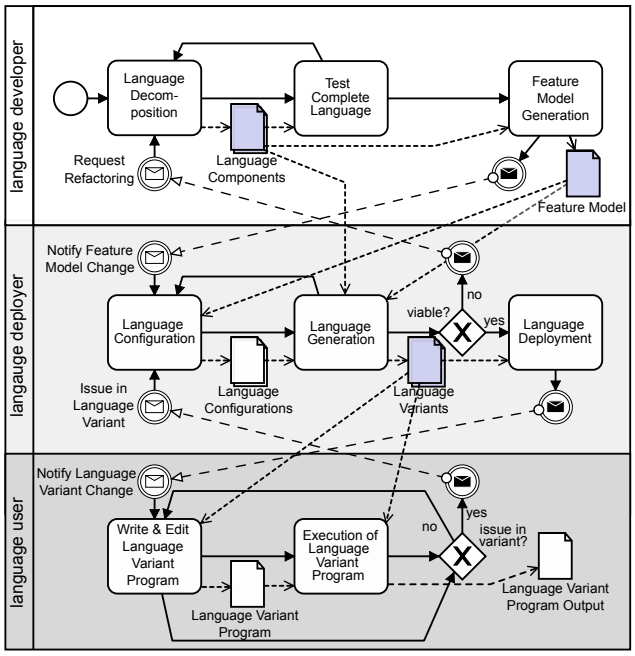

Figure 2: BPMN describing the LPL engineering process.

3.1 LPL Engineering Process

Although the general SPL engineering process presented in [27] and illustrated in [25] also applies for LPLs, we argue that it is still too coarse-grained to disclose the relevant users and views for an LPL development environment. Similarly, the LPL development process, described in [19], is not detailed enough as it neglects the language user. As a result, Figure 2 illustrates the process of LPL engineering as a *business process model* [9]. It shows the three distinct roles, i.e., language developer, language deployer, and language user, as different swim lanes (rows) of the process and their corresponding task. Moreover, it shows the artifacts created or refined by each task. Distributing the engineering process over several areas of responsibility allows for the concurrent development of LPLs while minimizing conflicts. Here, only the language components, the FM and the language variants are highlighted, because they are shared between several tasks of different roles, therefore they must be kept consistent between the views of the three roles.

3.2 Language Engineering

The *language developer* starts the LPL engineering process by decomposing a language into individual language components and iteratively testing a complete language variant (typically) against a monolithic reference implementation. Following the *bottom-up* approach [19], the FM can be regenerated from the set of language components. Finally, the *language developer* commits the LPL. To support these tasks, an LPL development environment must provide an editor for implementing language components, facilities to run and test a complete language variant, and finally an editor to review and analyze the generated FM for the LPL under construction. Naturally, the FM generation should be automated, e.g., by triggering the (re)generation of the FM each time a language component is added, modified, or deleted.

3.3 Language Configuration and Deployment

In contrast, the *language deployer* is notified about each FM change when checking out the LPL, to create or revise language configurations of the different language variants. The deployer incrementally generates language variants and tests their viability, i.e., whether it includes the desired language constructs and concepts. In case a language variant that is not yet viable is generated, e.g., due to a too coarse-grained language component, the language deployer requests a refactoring of language components by the language developer. Otherwise, the deployer (re)deploys and commits the language variant, which makes it available to its users.Consequently, the deployer requires a different set of views and services from an LPL development environment. They require a language configuration editor to easily select individual language features and create a viable language variant. However, as already outlined in [20], this language configuration editor should not only automatically select implied language features and ease the configuration of compilation phases (order of semantic actions), but also cope with the *domino effect*, i.e., specifically leaving dependencies unfulfilled enabling their resolution on a syntactical level or by selecting alternative language components. Besides that, the LPL development environment must provide means to generate a language variant and test its viability before deployment. Again, the generation of a language variant lends itself to automation, as it can be triggered whenever a language configuration is created or modified.

3.4 Usage of Language Variants

Once a language variant is deployed and committed, a *language user* can checkout the latest version to create, edit, and execute the corresponding programs. In case of issues in the language variant either on the syntactic or semantic level, they report them to the language deployer, who, in turn, could adapt the language configuration. For the language user, the LPL development environment should be indistinguishable from an IDE, and provide the expected basic IDE services. Simply put, a language variant's user needs a syntax highlighting editor, preferably with code-completion, as well as means to execute and/or debug the program written in the language variant. From their perspective, each language variant is its own isolated programming language supported by the IDE.

In conclusion, each role requires very different views and services provided by the LPL development environment. Granted, it is challenging to implement these views in one development environment. Yet, we argue that this enables distributed, incremental development of LPLs with tight feedback loops and rapid deployment, whereas the LPL development environment maintains the consistency between the shared artifacts, i.e., the language components, the FM, and language variants. The engineering process can also address conflicts in the requirements: requests from different language user can be balanced by configuring additional language variants while feature conflicts are translated into FM constraints by the environment.

Algorithm 1: ExpandFeatureModel (FM: Feature Model)

```
begin
    P := {p | p is a node in FM};
    for p ∈ P do
        S := {f | f ∈ syntactic_features(p)};
        for f ∈ S do
            generate concrete syntactic node n for f;
            children(n) := ∅;
            R := {f' | f' is a semantic feature compatible with f};
            for f' ∈ R do
                generate concrete semantic node n' for f';
                children(n) := children(n) ∪ {n'};
            end
            children(p) := children(p) ∪ {n};
        end
    end
    return FM;
end
```

4 MARRY NEVERLANG AND FEATUREIDE

Instead of implementing an LPL development environment from scratch, we opted to marry two established development environments, Neverlang and AiDE for LPL development and FeatureIDE for feature-oriented SPL development. This section outlines how we combined the two into a powerful LPL development environment.

4.1 Integrating AiDE into FeatureIDE

FeatureIDE and AiDE are standalone environments for the development of product lines. Our contribution is an integrated LPL development environment born by marrying the two. We refactored AiDE employing the FeatureIDE core to implement the *layered language feature model* as an extension of the default FeatureIDE FM class and a set of additional abstractions to represent the different syntactic and semantic features of an LPL, as well as cross-tree constraints. The IDE was implemented as an Eclipse plugin and provides several extension to FeatureIDE and native Eclipse: (1) the AiDE project nature for LPL projects; (2) a Neverlang incremental builder for said nature; (3) the AiDE composer for the creation of language artifacts from configuration files; (4) wizards for the creation of new LPL projects and language variants extending the *New Feature Project Wizard* and the *New Configuration Wizard* respectively; (5) the *Neverlang Configuration Editor* extending the FeatureIDE *Configuration Editor*. In addition, we utilize the *Neverlang Editor*, introduced in [21], for the development and usage of LPLs featuring basic IDE features. Yet, we extended its implementation to allow dynamic reloading of language variants.

4.2 AiDE

For generating a FM from language components, we extended the algorithm presented in [20]. The novel AiDE algorithm expands the FM one level deeper by distinguishing between syntactic and semantic language features. Algorithm 1 accepts the FM generated with the original AiDE algorithm from [20] as an input. First, only abstract features are present, then our extension creates the corresponding syntactic features as leaves. Finally, all semantic actions, attached to a syntactic feature are added as their leaves. This enables a more fine-grained customization of languages, as it enlarges the variant space.

Figure 1 shows the FM for LogLang generated by AiDE. It contains abstract features generated from the tags in defined **`modules`** (Listing 1 lines 3-4) and two layers of concrete features. The first holds the syntactic features of the LPL, whereas the second contains the corresponding compatible semantic features. To seamlessly support the LPL engineering process, the AiDE FM generation algorithm has been integrated with the Eclipse build process. Whenever a `module` is added, deleted or changed in the workspace, the incremental Neverlang compiler compiles the most recent version of the file. The pool of available language features and the FM are updated by the AiDE runtime environment.

4.3 Neverlang Language Configuration Editor

Language features can be combined into a language configuration using the Neverlang `language` construct. AiDE supports the automatic generation of `language` and `slice` files through the *Neverlang Language Configuration Editor* (Figure 3, language deployer layer). It extends the default FeatureIDE *Configuration Editor* to create all the language variants. The variability space of an LPL can be further expanded wrt. its SPL equivalent by allowing for language restrictions that would normally lead to invalid configurations. Due to the *domino effect*, removing a language feature requires all features dependent on it to be removed as well. In case of language grammars, this is often due to open non-terminals. Neverlang permits `renaming` to stop the domino effect, i.e., an open non-terminal can be renamed to a provided non-terminal to fill the gap and obtain a viable language variant although the feature configuration is invalid. The *Neverlang Language Configuration Editor* adds a *Renames* tab (Figure 3, language deployer layer, right side) to incorporate this functionality into FeatureIDE. In addition, the compilation phases for the interpreter/compiler can be specified in the *Roles* tab by defining the succession and traversal of semantic actions. The source generation process is triggered whenever changes are saved. This will automatically generate all **`slices`** by collecting the selected syntactic and semantic features and compile the corresponding language variant. The generated code for the language variant can be inspected in the *Neverlang source* tab. Since the Neverlang compiler translates `language` files into Java classes, the language variant's interpreter can be immediately tested by running the generated Java class as *Java application*.

4.4 Neverlang Editor

The *Neverlang Editor* (Figure 3, language developer and language user layers) is an LPL-driven editor, introduced in [21]. It collects and integrates IDE services specified in **`modules`** to deploy a tailored editor for language variants.

Since the Neverlang compiler is bootstrapped, the *Neverlang Editor* serves as an environment for both the development of language components and the usage of language variants. Moreover, language variants can be dynamically loaded and the editor's IDE services adapted at runtime. Furthermore, it provides syntax highlighting and code-completion services by cross-referencing the IDE specifications within the language components in language variants, e.g., `categories` hold stylistic information for a grammar fragment, `out-buckets` are fed with text from a terminal or non-terminal symbol and can be retrieved to provide suggestions for

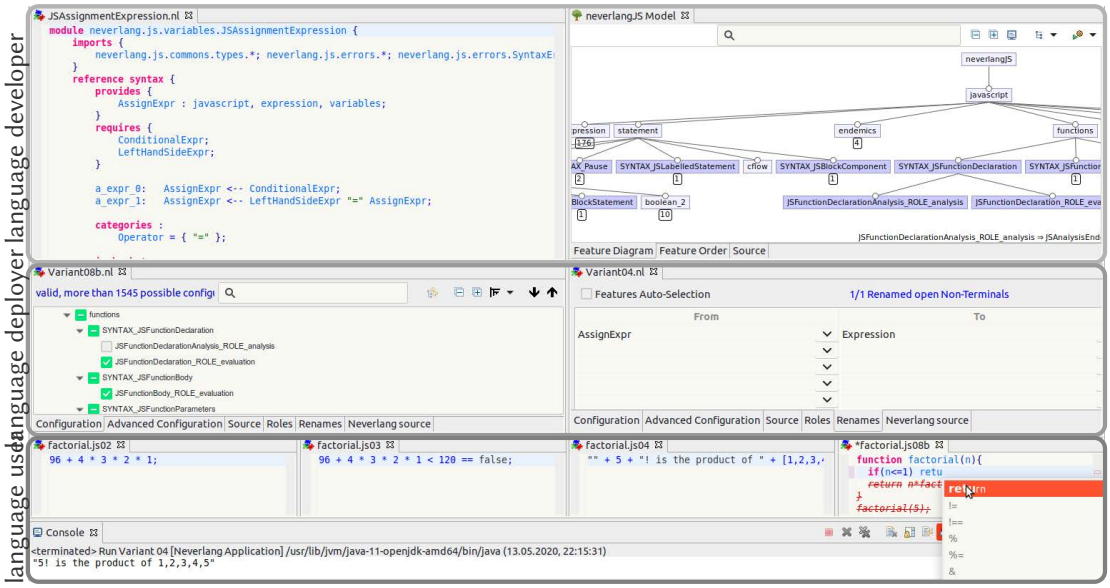

Figure 3: Overview on the Neverlang LPL development environment highlighting the views provided to each role.

code-completions using the `in-buckets` directive (Listing 1 lines 7-9). For further information on IDE specifications in Neverlang and their implementation, we refer the reader to [21]. This work contributes to the *Neverlang Editor* by integrating dynamic reloading of language implementations within the same Eclipse instance to better suit the LPL engineering process and the incremental development of multiple language variants.

4.5 Deployment of a Language Product

FeatureIDE can deploy languages and language families using the AiDE library and Gradle. Upon creation of an LPL project, AiDE optionally generates a `build.gradle` file with a `distribution` task, which generates a Java archive containing the required project binaries and their dependencies. Binaries of language variants can be registered by referring to either binary project folders or `jar` archives and specifying the fully-qualified name of the language class file within the Neverlang `config.json` file.

Once a language is registered, the *Neverlang Editor* can load its basic IDE services. Programs compliant with any registered language variant can be executed inside the Eclipse console (Figure 3, language user layer, bottom part) by producing a *Neverlang run configuration* for that file specifying the desired language variant.

4.6 Consistency Preservation

Consistency preservation between the LPL artifacts is maintained in part by the Neverlang compiler, AiDE, and FeatureIDE. Any change to a Neverlang source file (language components) in the workspace triggers the Neverlang compiler. The compiler is responsible for both the translation of Neverlang into the target language (Java by default) and the synchronization of an *environment* holding all the language features relevant to the FM creation. AiDE issues the regeneration of the FM according to Algorithm 1, when the Neverlang source update forces an update to the environment. The language developer reviews the updated version of the FM inside the FM Editor and decides to either discard or accept the changes by canceling or performing the save operation. Finally, he commits the LPL. After checking out the recent LPL, FeatureIDE notifies the language deployer by applying a warning on any inconsistent configuration wrt. the current FM. The language deployer reviews the language configurations, to solve any inconsistency in the FM, and commits any change. The *Neverlang Language Configuration Editor* automatically reestablishes a consistent language configuration whenever the FM change is not substantial—i.e., no concrete features are renamed or removed. In this way FeatureIDE supports the incremental development of LPLs with little to no side effects.

5 DEMONSTRATION CASE STUDY

This section showcases the applicability of the engineering process we presented in Sect. 3, as well as the suitability of our development environment for the three roles, i.e., language developer, language deployer, and language user, by illustrating its use for gradually teaching programming as featured in [19]. The whole experiment was undertaken in a distributed environment and versioned using `git`.[4] The repository features 3 different authors, each one embodying one of the three roles: *language developer*, *language deployer*, and *language user*; the contribution made by each author can be reviewed by inspecting the commit history and highlights the underlying distributed, incremental LPL engineering process.

5.1 Family of JavaScript-based Languages

For our case study, we employed and refactored the LPL for the family of JavaScript-based languages introduced in [30]—neverlang.JS.

[4]The repository is available at:

`https://cazzola.di.unimi.it/aide/neverlangjs-lpl.git.tgz`.

To setup a copy of the repository on your machine please download it, extract it, switch to the new directory and run: **`git clone .git neverlangjs-lpl`**.

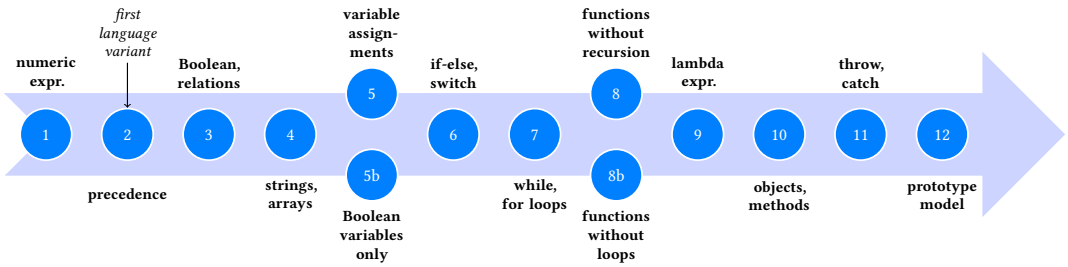

Figure 4: Course schedule and language products for teaching JavaScript, adapted from [5].

Although JavaScript provides a realistic level of complexity and variety of language features, its implementation only amounts to 3599 *lines of code* (LoC) in 79 slices, 83 modules, and 3 endemic slices and support classes. Moreover, the neverlang.JS interpreter mostly conforms to the *ECMAScript 3 Language Specification* (ECMA-262), except for several built-in functions. Despite its limited size, the FM generated by AiDE, as partially depicted in Figure 5, comprises 234 language features (including 43 abstract features) and 162 cross-tree constraints.[5] Notably, some of them are redundant, this, however, results from the individual generation of cross-tree constraints. As such, the FM gives rise to at least 139713 valid feature configurations (i.e., fulfilling all tree and cross-tree constraints) and corresponding language variants (estimated via FeatureIDE's number of products analysis). Note that this number is an underestimation because the Neverlang renaming mechanism could still be used to derive viable language variant from invalid feature configurations. Unfortunately, we cannot give a more accurate estimate on the number of viable language configurations and, thus, language variants, as the viability of a renaming depends on the language implementation.

5.2 Teaching a Growable Language

For our demonstration case study, we adapted the teaching schedule, proposed in [5], for gradually teaching programming to students. Figure 4 highlights the 14 language variants (circles) which gradually introduce new language features over the duration of the programming course. In particular, we included three additional language variants. The first, Variant 5b, is a language specialization that only permits Boolean expressions and the declaration of Boolean variables. It mirrors Variant 5, yet focuses on propositional logical formulas. In contrast, Variant 8 and Variant 8b were introduced to teach recursion or lack thereof. While the former, permits function calls yet prohibits recursion, the latter supports recursive function calls yet removes loops from the language variant. Henceforth, we will take the perspective of the teacher as the language deployer tasked to configure and deploy the 14 language variants to students as language users.

5.3 Growing a JavaScript Variant

To create the increasingly complex language variants, the language developer clones the neverlang.JS git project provided by the language developer and opens it within our LPL development environment. Then they use the *Neverlang Language Configuration* wizard to create a new language configuration and open it in the *Neverlang Language Configuration Editor*. Initially the configuration is empty and they proceed by selecting the desired language features from the *Configuration* tab, as in the second layer of Figure 3. Henceforth, we use a dot-notation to denote the path to a feature from the root of the FM, shown in Figure 5.

In our case, we started with Variant 2, a language variant only allowing numerical expressions. For this example, Figure 5 highlights the relevant features with a yellow background. The language deployer starts by selecting the desired language features for this language specialization found under `expression.numbers`, i.e., `sum.JSAdditiveExpression` and `literal.JSNumbericLiteral`. Additionally, the corresponding `evaluation` semantic features are selected too. The configuration editor automatically selects `endemics.JSMathEndemic` but yields an invalid configuration due to unsatisfied cross-tree constraints. However, it suggests to select `mul.JSMultiplicativeExpression` and `literal.JSNumbericLiteralProd` from the `expression.numbers` subtree. Although this yields a valid configuration, the resulting language variant is still empty, as the selected productions are never reached. The language deployer additionally selects the language features leading from the start symbol (`Program` by convention) to the numerical expressions to make them reachable. Such features include `JSMain` and `statement.JSStatement` (top of Figure 5) and `JSExpressionStatement` and `JSPrimaryExpressions` (from the `expression` subtree). At this point, the configuration editor suggests the `JSExpression` feature which cascades into additional dependencies that the language variant should not include. In fact, at this point the deployer requests help from the language developer who, in turn, assesses that in this case renaming is not enough and introduces two alternative language components, i.e., `AddExprRestriction` and `UnaryExprLiteralRestriction`, that introduce the missing productions needed to make this language variant viable. Immediately, after adding the new language components the FM is regenerated. The language developer can now commit the newly created Neverlang modules and the changes to the FM file to the repository. The language deployer pulls the latest changes which cause FeatureIDE to automatically reload the *Neverlang Language Configuration Editor*, i.e., publishing the corresponding language features within the `expression.restrictions` subtree. Finally, the configuration is valid and the deployer only needs to declare the sequence of the semantic actions and corresponding traversal, i.e., *evaluation* with *preorder*, in the *Roles* tab. Once the configuration is saved, the language variant is automatically generated and ready

[5]The complete feature model is available: `https://cazzola.di.unimi.it/aide/neverlangJS-fullfm.png`.

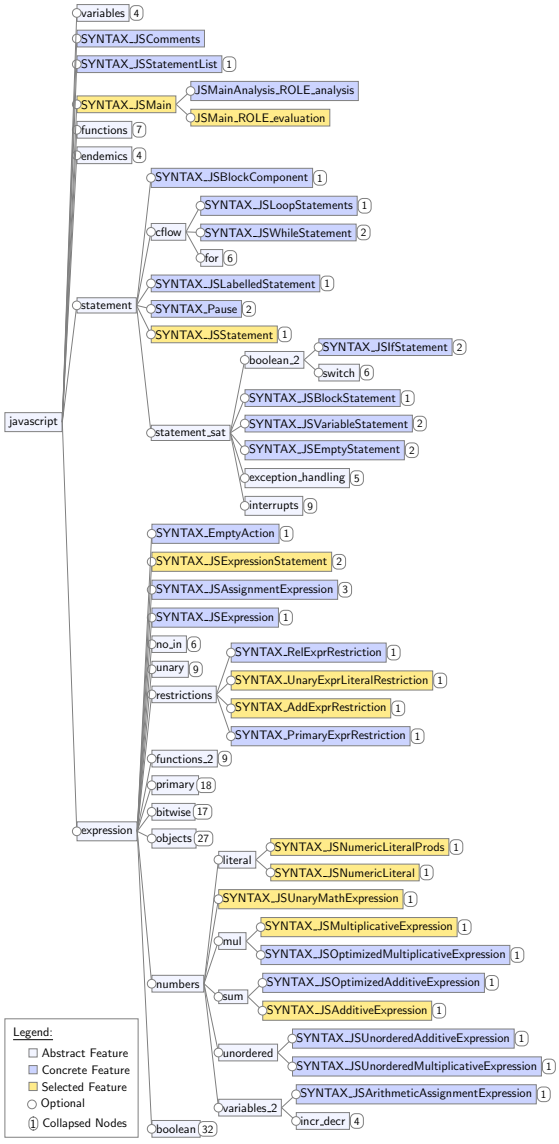

Figure 5: FM generated by AiDE for the JavaScript-family.

to be published to the repo for testing. To do this, the *language user* can simply update the repository and run the generated Java program within the `gencode.aide` package in the `gen-src` folder as *Java application*. This starts an interactive interpreter of the language variant in the *Console*. After evaluating that the language variant correctly parses and evaluates the desired numerical expressions, they can proceed to package the language variant via gradle and registering it to Neverlang selecting a unique file extension, e.g., `js02`. As a result, language users can now use the *Neverlang Editor* to edit all `js02` files with the correct syntax highlighting and code-completion. They can also run the files using the *Neverlang run configuration* with the corresponding language variant, i.e., *Variant 2* and file path, as showcased in [21]. Figure 3 (language user layer) showcases the editors for the first four language variants with syntax highlighting and code-completion, and the execution of the `factorial.js04` program with the fourth language variant.

Building on these language variants, the language deployer can reuse previous language configurations to derive the other language variants. Moreover, with feedback from the language developer they can also derive language variants, which require renaming, such as, Variant 4 to exclude assignments and Variant 5b to exclude bitwise operations and relations, or diverging semantics, such as, Variant 1 requiring language components violating the precedence rules and Variant 8 which prohibits recursive function calls. Henceforth, we change the perspective to the language developer detailing how Variant 5b and 8b were facilitated.

5.4 Refactoring the neverlang.JS LPL

Variant 5b was derived restricting variant 5 by removing support for numerical values, strings and arrays to allow the language deployer to configure a language variant of only Boolean assignments and expressions. Introducing the new variant in the LPL highlighted a refactoring opportunity. Listing 2 showcases the implementation of assignments before the refactoring process. The syntax definition for the `JSAssignmentExpression` `module` was too coarse-grained and required the `AssignOperator` non-terminal to be defined leading to invalid language configurations when numerical values and arithmetic assignment operators are not present in the variant. To fix this we refactored the `JSAssignmentExpression` into two **`modules`** (cf. Listing 3) to distinguish standard assignments from arithmetic assignments, e.g., `+=`. In total, the added `JSArithmeticAssignmentExpression` amounts to 61 additional LoC. After completing the refactoring the LPL development environment automatically compiles all changed language components and regenerates the FM. Then the language developer only needs to review and save the FM via the FM Editor. As previously outlined, this change is propagated to all Language Configuration Editors. Additionally, all variants 5–13 that were using the modified `JSAssignmentExpression` feature are marked with a warning indicating that they need to be reviewed by the language deployer.

A similar refactoring process was required to derive Variant 8 from Variant 8b to introduce functions. Recall, Variant 8 adds functions with loops but prohibits recursive calls whereas Variant 8b permits recursive functions but lacks loops. For the former, this required selectively changing the language semantics. Hence, the language developer opted for preceding the evaluation with an additional semantic role, i.e., the `analysis` role. This role was added to `JSFunctionDeclaration`, `JSFunctionCalls`, and `JSMain` to retrieve the static call graph from the parsed program and prevent its evaluation when a cycle is detected.

```
1 module neverlang.js.variables.JSAssignmentExpression {
2   reference syntax {
3     provides { AssignExpr: expression, variables; }
4     requires { LeftHandSideExpr; AssignOperator; }
5     AssignExpr ⟵ LeftHandSideExpr "=" AssignExpr;
6     AssignExpr ⟵ LeftHandSideExpr AssignOperator AssignExpr;
7   }
8   role (evaluation) {/*...*/}
9 }
```

Listing 2: Too coarse-grained `module` defining assignment.

```
module neverlang.js.variables.JSAssignmentExpression {
  reference syntax {
    provides { AssignExpr: expression, variables; }
    requires { LeftHandSideExpr; }
    AssignExpr ⟵ LeftHandSideExpr "=" AssignExpr;
  }
  role (evaluation) {/*...*/}
}
module neverlang.js.variables.JSArithmeticAssignmentExpression {
  reference syntax {
    provides {
      AssignExpr: expression, variables, numbers, strings;
      AssignOperator: ..., operators;
    }
    requires { LeftHandSideExpr; AssignExpr; }
    AssignExpr ⟵ LeftHandSideExpr AssignOperator AssignExpr;
    AssignOperator ⟵ "+="; // other rules for *= /= %= -=
  }
  role (evaluation) {/*...*/}
}
```

Listing 3: Assignment split into separate `modules`.

```
module neverlang.js.analysis.JSFunctionDeclarationAnalysis {
  reference syntax from
    neverlang.js.functions.JSFunctionDeclaration
  role(analysis) {
    func_decl: .{
      String foo = $func_decl[1].toTerminalString();
      String args = $func_decl[2].toTerminalString();
      $$CallStackBuilder.declare(foo, args.split(","));
      eval $func_decl[3]
      $$CallStackBuilder.pop();
    }.
  }
}
module neverlang.js.analysis.JSFunctionCallsAnalysis {
  reference syntax from
    neverlang.js.expression.JSFunctionCalls
  role (analysis) {
    c_expr: .{
      String foo = $c_expr[1].toTerminalString();
      String args = $c_expr[2].toTerminalString();
      $$CallStackBuilder.call(foo, argsString.split(","));
    }.
  }
}
```

Listing 4: Semantic actions for creating the call graph.

Listing 4 shows the implementation of the semantic actions added to function declarations and function calls. The former uses a stack to keep track of the current function in scope whereas the latter adds call edges from the current function to the called function. Excluding the employed third-party graph library, this extension only introduced 52 new LoC. The impact of this refactoring on the FM is limited to the added language features without changing its structure. Thus no other language configuration was affected by the change. Hence, to configure Variant 8, the language deployer only needs to pull the FM with the new features, add those features to the configuration, select the corresponding analysis role and list it as preceding the *evaluation* phase in the *Roles* tab.

To recap, these cases illustrate the benefits of a tight feedback loop from the language variant's deployer to the LPL developer in a distributed environment, such that the language developer only needs to make small changes to the LPL to gradually make more of the language deployer's language variants viable.

5.5 Comparison of the 15 Language Variants

During the course of this case study, we incrementally created 15 distinct language variants of increasing complexity including the 14 variants for the teaching schedule and the full-fledged version of JavaScript while refactoring the underlying neverlang.JS LPL according to need. Accordingly, Table 1 provides an overview on the 15 created language variants. For each language variant it highlights the used semantic roles. Moreover, it indicates the total number of selected (abstract and concrete) features and in brackets the number of features introduced and selected for language specialization. The four newly introduced features were needed for all the variants except 6, 7 and the full-fledged JavaScript whereas variants 2, 3 and 4 needed two specializations each. `UnaryExprLiteralRestriction` was used eleven times, `RelExprRestriction` was used twice and the remaining two slices were used once. The addition of language components for specialization was mainly needed to deal with the *domino effect*, but in case of Variants 4 and 5 it could be resolved by renaming nonterminals, as indicated in the *Renames* column. In these two cases, the automatic feature selection was disabled. Consequently, all but these two language configurations are considered valid by *FeatureIDE*, yet all produce viable language variants. To sum up, there were a total of eight language features that caused the *domino effect*, four of which were solved by adding dedicated features to the LPL and four by means of nonterminal renaming.

From our experience, we observed that the distributed, incremental development, rapid testing and deployment significantly reduced the effort to create new and provision language variants. Due to the power of FeatureIDE's feature configuration with automatic feature selection and feature suggestion, the creation of viable language variants was significantly reduced when compared to manually writing Neverlang language files. Additionally, the total number of LoC required to build each language variant from scratch is shown in the last column whereas the lines of included Java code is shown in brackets. Granted, this assumes that each language variant would have been built from scratch using *Neverlang*, still it illustrates how LPL engineering could speed up the creation of language variants and improve reuse among members of a family of languages. Last but not least, with this case study we could illustrate the suitability of our LPL development environment for the teaching case, as it simplified the teacher's task to create viable language variants. Conversely, we argue that our LPL development environment is applicable for the distributed, incremental development, configuration and deployment of LPLs, as it directly supports the LPL engineering process. As showcased in this section, our LPL development environment provides all views and services required/expected by language developers, deployers, and users.

6 RELATED WORK

Language workbenches and the development of domain-specific languages are established research topics. The problem of IDE support for DSLs is well known. It is addressed by most recent language workbenches [14], such as, Spoofax [35], MPS [33], MontiCore [18] and Melange [11]. Some of these approaches generate an IDE using templates, thus neglecting feature modularity and the specific characteristics of the DSL under development. EMF-based tools [28] such as EMFText [16] support modular language implementation

Table 1: Overview on the 15 NeverlangJS language variants highlighting the selected features including features for language specialization in brackets as well as total lines of code (LoC) including Java code in brackets.

Variant	Description	Roles	Features	Renames	LoC
1	Numeric expressions and operators (without precedence)	evaluation	37 (11)	-	330 (80)
2	Numeric expressions and operators (correct precedence)	evaluation	42 (5)	-	354 (82)
3	Booleans and relational operators	evaluation	70 (5)	-	780 (215)
4	Strings, arrays and their operators	evaluation	76 (5)	1	866 (229)
5	Variables and assignments	evaluation	53 (3)	-	1356 (350)
5b	Only Boolean assignments and expressions	evaluation	106 (3)	3	641 (154)
6	Conditional statements (e.g., if, else, switch)	evaluation	145 (0)	-	2010 (571)
7	Loop statements (e.g., while & for)	evaluation	162 (0)	-	2479 (821)
8	Loops and functions without recursion	analysis, evaluation	171 (10)	-	2727 (877)
8b	Functions with recursion, but without loops	evaluation	183 (3)	-	2470 (739)
9	Functions and lambda expressions	evaluation	181 (3)	-	2766 (884)
10	Objects and Methods	evaluation	186 (3)	-	2937 (974)
11	Exception Handling	evaluation	193 (3)	-	2990 (984)
12	Constructors and prototype model	evaluation	208 (3)	-	3224 (1054)
Complete	Variant conforming to ECMAScript 3	debug, evaluation	234 (0)	-	3599 (1194)

and IDE generation for DSLs. EMFText is similar to Neverlang since it uses attribute grammars to share IDE implementations through languages, however it does not explicitly consider language variability for LPL development. Among the most successful approaches, Monticore, Spoofax and MPS directly or indirectly provide LPL engineering capabilities. Monticore directly supports compositional development of DSLs and IDEs by means of language embedding and inheritance. Butting *et al.* [3] presented an approach to manage syntactic variability of extensible LPLs using Monticore. Spoofax supports generation of a wide variety of IDE tools for Eclipse and IntelliJ including syntax highlighting, code-completion and parse error recovery, but also the creation of language configurations. LPLs are not addressed directly but emerge from the incremental development of language features and language variants. Liebig *et al.* [24] used Spoofax alongside FeatureHouse for the representation and composition of language features. MPS offers full IDE support and customizable abstract syntax tree manipulation through the Projection Editor. Languages can be defined as standalone languages or as modular extensions of existing languages. Most notably, mbeddr [34] is a project built on top of MPS, presented as a set of integrated and extensible languages based on C for embedded software engineering with an IDE and support for SPL development. MPS arguably represents a fully-fledged LPL development environment with IDE support, however, to the best of our knowledge, there is no contribution for a dedicated bottom-up LPL development environment covering all the evolution phases of an incremental LPL, including creation, development of language artifacts, configuration and deployment. Other works focus on one or a few of said aspects and embrace LPL engineering in a top-down fashion. Except for previous works on AiDE [19–21, 31, 32], we could not find any reference to a language workbench using a variability management tool capable of abstracting language features, a feature model and cross-tree constraints from language artifacts. Nevertheless, while the engineering process we propose in this paper was tailored for the bottom-up LPLs, we argue that it could be easily adapted to fit the needs of top-down LPLs: in this scenario the language developer should commit to update the FM manually, but the workflow of language deployer and language user would be unchanged.

7 CONCLUSION

Current language development environments lack support for dealing with families of languages. To remedy this, recent LPL development environments approached the development of families of languages, denoted LPL. Yet, none of them could equally support the full LPL engineering process and needs of language developers, language deployers, and language users within one IDE. To this end, we described the underlying (bottom-up) LPL engineering process established between them and introduced an LPL development environment supporting this engineering process: our environment is suitable for distributed, incremental development, language configuration, rapid deployment and testing, as well as, simply working with a language variant. In particular, we married the established SPL engineering environment, FeatureIDE [25], with the LPL development environment Neverlang [30] and AiDE. Moreover, we demonstrated the applicability of our IDE for the development of LPLs using the neverlang.JS LPL. Furthermore, we illustrated the suitability of our LPL development environment for the distributed and incremental development, rapid configuration and deployment of language variants, as well as their use, by creating 15 distinct language variants, adapted from the teaching schedule, used for teaching a growable language [5].

Nonetheless, to improve the usability of our LPL development environment the implementation must be further optimized for better performance and robustness, as well as include more of the features already provided by Neverlang, such as agents for concerns cross-cutting multiple language modules, guards for multi-actions and support for remapping nonterminals.

ACKNOWLEDGMENTS

We want to thank Ralf Reussner who partially funded this project. We also want to thank Sebastien Krieter, Thomas Leich and Gunter Saake for their insights on the FeatureIDE implementation and their interest in this work.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts.

[2] Sven Apel, Alexander von Thein, Philipp Wendler, Armin Größlinger, and Firk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, Betty H. Chang and Klaus Pohl (Eds.). IEEE, San Francisco, CA, USA, 482–491.

[3] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software Intensive Systems (VAMOS'18)*. ACM, Madrid, Spain, 75–82.

[4] Walter Cazzola. 2012. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proceedings of the 11th International Conference on Software Composition (SC'12) (Lecture Notes in Computer Science 7306)*, Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book (Eds.). Springer, Prague, Czech Republic, 162–177.

[5] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (Sept. 2016), 404–415. https://doi.org/10.1109/TETC.2015.2446192 Special Issue on Emerging Trends in Education.

[6] Walter Cazzola and Davide Poletti. 2010. DSL Evolution through Composition. In *Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*. ACM, Maribor, Slovenia.

[7] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2: Componentised Language Development for the JVM. In *Proceedings of the 12th International Conference on Software Composition (SC'13) (Lecture Notes in Computer Science 8088)*, Walter Binder, Eric Bodden, and Welf Löwe (Eds.). Springer, Budapest, Hungary, 17–32.

[8] Zhiqun Chen. 2000. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley, Reading, MA, USA.

[9] Michele Chinosi and Alberto Trombetta. 2012. BPMN: An Introduction to the Standard. *Computer Standards and Interfaces* 34, 1 (Jan. 2012), 124–134.

[10] Michelle L. Crane and Juergen Dingel. 2005. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05) (Lecture Notes in Computer Science 3713)*, Lionel Briand and Clay Williams (Eds.). Springer, Montego Bay, Jamaica, 97–112.

[11] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, Davide Di Ruscio and Markus Völter (Eds.). ACM, Pittsburgh, PA, USA, 25–36.

[12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12)*, Anthony M. Sloane and Suzana Andova (Eds.). ACM, Tallinn, Estonia.

[13] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-Based Syntactic Language extensibility. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'11)*. ACM, Portland, Oregon, USA, 391–406.

[14] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Alex Kelly, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van del Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures* 44 (Dec. 2015), 24–47.

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

[16] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. 2011. Model-Based Language Engineering with EMFText. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11) (Lecture Notes in Computer Science 7680)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, Braga, Portugal, 322–345.

[17] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[18] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 5 (Sept. 2010), 353–372.

[19] Thomas Kühn and Walter Cazzola. 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *Proceedings of the 20th International Software Product Line Conference (SPLC'16)*, Rick Rabiser and Bing Xie (Eds.). ACM, Beijing, China, 50–59.

[20] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, Goetz Botterweck and Jules White (Eds.). ACM, Nashville, TN, USA, 71–80.

[21] Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi. 2019. Piggyback IDE Support for Language Product Lines. In *Proceedings of the 23rd International Software Product Line Conference (SPLC'19)*, Thomas Thüm and Laurence Duchien (Eds.). ACM, Paris, France, 131–142.

[22] Thomas Kühn, Ivo Kassin, Walter Cazzola, and Uwe Aßmann. 2018. Modular Feature-Oriented Graphical Editor Product Lines. In *Proceedings of the 22nd International Software Product Line Conference (SPLC'18)*, Paulo Borba and Thorsten Berger (Eds.). ACM, Gothenburg, Sweden, 76–86.

[23] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Proceedings of the 7th International Conference Software Language Engineering (SLE'14) (Lecture Notes in Computer Science 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jürgen Vinju (Eds.). Springer, Västerås, Sweden, 141–160.

[24] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Philippe Collet and Klaus Schmid (Eds.). ACM, Pisa, Italy.

[25] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer.

[26] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlberg. 2011. *The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis.* White Paper. Microsoft.

[27] Klaus Pohl, Klaus Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer.

[28] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework.* Addison-Wesley.

[29] Laurence Tratt. 2008. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems* 30, 6 (Oct. 2008), 31:1–31:40.

[30] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures* 43, 3 (Oct. 2015), 1–40. https://doi.org/10.1016/j.cl.2015.02.001

[31] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, Patrick Heymans and Julia Rubin (Eds.). ACM, Florence, Italy, 167–176.

[32] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *Proceedings of 6th International Conference on Software Language Engineering (SLE'13) (Lecture Notes on Computer Science 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, Indianapolis, USA, 76–95.

[33] Markus Völter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, Zürich, Switzerland, 1449–1450.

[34] Markus Völter, Daniel Ratiu, Bernhard Schätz, and Bernd Kolb. 2012. mbeddr: an Extensible C-Based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12)*. ACM, Tucson, AZ, USA, 121–140.

[35] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (Sept./Oct. 2014), 35–43.