Piggyback IDE Support for Language Product Lines

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

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

Nicola Pirritano Giampietro
Computer Science Department
Università degli Studi di Milano, Italy
nicola.pirritanogiampietro@studenti.unimi.it

Massimiliano Poggi
Computer Science Department
Università degli Studi di Milano, Italy
massimiliano.poggi@studenti.unimi.it

ABSTRACT

The idea to treat domain-specific languages (DSL) as software product lines (SPL) of compilers/interpreters led to the introduction of language product lines (LPL). Although there exist various methodologies and tools for designing LPLs, they fail to provide *basic IDE services* for language variants—such as, syntax highlighting, auto completion, and debugging support—that programmers normally expect. While state-of-the-art language development tools permit the generation of basic IDE services for a specific language variant, most tools fail to consider and support reuse of basic IDE services of families of DSLs. Consequently, to provide basic IDE services for an LPL, one either generates them for the many language variants or designs a separate SPL of IDEs scattering language concerns. In contrast, we aim to piggyback basic IDE services on language features and provide an IDE for LPLs, which fosters their reuse when generating language variants. In detail, we extended the Neverlang language workbench to permit piggybacking syntax highlighting and debugging support on language components. Moreover, we developed an LPL-driven Eclipse-based plugin that includes a syntax highlighting editor and debugger for an LPL with piggybacked basic IDE services, i.e., where modular language features include the definition for syntax highlighting and debugging. Within this work, we introduce a general mechanism for fostering the basic IDE services' reuse and demonstrate its feasibility by realizing context-aware syntax highlighting for a Java-based family of role-oriented programming languages and providing debugging support for the family of JavaScript-based languages.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Integrated and visual development environments**; **Software product lines**;

KEYWORDS

Domain Specific Languages, Language Product Lines, Integrated Development Environment, Feature Modularity, Neverlang

ACM Reference Format:
Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi. 2019. Piggyback IDE Support for Language Product Lines. In *23rd International Systems and Software Product Line Conference - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3336294.3336301

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09…$15.00
https://doi.org/10.1145/3336294.3336301

1 INTRODUCTION

Tools for the development of *domain-specific languages* (DSL) and programming languages have made a major leap. Employing state-of-the-art language workbenches, designing custom DSLs or extensions to established languages became feasible for researchers and practitioners alike [15]. This led to a large amount of DSLs and multiple language extensions. Recently, researchers started investigating combining different language variants to develop families of DSLs as well as programming languages, e.g., [16, 25, 28]. Yet, as state-of-the-art language development tools have limited support for reusing language features[1] between different languages, they are not suitable to develop families of DSLs and programming languages. To overcome their limitations, researchers currently apply ideas from *software product lines* (SPL) to embrace the need for multiple variants of a language. Simply put, language families can be created as an SPL of compilers/interpreters, whereas each product corresponds to a language variant [23]. These product lines are denoted *language product lines* (LPL) and have been successfully employed for families of DSLs [17, 26, 33, 39, 40] and general purpose programming languages [6, 22, 23]. While these approaches introduced various methodologies and tools to design LPLs, they fail to provide an *integrated development environment* (IDE) for users of a selected language variant. This hinders the acceptance of these approaches among users and programmers, as they expect at least *basic IDE services*, such as syntax highlighting, auto completion, and debugging support. Although language development tools permit easy generation and/or implementation of basic IDE services for a specific DSL [15], it is infeasible to generate and/or implement them for all possible language variants. Similarly, designing and maintaining a separate SPL of IDEs corresponding to

[1] In line with [23], *language features* are either language constructs, e.g., *for loop*, or language concepts (without concrete syntax), e.g., *scope* and *coercion*.

the LPL leads to scattering of language concerns between two product lines. Arguably, a better solution follows *"the road to feature modularity"* [19] and packs a language feature together with its basic IDE services. Hence, we aim to piggyback basic IDE services on language features and, therefore, to automatically provide IDE support for LPLs, which fosters their reuse when generating language variants. In fact, we extended the Neverlang language workbench to piggyback syntax highlighting and debugging support on language components. Moreover, we propose an LPL-driven IDE that includes a syntax highlighting editor and debugger for arbitrary LPLs, whose basic IDE services ride piggyback on language features, i.e., where modular language components include the definition for syntax highlighting and debugging. To demonstrate its feasibility, we implemented an LPL-driven IDE, an Eclipse plugin providing a context-aware syntax highlighting editor and an LPL-driven debugger. To further illustrate its applicability for basic IDE services we employ two case studies. First, we showcase context-aware syntax highlighting for the family of Java-based role-oriented programming languages (RPLs) [22]. Second, we demonstrate adding and employing debugging support for the family of JavaScript-based languages [23]. In conclusion, piggybacking basic IDE services on language features enhances the modularity and reuse of basic IDE services and permits using an LPL-driven IDE.

The paper is structured as follows. In the beginning, Sect. 2 briefly introduces LPLs, the Neverlang language workbench, the corresponding AiDE LPL configurator, and the notion of IDE services. Sect. 3 proposes our approach for piggybacking basic IDE services on language components of LPLs. Sect. 4 presents our implementation. Sect. 5 outlines the case studies we conducted on two different LPLs. The paper is concluded by discussing related approaches in Sect. 6, and summarizing our results in Sect. 7. The Appendix highlights the artifact provided to reproduce our results.

2 PRELIMINARIES

2.1 Language Product Lines

The development of families of programming and domain-specific languages has gained popularity among researchers and practitioners, e.g., [16, 25, 28]. Following the ideas of software product lines (SPLs), a LPL facilitates the process of language development, which can be *customized* by selecting individual *features*. Similar to SPLs, a language could be designed to specifically suit a certain use case or application domain. For instance, authors [11, 31, 34] have shown that the many variants of state machine languages could be modeled as one single family of programming languages. Nonetheless, this is also true for *general-purpose programming languages*, from which *dialects* may be defined for DSL purposes. On one side, specialized versions of full-fledged programming languages can be employed in case of security purposes (*e.g.*, Java Card [10]) or teaching programming [6, 13]. 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 [14].

2.2 Neverlang[2] in a Nutshell

The Neverlang [4, 8, 32] framework is built around the *language feature* concept. Language components, called *slices*, embodying the language features are developed as separate units that can be compiled and tested independently, enabling developers to share and reuse the same units across different language implementations. Here the development base unit is the `module` (Listing 1). A module may contain a `syntax` definition and/or semantic roles. A `role` defines 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`**. Let us consider the Neverlang realisation of the `State` module for the vacuum cleaner example shown in Listing 1.

```
1  module StateModule {
2    reference syntax {
3      State <- StateName "=" Expr ;
4    }
6    role(evaluation) {
7      0.{ newstate $1 state, $2 action); }.
8    }
9  }
11 slice State {
12   concrete syntax from StateModule
13   module StateModule with role evaluation
14 }
16 language HooverLang {
17   slices Program StateDecl EventDecl TransDecl StateLst State
18       StateName EventList Event EventName Expr BExpr TransList
19       Transition Support
20   roles syntax < execution
21 }
```

Listing 1: Syntax and semantics for the state concept.

Here the module `StateModule` declares a reference syntax for the state concept (Lines 2-4) and actions are attached to the nonterminals on the right of the production (Line 7). Semantic actions are attached to nonterminals by referring to their position in the grammar or through a label: numbering starts with 0 from the top left to the bottom right. Thus, the first `State` on Line 3 is referred to as 0, `StateName` as 1, and `Expr` as 2. The slice `State` declares in Line 12 that we will be using *this* syntax (which is the `concrete syntax`) in our language, with those particular semantics (Line 13). Finally, the `language` descriptor indicates (Lines 17-19) which slices are to be composed together to generate the language interpreter. Composition in Neverlang is, therefore, twofold: (1) between modules, which yields slices, and (2) between slices, which yields a language implementation. The composition result is independent of the order of specified slices. 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 order specified in the **`roles`** clause of the `language` descriptor. Please see [32] for further details.

2.3 AiDE[3] in a Nutshell

AiDE [22, 23] is an interactive configuration tool especially tailored to develop language product lines. It implements the method presented in [23, 33, 34] to automatically synthesize the *feature model* of a given language family out of language components developed with Neverlang [32]. Through its graphical user interface, depicted in Fig. 1, the user can explore the feature model, choose features,

[2] Available at https//neverlang2.di.unimi.it
[3] Available at https://aide.di.unimi.it.

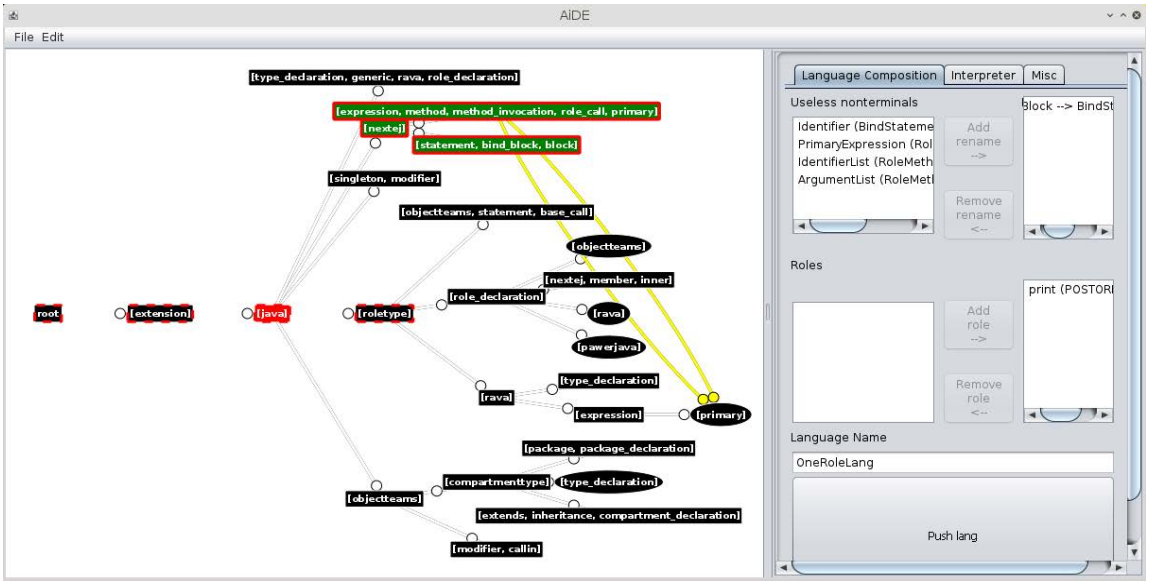

Figure 1: AiDE language configurator for the family of RPLs employed in [22].

create a language variant, and test it. Because feature models of LPLs tend to be large [22], AiDE initially shows the first level of the tree, however, allowing it to be expanded on demand.
Moreover, while the user configures a language variant (or product), AiDE tracks all unresolved dependencies—i.e., all open nonterminals in the current selection—and provides the user with mechanisms, such as renaming, to bind them to other nonterminals already in the selection. Thus, users can easily resolve dependencies during the component 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. When the language satisfies the expectations, a stable copy of the development environment is prepared and ready to be dispatched to any JVM compliant workstation. In sum, AiDE is able to guide users towards the generation of consistent language variants by supporting multiple dependency resolution strategies and continuous generation of the language's compiler/interpreter.

2.4 Basic IDE Services

IDEs provide basic IDE services, like syntax highlighting, code completion, error marking, and debugging support. However, in line with Erdweg *et al.* [15], we distinguish between *syntactic services* and *semantic services*. Syntactic services only depend on the language's syntax and are usually generated from its grammar, e.g., a syntax highlighting editor, an outline view, syntactic completion, and a pretty printer. In contrast, semantic services need to take the language's semantics into account. Yet, only the simplest of them, e.g., reference resolution and semantic completion, can be automatically generated from a specified language. Yet, more complex services require additional programming effort, such as refactoring, quick fixes, and debugging. Granted syntactic services are easier to integrate into an LPL, we argue that programmers equally require syntactic and semantic services from their IDE for a given language. Hence, we focus on syntax highlighting as a syntactic service and debugging as a complex semantic service.

3 PIGGYBACKING IDE SERVICES ON LANGUAGE COMPONENTS

Our main goal is to provide feature modularity and reuse for *basic IDE services* wrt. the language features they correspond to. To reach this, we have to reconsider feature modularity for languages. As Kästner *et al.* [19] argued, feature modularity relies on locality and cohesion, as well as on information hiding and encapsulation.

3.1 Integration into Language Product Lines

In case of LPLs, this entails that all information relevant for a language feature including the corresponding part of an IDE service should be part of one language component. This section shows how partial *IDE service specifications* relevant for a language feature can be added to each language component. Moreover, these specifications can be interpreted inside an LPL-driven IDE to dynamically provide the embedded syntactic and semantic services. Notably though, the possible integration of these service specifications into an LPL greatly depends on the nature of the IDE service.

3.2 Integration of Syntactic Services

Integrating a syntactic service into language components is straightforward. First, the service specification is decomposed wrt. to the syntactic production they contribute to.

```
module neverlang.statement.TryCatchStatement {
  reference syntax {
    Statement    <- TryStatement ;
    TryStatement <- "try" Block CatchClause ;
    TryStatement <- "try" Block CatchClause Finally ;
    CatchClause  <- "catch" "(" CatchParameter ")" Block ;
    Finally      <- "finally" Block ;
  categories :
    Keyword  = { "try", "catch", "finally" }
      with style "trycatch.json",
    Brackets = { "(", ")" } with style "java.json" ;
  }
  /* ... */
}
```

Listing 2: Terminal categorization in a language component.

```
{ "Keyword": { "italic": true, "color":      "#7f0000",
               "bold":   true, "background": "#8F8F8F" } }
```

Listing 3: Style definition per category (`trycatch.json`).

Afterwards, the language component is extended to permit adding a partial specification of a corresponding syntactic service, as well as an interface to retrieve this specification. Finally, an LPL-driven IDE will query this interface of selected language components to provide the desired syntactic IDE service.

In case of syntax highlighting, the language component is extended with a style specification for each *terminal* in each production (right hand side). Although this approach would be feasible, if the style specification is compiled into the language components, the user cannot easily change the highlighting of a specific language feature. To remedy this, we permit language developers to define a custom category for each terminal in a language component.
Simply put, a category is a name employed to retrieve a terminal's highlighting from a user controlled style file. Conversely, we augmented Neverlang *slices* to allow for defining categories for sets of terminals, as outlined in Listing 2 for the `TryCatchStatement` slice. In particular, the slice defines the category `Keywords` for the terminals `try`, `catch`, and `finally`, as well as the category `Brackets` for curly braces. Additionally, the `with style` clause denotes the user controlled file from where the highlighting style should be retrieved. Thus, while the IDE uses the default Java highlighting (`java.json`) for braces, it employs the style definition in `trycatch.json` for the keywords `try`, `catch`, and `finally`, shown in Listing 3. Specifically, the style definition does not only permit changing the font style, but also the font color and background color. Note that, `with style` enables overriding the default style of a category, such that the keywords of the try catch statement are highlighted in the context of this language component. In sum, while slices defines the categories for terminals, style files define the actual highlighting of terminals inside the textual editor.

3.3 Integration of Semantic Services

Compared to syntactic services, integrating semantic services into an LPL is complex, due to the fact that they depend on the language's semantics. Especially, this entails that the language's semantic actions must be intercepted and/or adapted by the IDE to

```
module neverlang.statement.Throwable {
  reference syntax {
    Statement      <- ThrowStatement;
    ThrowStatement <- "throw" Expression SemiColonOpt;
    /*...*/
  }
  role (debug) {
    0 @{ $0.isExecutionStep = true; }.
  }
  role (evaluation) {
    0 @{ /*...*/ }.
    1 @{ /*...*/ }.
   }
}
```

Listing 4: Adding debugging to a language component.

extract the required information for semantic services, such as reference resolution, error marking, and debugging support. Most generative LPL approaches, i.e., that generate interpreters/compilers of language variants, do not support the dynamic adaptation of their semantic actions. Yet, recently Cazzola and Shaqiri introduced *open programming language interpreters* for Neverlang-based LPLs, which enables language developers to adapt languages' semantic actions at predefined hooks [7]. As illustrated in Fig. 2, Neverlang encodes semantic actions, e.g., the evaluation of an expression, as attributes of nodes of the parse tree, such as $E.val \leftarrow E_1.val * E_2.val$, which are computed upon traversing the parse tree. By contrast, open interpreters allow language developers to attach language agents to hooks of syntax nodes (e.g., `E`) without requiring to change the language variant's implementation. Depending on the hook the attached agents are called before, instead, or after the node's evaluation [7].

Conversely, if the LPL is implemented as Neverlang's open programming language interpreter, it becomes possible to add a debugging service by adding a debugging language agent to an LPL. This can be done independently of the LPL, as long as the encompassed language components provide a `debug` role, which is evaluated before the actual `evaluation` role. Inside the `debug` role, the language developer only needs to add semantic actions to those productions that represent a computation step. These actions only mark a syntax node as an execution step for the debugger to distinguish between executable statements and non-executable language elements. For instance, the `Throwable` *slice*, depicted in Listing 4, for the **`throw`** statement. Here, the `debug` role only adds a semantic action to the `ThrowStatement` (Line 8) adding an attribute `isExecutionStep` set to true. During the evaluation the debugging agent checks whether this attribute is set for the syntax node, to decide when to suspend the debugger. The implementation of the language agents is outlined in Sect. 4.3.

4 A LANGUAGE PRODUCT LINE-DRIVEN IDE

After integrating basic IDE service specifications into an LPL, an LPL-driven IDE is needed to enable both researchers and practitioners to utilize them. This section describes a design process for a language engineer to configure and deploy a Neverlang-based LPL with included IDE services to an LPL-driven IDE. Moreover, we outline the implementation of our Eclipse plugin, denoted textsfNeverlangIDE, that facilitates our LPL-driven IDE featuring a context-aware syntax highlighting editor, as well as an LPL-driven debugger.

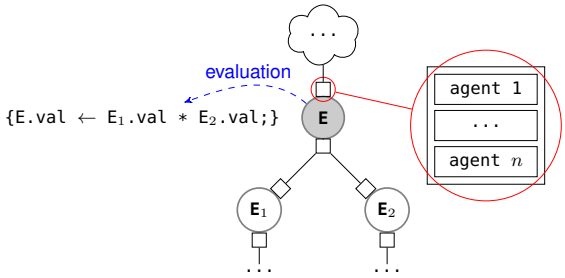

Figure 2: Parse tree with hooks and *before* agents, from [7].

4.1 Language Configuration and Deployment

To benefit from the LPL-driven IDE, the language components must have been implemented as Neverlang slices. These slices must include suitable tags for the feature model generation [23], as well as style *categories* and *debug* roles for syntax highlighting and debugging support (cf. Sect. 3), respectively. As a result, such an LPL can be loaded into AiDE [23], which, in turn, generates the language's feature model and guides the language engineer to choose and pick a valid language variant. After selecting a variant, AiDE generates a corresponding `language` descriptor. From this descriptor Neverlang compiles a corresponding compiler or interpreter for the language variant as an executable Java archive (`jar`). Additionally, this executable `jar` file also contains the style files, the *category* accessors, and the executable *debug* roles. Finally, it can be deployed to any JVM 1.8 compliant system where Neverlang is installed. Yet, to benefit from syntax highlighting and debugging support, an Eclipse *(≥4.10.0)* instance is required where the NeverlangIDE plugin is installed. Now, only the location of the deployed `jar` must be announced to our plugin. After start up, the plugin loads all announced language variants, and allows users to benefit from the LPL-driven Neverlang Editor as well as the debugging support via Neverlang run configurations and an LPL-driven debugger. Henceforth, we will delve into their implementation.

4.2 LPL-Driven Syntax Highlighting

To implement our LPL-driven text editor we extended Eclipse's `TextEditor`. This editor requires an `ITokenScanner` to parse a given resource and decide how to highlight each lexical element. Fortunately, as any Neverlang language can be parsed using its extensible lexer [9, 32, cf. Lexter], we simply implemented the adapter to the `ITokenScanner`, sketched in Listing 5. In detail, the `LexterAdapter` queries for the language descriptor for the file extension of the underlying resource. Using this language descriptor both a Neverlang `lexer` and category-aware `parser` are retrieved. While the former is mandatory, the latter permits context-aware syntax highlighting. This becomes evident in the `nextToken` method, where the next token is retrieved from both the `lexer` (Line 31) and the `parser` (Line 33). In case the `category` retrieved for this token is defined, the `parser` will be requested to return a *contextual style* for this category (Line 38). Because each node in the parse tree is linked to its defining slice, it can expose its category and style definition. By contrast, if this style is **`null`** or the parser failed, syntax highlighting falls back to a style provided for the whole language, whereas the `tokenId` of the `lexer` is used as category (Line 42). Finally, the

```
1  public class LexterAdapter implements ITokenScanner {
2    private final LanguageProvider language;
3    private final LexterStream lexer;
4    private final StyledText editor;
5    private final SemanticHighlighterDexter parser;
6    private QualifiedToken lastToken, lastParserToken;
7    private int lastOffset;
9    public LexterAdapter(String lang,StyledText editor){
10     this.editor = editor;
11     this.language = LanguageProvider.getInstance(lang);
12     this.lexer = (LexterStream) lang.getDexter().getLexter();
13     this.parser =
14       new SemanticHighlighterDexter(language.buildLanguage());
15     this.lastToken = null;
16     this.lastOffset = -1;
17     /*.Initialize lexer and parser..*/
18   }
19   @Override
20   public int getTokenLength(){
21     return (lastToken == null ? -1 : lastToken.text.length());
22   }
23   @Override
24   public int getTokenOffset(){
25     return (lastToken == null ? -1 : lastOffset);
26   }
27   @Override
28   public IToken nextToken(){
29     if (lastToken != null)
30       lastOffset += lastToken.text.length();
31     QualifiedToken nextToken = (QualifiedToken) lexer.getNext();
32     lastToken = nextToken;
33     lastParserToken = parser.nextToken();
34     String category = parser.lastTokenCategory();
35     TextAttribute attr = null;
36     /* Retrieve context-dependent style */
37     if (category != null)
38       attr = parser.contextualStyle(category);
39     /* Fallback to default language style */
40     if (attr == null)
41       attr = language.getTokenToAttributeMapper()
42                      .get(nextToken.tokenId);
43     return new WrappedToken(nextToken, attr);
44   }
45   @Override
46   public void setRange(IDocument document,int offset,int length){
47    /*...*/
48   }
49 }
```

Listing 5: Implementation of the highlighter's `TokenScanner`.

`nextToken` returns an Eclipse `IToken`, which wraps both the Neverlang token and corresponding style. Internally, these tokens are forwarded to Eclipse's syntax highlighting `TextEditor`. For simplicity, we currently do not employ an intelligent damage, repair, and reconciliation strategy and always parse the whole document upon changes. Granted, this incurs a huge performance overhead, yet, suffices as a proof of concept. In sum, our LPL-driven editor does not only permit the syntax highlighting of the programs written in the selected language variant, but can also emphasize the provenance of language features, as will be demonstrated in Sect. 5.1.

4.3 Language Agent-Based Debugging

To provide debugging support for an LPL-driven IDE, we first needed an LPL-driven debugger. Consequently, we implemented a prototypical, external debugger for Neverlang-based languages that can communicate with an external tool both synchronously and asynchronously.

```
1  public class DebugAgent extends Agent {
2    public enum State {
3      INIT, SUSPENDED, RUNNING, STEPPING, DISCONNECTED
4    }
6    private State state=State.INIT;
7    private BreakpointManager breakpoints=new BreakpointManager();
8    private DebugMessageSender debugSender=new DebugMessageSender();
9    private NodeInfo stepOverNode;
11   public DebugAgent(OpenNeverlang interpreter) {
12     super(interpreter); debugSender.send("started");
13   }
14   @Override
15   public void notifyEvent(ExecutionEvent executionEvent) {
16     switch (executionEvent) {
17       case FILE_LOADED:
18         interpreter.registerAgent(this, new AnyPattern(), null,
19           HookType.BEFORE_AND_AFTER); break;
20       case EXECUTION_FINISHED:
21         this.terminate(); break;
22     }
23   }
24   /*...*/
25   private boolean suspendAt(NodeInfo node) {
26     return node.isExecutionStep()
27       && (state == State.SUSPENDED || state == State.STEPPING
28           || breakpoints.hasAssociatedBreakpoint(node));
29   }
30   private void sendStopEvent(NodeInfo node) {/*...*/}
31   @Override
32   public void before(IPatternMatch iPatternMatch) {
33     NodeInfo node=interpreter.getCurrentNode();
34     handleAsyncCommands();
35     if (suspendAt(node)) {
36       sendStopEvent(node);
37       state=State.SUSPENDED;
38       handleSyncCommands(node);
39     }
40   }
41   @Override
42   public void after(IPatternMatch iPatternMatch) {
43     if (interpreter.getCurrentNode().equals(stepOverNode)) {
44       setState(DebugAgent.State.STEPPING);
45       stepOverNode = null;
46     }
47   }
48   @Override
49   public void interpreterChanged() {}
50 }
```

Listing 6: Excerpt of the debugging agent implementation.

This debugger supports the basic operations for setting up *breakpoints*, *suspending* an execution, *stepping into* or *stepping over* a suspended execution, as well as retrieving all variables (including values) of the current execution step. In its core the debugger adapts a provided Neverlang language variant by adding a `DebugAgent`, outlined in Listing 6. This agent is added to *each* node of the parse tree of the program to debug (Line 18) hooking itself before and after a node's evaluation (Line 19). Consequently, the agent has access to all values of a visited node in previous runs, such as the `isExecutionStep` property set in the `debug` role.

The `DebugAgent`'s behavior is determined by its internal state ranging from the initial state `INIT` via the `RUNNING` state through to `STEPPING` and `SUSPENDED` to finally `DISCONNECTED`. The `before` hook method first handles all asynchronous requests (Line 33) and afterwards determines whether the debugger should be `suspendAt` the current node (Line 34). As defined in Lines 25–29, the debugger only stops at execution steps, i.e., nodes where `isExecutionStep` has been set to **`true`**. Besides that, a running debugger is suspended if it reaches a user defined breakpoint, i.e., a node whose line number in the source code equals to the line number of a breakpoint. Otherwise, the debugger stops if it is either `STEPPING` or `SUSPENDED`. In case of `STEPPING`, the debugger will *step into* the next execution step found in the parse tree, e.g., the body of a for loop, the body of a called method or just the next statement. In contrast, *stepping over* requires to evaluate a complete subtree before suspending the debugger again. To achieve this, the debugger memorizes the `stepOverNode` for which a *step over* was issued, and resumes the debugger until this node is reached again in the `after` hook method (Lines 42–47). This ensures that the full subtree has been evaluated, before the debugger is set back into `STEPPING` state. Nonetheless, the language engineer can customize the debugger's behavior. In case of the **`try`** statement, the *step over* could either completely skip both the **`try`** and **`catch`** block by marking the `TryStatement` node as execution step or continue successively in the **`try`** and **`catch`** `Block` by marking each as execution step. As a result, the granularity and stepping behavior of the debugger is customizable by the language engineer through language components.

The LPL-driven debugger is a separate process that can communicate via TCP with any IDE providing a simple command interface and JSON-based data exchange. This simplified the integration of our debugger into Eclipse, as UI actions are delegeted to the debugger, which returns serialized `VariableInfo` and `ValueInfo` objects for each variable, object, and object member in the scope of the current execution step to be shown in the *Variables* view. The `VariableInfo` interface describes a variable with its name, its type, and its value, i.e., a `ValueInfo` object. The `ValueInfo` interface, in turn, represents a value by means of its runtime type and its string representation. The LPL-driven debugger will triggered for Neverlang-based language variants, just like any Java program, by creating a run configuration for a program, selecting the corresponding language variant, and starting the debugger via the **Run ➥ Debug** menu entry. The LPL-driven editor permits to toggle breakpoints and shows the debugger's current position.

5 DEMONSTRATION CASE STUDIES

Admittedly, one could indicate the suitability of an LPL-driven IDE with toy examples, yet we belief that the benefits of piggybacking IDE services on LPLs shine when dealing with realistic LPLs. Consequently, we demonstrate the suitability of our LPL-driven syntax highlighting editor by augmenting the family of Java-based *role-oriented programming languages* (RPLs) [22] with style categories and style definitions for each language extension. Likewise, the suitability of our LPL-driven debugger is illustrated by adding debugging support to the family of JavaScript-based languages [23].

5.1 Family of Role-Oriented Programming Languages

The family of role-based modeling and programming languages was already identified in [25, 29]. A closer examination of RPLs in [22] revealed that most of them were extensions to Java. Accordingly, five of them were implemented as a Neverlang-based LPL [22], whereas each was implemented as an extension to a Neverlang-based Java parser, denoted Neverlang.Java.

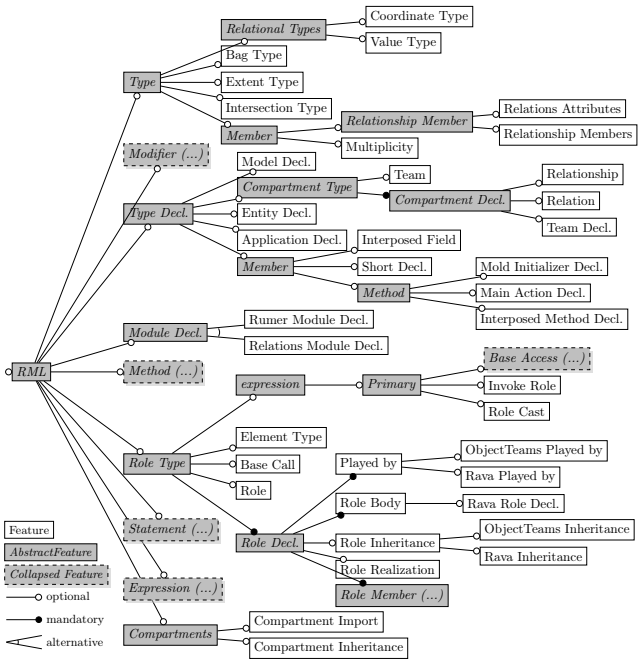

Figure 3: Generated feature model of role-based programming languages, from [22].

Feature Model and Language Decomposition. The resulting family of RPLs follows a bottom-up LPL approach [22], which entails that the feature model, partially shown in Fig. 3, was generated by AiDE. The resulting feature model is rather big consisting of 73 features and 29 abstract features [22]. For brevity, several abstract features, e.g., *Modifier*, *Method*, and *Statement*, have been collapsed and all cross-tree constraints were omitted.

Thus, the resulting feature model only emphasizes language features of RPLs and not the underlying Java LPL.[4] Regardless, AiDE could still be employed to select valid language variants [22]. However, these language variants were hard to use, because different RPLs introduced different syntax with the same intentional semantics. Let us consider the various keywords to declare *roles*, e.g., **`role`**, **`definerole`**, **`participants`**, **`class`**. While a context-agnostic syntax highlighter could be defined, the user would still loose track of which language feature belongs to which language extension.

Syntax Highlighting for Individual Language Features. To approach this issue, syntax highlighting editors of state-of-the-art workbenches would not suffice. By contrast, our context-aware syntax highlighting editor for LPLs provides provenance for language features by highlighting related language features in the same style. In case of the family of RPLs, we added categories to all Neverlang slices in the LPL and included six style files within the LPL. One file defines the style for Neverlang.Java and the others define individual styles for each of the five role-oriented language extensions, such that each language feature provided by an extension uses a distinguishing syntax highlighting. Specifically, Rava keywords have

[4]The full feature model for RPLs is available at `http://neverlang.di.unimi.it/aide/rplj_fm.pdf`.

Table 1: Specifying syntax highlighting for Java and five language extensions, from [22].

Language	Slices	Classes	LoC	Highlighting
Java	189	2	6843 (323)	208 (30)
Common	6	3	184 (850)	4
Rava	5	0	213	17 (12)
powerJava	7	0	243	14 (15)
OT/J	16	0	715	43 (15)
Rumer	31	5	1238 (146)	75 (16)
Relations	20	1	756 (33)	32 (16)

a lime background, powerJava violet, ObjectTeams/Java orange, Rumer cyan, and Relations blue. As shown in Table 1, piggybacking syntax highlighting on the family of RPLs required only 393 additional *lines of code* (LoC) and 74 lines in JSON style files (numbers in brackets). Naturally, we defined the nine categories of terminals according to the typical lexical tokenization of Java, i.e., `Identifier`, `Type`, `Operator`, `Brackets`, `Keyword`, `String`, `Number`, `Boolean`, and `Character`. However, most of the role-oriented extensions only override the styling of keywords, brackets, and operators, with the exception of Rumer that introduces the new type `Extent`. In sum, the manual implementation for adding syntax highlighting to the existing LPL for RPLs was limited and could technically be completely replaced by a code generator.

Context-aware Syntax Highlighting Editor. To test the LPL for RPLs, we generated multiple language variants ranging from the five RPL languages to a feature complete variant. The latter, allows for combining the various role declarations in one combined language. This language variant is used to showcase the contextual awareness of our syntax highlighting editor. After announcing this language variant to the NeverlangIDE plugin, we can open an `example.rolejava` file in Eclipse with the `Neverlang Editor`. In fact, right clicking on the file in Eclipse and selecting the `Open With... ▸ Other` context menu item, opens a dialog where our `Neverlang Editor` can be selected. Fig. 4 depicts a screenshot of the editor showing the content of the `example.rolejava` file. This file contains a role-based banking application implemented using role definitions (language features) provided by the different extensions. In contrast to typical syntax highlighting, for the first time, users are able to track the provenance of employed language features by means of a simple color coding. This helps to notice errors that would be otherwise missed. For instance, it becomes evident that a Rava role call (**`@INVOKEROLE`**) is used within a powerJava role (Lines 27–33). Thus, while users get insight into the provenance of employed language features, they are still able to customize their highlighting by modifying the style files.

5.2 Family of JavaScript-based Languages

The family of JavaScript-based languages, denoted Neverlang.JS, was initially designed as a real world case study for Neverlang [32], yet it has proven its worth for *gradually* teaching JavaScript [6]. In detail, students were provided with specialized JavaScript variants, whereas each variant focuses on teaching another language feature, e.g., loops, recursion, exception handling, object orientation [6].

example.rolejava

```
/* Pure Java */
class Account{
  void increase(double amount){/*...*/}
  void decrease(double amount){/*...*/}
}
class Person {/*...*/}

/* powerJava role interface */
role Customer playedby Person{/*...*/}
/* ObjectTeams context */
team class Bank {
  /* ObjectTeams role */
  class Checking playedBy Account {
    double limit = 500.0;
    callin void decrease(Double amount){
      if (amount<=limit) base.decrease(amount);
    }
  }
  /* Rava role */
  ROLE Savings roleof Account {
    double fee = 1.05;
    void decrease(double a) {
      @core Account().decrease(a * fee);
    }
  }
  /* powerJava role */
  definerole BankCustomer realizes Customer {
    boolean transfer(Account f, Account t,double a) {
        /* Rava invocation */
        if (f.plays(Savings))
            @INVOKEROLE(f, Account, Savings, decrease, a);
        if (f.plays(Checking))
            @INVOKEROLE(f, Account, Checking, decrease, a);
        t.increase(a);
    }
  }
}
/* Rumer relationship */
relationship Ownership participants(Person owner, Account owned) {
  extent boolean addAccountunt(Person owner, Account owned) {
    return these.add(new Ownership(owner,owned));
  }
}
```

Figure 4: Contextual highlighting of role-oriented extensions to Neverlang.Java.

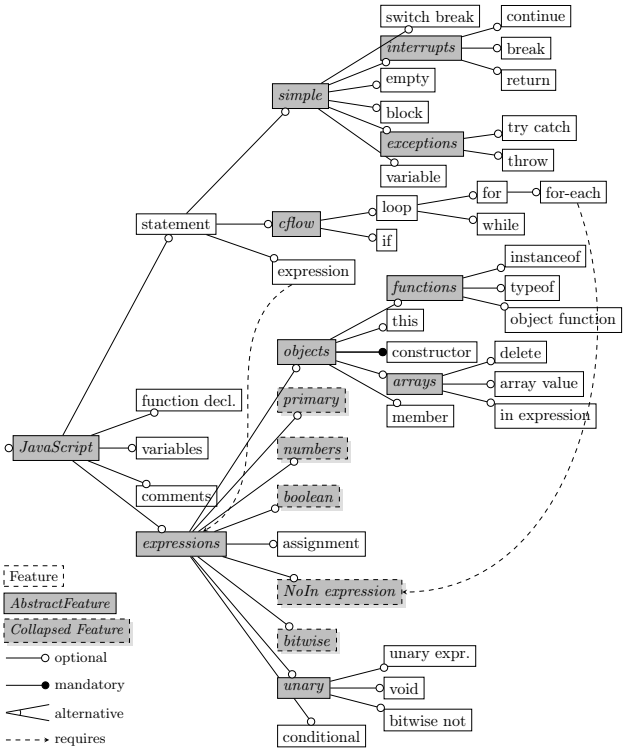

Figure 5: Generated feature model of Neverlang.JS, from [23].

Although our experience showed the viability of this approach, we concede that our students struggled to find errors in their implementation, especially, in case of runtime errors. What our students missed most, was debugging support for the various language variants to easily set breakpoints and step through their running program. Conversely, this case study finally introduces debugging support to Neverlang.JS and all its variants. Moreover, the editing, the executing, and the debugging of JavaScript language variants is integrated into the Eclipse IDE.

Feature Model and Language Decomposition. Neverlang.JS is a fully decomposed version of JavaScript, whereas each `module` and `slice` corresponds to a specific language feature. Its implementation amounts to 3043 LoC and 228 production rules [32]. Each valid language variant is a functional JavaScript interpreter. In particular, the feature complete Neverlang.JS variant conforms to the ECMAScript 3 Language Specification (ECMA-262) and covers about 70% of the corresponding language specification [32]. Notably though, the remaining 30% amount to implementing built-in libraries, which is merely a technicality and is not required to showcase the debugging support.

Besides that, the Neverlang.JS LPL was one of the first LPLs to be configured by AiDE. Resulting from JavaScript's complexity, AiDE generated a very large feature model with a maximum depth of 6 as well as 19 abstract features and 73 language features [23].

Due to space restrictions, Fig. 5 shows a reduced feature model, where the abstract features *primary*, *numbers*, *boolean*, *NoIn expressions*, and *bitwise* have been collapsed.[5] In addition, the feature model captures that *constructors* are mandatory for *objects* and *for-each* loops depend on *NoIn expressions*, which define the `in` expression only inside *for-each* loops. However, while the Neverlang.JS LPL produces an interpreter for each member of the family of JavaScript-based languages [6], these interpreters lack direct debugging support.

Adding Debugging to Neverlang.JS. As outlined in Sect. 3.3 and Sect. 4.3, adding debugging support to an existing LPL includes two major steps: (1) marking execution steps and (2) exposing variables and their values in the current execution context. To improve usability, we kept debugging on the level of statements rather than expressions. In fact, most debuggers operate on this granularity, as stepping through expressions would be tedious.

Consequently, in the first step, we extended all Neverlang slices that represent statements with a `debug` role with a corresponding action setting `isExecutionStep` to **`true`**. As depicted in Table 2, this includes statements, such as **`if`**, **`switch`**, **`for`**, `for-each`, as well as function calls, interrupts and conditional expressions, with the exception of the general *Block* and *Loop* statement. Besides marking execution steps for the debugging agent, the second step entails writing two classes implementing the `VariableInfo` and `ValueInfo` interface to expose variables and their values, respectively.

[5] A full version is available at `http://neverlang.di.unimi.it/aide/njs_graph.png`.

Table 2: Introducing debugging to the JavaScript-based language family, from [23].

Features	Slices	LoC	Debugging
Core			
Language core	11	277	10
Expressions			
Arithmetic	3	128	-
Boolean	3	92	-
Relational	2	137	-
Conditional	1	32	7
Bitwise	5	216	-
Typing	2	65	-
Function call	2	113	12
Construct call	1	56	-
Types			**43**
String	1	21	-
Number	1	24	-
Boolean	1	23	-
RegExp	1	23	-
Object	4	189	30
Array	3	131	-
Function (definition)	2	100	-
This resolution	1	17	-
Statements			
Block Statement	1	32	-
If Statement	1	45	8
Switch Statement	1	102	5
(Loop Statements)	1	19	-
While Statement	1	50	8
For loop	1	57	7
For-each loop	1	113	13
(NoIn expressions integration)	11	305	16
Interrupts (break,continue,...)	3	74	15
Exception handling	2	122	5
Variables			
Variable assignment	5	226	11
Variable resolution	1	24	-
Symbol Table	0	230	8

In Neverlang.JS these classes are implemented as `JSVariableInfo` and `JSValueInfo` and amount to 43 LoC (cf. *Types*). Note, these `VariableInfos` must be programmatically added to Neverlang's `VariableInfoManager` during the program's interpretation, e.g., whenever a variable is declared. Only then, the `DebugAgent` can access the variables visible in the current execution context, and finally expose them to Eclipse's `Variables` view. Consequently, this has been done for the language features *assignments*, *function calls*, and *objects* leading to an additional 53 LoC. Although, the first step can be automated by annotating production rules, the second step requires manual work to expose the interpreters internal data structures, such as, variables, objects, and arrays. In sum, the implementation overhead to add debugging support to the Neverlang.JS LPL only amounts to 198 LoC, which is surprisingly small considering the benefit it provides to its users.

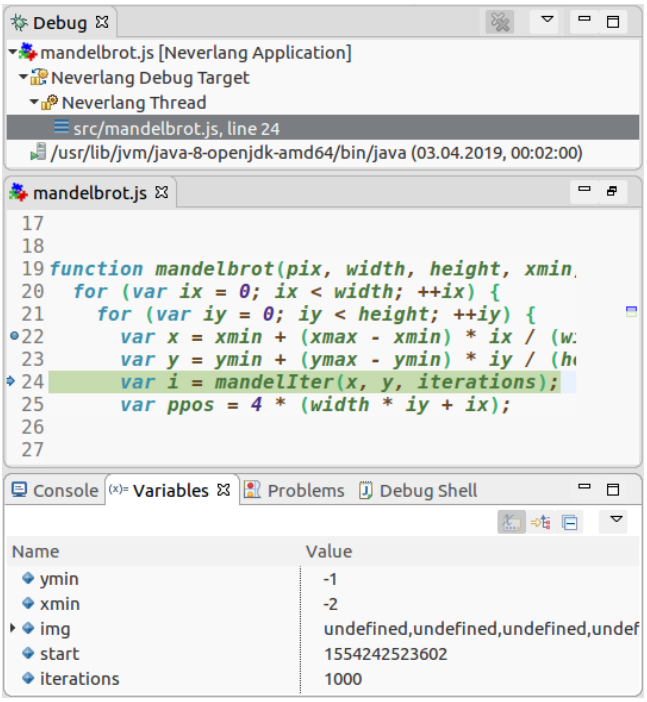

Figure 6: Debugging a variant of Javascript in Eclipse.

Debugging JavaScript-based Language Variants. By employing AiDE we have created and tested multiple specializations of JavaScript as well as a feature complete variant. Yet, the latter will be used henceforth to demonstrate our debugger. Just like a language variant with piggybacked syntax highlighting, a language variant with piggybacked debugging support must only be announced to the NeverlangIDE plugin. Then once Eclipse has started, users can create, open, and edit JavaScript files (`*.js`). For simplicity, we created a `mandelbrot.js` file, which encompasses a simple JavaScript program that computes the Mandelbrot set, inspired by [5, Listing 7]. Now, we can execute this program by creating a new Neverlang *debug configuration* via the **Run ➥ Debug Configuration...** menu item; selecting `neverlang.js.JSLang` from the set of announced language variants and the executable `mandelbrot.js`; and finally clicking on the `Debug` button. This will trigger the LPL-driven debugger by providing it with the language variant as well as the `mandelbrot.js`. Afterwards, Eclipse sets up a TCP connection to the debugger, which permits to directly interact with the debugger through Eclipse's debug interface. One such run is shown in Fig. 6, where a breakpoint was set in Line 22 via the context menu entry `ToggleBreakpoint`. This, in turn, suspended the debugger in Line 22, where a user clicked the *step over* button twice, such that the current location of the program is in Line 24. From this location, a user can either *step into* the implementation of the `mandelIter` or *step over* its execution such that the debugger reaches Line 25. In addition to the editor, the *Debug* view indicates both the current state of the debugger and the current location in the source code. Besides that, the *Variables* view (below) lists all variables in the scope of the current location and their corresponding values, whenever the debugger has been suspended or performed a step.

Finally, once the debugging is completed the debugger can be either terminated or resumed until it reaches another breakpoint. In conclusion, it is not just feasible to develop a Neverlang-based LPL with piggybacked debugging support, but the latter also provides similar usability as the standard Java debugger.

5.3 Discussion

In summary, the two presented demonstration studies provided evidence for the feasibility of piggybacking both syntactic and semantic IDE services on language components. Granted, one might argue that syntax highlighting is the simplest syntactic service, yet, our context-aware syntax highlighting editor exceeds editors generated by state-of-the-art language workbenches, especially as it can provide provenance of language features. Moreover, our demonstration study indicated that the implementation overhead for piggybacking syntax highlighting on Neverlang slices is negligible. Finally, as Neverlang's slices are typically tied to a `reference syntax`, other syntactic IDE services could be piggybacked in a similar way. That is, a minimal service specification could be added to slices, which is then retrieved and drives the behavior of a generic Eclipse-based View/Editor. Although no one will argue that *debugging* is a simple semantic IDE service, a similar argument can be drawn for other semantic IDE services. In fact, due to Neverlang's support for *open programming language interpreters* [7], other language agents could be attached to the interpreter, which can expose internal information required for a specific semantic IDE service. This information can then be used by an extended editor to provide the desired semantic IDE service. In conclusion, most of the effort will be put into providing an LPL-driven IDE service for a particular platform, e.g., Eclipse, whereas the implementation overhead for a language engineer will be limited to marking special syntactical elements or exposing semantic information via predefined interfaces.

6 RELATED WORK

Nowadays, the development of DSLs is a hot topic and a lot of research efforts are spent in this direction. Several language workbenches have been developed, such as Spoofax [38], MPS [35], MontiCore [21] and Melange [12]. All of these approaches provide a way to generate the IDE support for the DSL under development. In all these cases, the IDE support is tied to the developed DSL but its support is general and it does not exploit specific characteristics of the DSL. The IDE is automatically generated from templates neglecting the DSLs feature modularity, such that IDE services cannot be specified within DSL and reused from time to time as in this proposal. MontiCore [2, 3] and Melange [27] support, the development of LPLs but their approaches only support basic syntax highlighting and code completion.

Besides that, EMFText [18] must also be highlighted, as another EMF-based tool [30] (like Xtext) that supports modular language implementation and explicitly supports the IDE generation for the developed languages. Even if EMFText does not explicitly consider variability in the IDE development, it has several commonalities with our LPL-driven IDE starting from the use of attributed grammars for sharing information between languages and IDE implementation but also because of a specific DSL dedicated to the description of the IDE.

Looking at the current literature in software variability, some efforts have been made towards the description of families of graphical (modeling) editors, yet not for IDEs. Several tools of this kind appeared over the years, e.g., EuGENia [20], Graphiti [36] and Sirius [37]. EuGENia and Sirius are model-based, while Graphiti relies on Java code. Both EuGENia and Graphiti are based on code generation, whereas Sirius is dynamically interpreted. In general, all these approaches lack direct support for variability provided by SPL methodologies. Thus, variability must be hard coded leading to hard to maintain and hard to extend product lines.

In this respect, our latest contribution, FRaMED [24] properly supports variability in the development of graphical editors but not IDE. It focuses on the visualization/modeling of the program rather than providing support for its debugging or other typical needs of code-based development. FRaMED follows a top-down approach to the LPL construction whereas our LPLs follow a bottom-up approach; therefore the editor is generated through model composition instead of language component composition.

7 CONCLUSION

In this paper, we presented the idea of piggybacking portions of IDE services to the corresponding language component, enabling *de facto* the possibility of dealing with language variability *together* with IDE variability. Evidently, a language and its IDE are deeply interconnected and keeping their development separated violates feature modularity limiting the support an IDE could provide.

Conversely, our contributions are manifold. We introduced the concept of IDE services and how these can be piggybacked on language components. We explored the various kinds—syntactic and semantic services—of support an IDE provides and explained how these can be bound to the developed language variant through a LPL. Although the concept is generally applicable to all syntax-directed language workbenches, we have extended the Neverlang language workbench and applied it to families of real languages such as JavaScript and the Java-based role-oriented languages. The demonstration cases showed the feasibility of our approach, in general, as well as nice side-effects of context-aware syntax highlighting, such as tracing the provenance of language features.

Arguably, this work presented a proof-of-concept—even if it can work on real cases—so there is still space for improvement. In the future, we will integrate more syntactic and semantic IDE services, e.g., error marking, and semantic auto-completion. Moreover, we intend to permit dynamically reconfiguring the language variant within the IDE, such that both debugging and syntax-highlighting are dynamically adapted. Furthermore, we consider to support the *language server protocol* and widen the support for IDEs beyond Eclipse. Finally, we hope that other syntax-directed language development tools apply our idea.

REFERENCES

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

[2] 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.

[3] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*, Thorsten Berger and Paulo Borba (Eds.). ACM, Gothenburg, Sweden, 65–75.

[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, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. 2018. μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures* 51 (Jan. 2018), 71–89. https://doi.org/10.1016/j.cl.2017.07.003

[6] 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.

[7] Walter Cazzola and Albert Shaqiri. 2017. Open Programming Language Interpreters. *The Art, Science, and Engineering of Programming Journal* 1, 2 (April 2017), 5–1–5–34. https://doi.org/10.22152/programming-journal.org/2017/1/5

[8] 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.

[9] Walter Cazzola and Edoardo Vacchi. 2014. On the Incremental Growth and Shrinkage of LR Goto-Graphs. *Acta Informatica* 51, 7 (Oct. 2014), 419–447. https://doi.org/10.1007/s00236-014-0201-2

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

[11] 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) (LNCS 3713)*, Lionel Briand and Clay Williams (Eds.). Springer, 97–112.

[12] 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.

[13] 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 Sloane and Suzana Andova (Eds.). ACM, Tallinn, Estonia.

[14] 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.

[15] 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.

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

[17] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, Klaus Pohl and Birgit Geppert (Eds.). IEEE, Limerick, Ireland, 139–148.

[18] 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) (LNCS 7680)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, Braga, Portugal, 322–345.

[19] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The Road to Feature Modularity?. In *Proceedings of the 15th Internation Software Product Line Conference (SPLC'11)*, Ina Schaefer, Isabel John, and Klaus Schmid (Eds.). ACM, Münich, Germany.

[20] Domitrios Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. 2017. EuGENia: Towards Disciplined and Automated Development of GMF-Based Graphical Model Editors. *Software System and Modeling* 16, 1 (Feb. 2017), 229–255.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, and Benoît Baudry. 2017. Reverse Engineering Language Product Lines from Existing DSL Variants. *Journal of Systems and Software* 133 (Nov. 2017), 145–158.

[28] 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.

[29] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data and Knowledge Engineering* 35, 1 (Oct. 2000), 83–106.

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

[31] 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.

[32] 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

[33] 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.

[34] 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.

[35] 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.

[36] Vladimir Vujović, Mirjana Maksimović, and Branko Perišić. 2014. Comparative Analysis of DSM Graphical Editor Frameworks: Graphiti vs. Sirius. In *Proceedings of the 23rd International Electrotechnical and Computer Science Conference (ERK'14)*. Portorož, Slovenia, 7–10.

[37] Vladimir Vujović, Mirjana Maksimović, and Branko Perišić. 2014. Sirius: A Rapid Development of DSM Graphical Editor. In *Proceedings of the IEEE 18th International Conference on Intelligent Engineering Systems (INES'14)*, Tamás Haidegger and Levente Kovács (Eds.). IEEE, Budapest, Hungary.

[38] 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.

[39] Christian Wende, Nils Thieme, and Steffen Zschaler. 2009. A Role-Based Approach towards Modular Language Engineering. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09) (Lecture Notes in Computer Science 5969)*, Mark van den Brand, Dragan Gasevic, and Jeff Gray (Eds.). Springer, Denver, CO, USA, 254–273.

[40] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt. 2009. Improving Domain-specific Language Reuse with Software Product-Line Configuration Techniques. *IEEE Software* 26, 4 (July-Aug. 2009), 47–53.

A ARTIFACT

In addition to this publication, we provide an artifact to reproduce our results. In particular, we published a virtual machine packed with the ready to use syntax highlighting editor with debugging support, denoted NeverlangIDE. In detail, it is an Eclipse plugin that includes a syntax highlighting editor and debugger for two *language product lines* (LPL) with piggybacked basic IDE services, i.e., where modular language features include the definition for syntax highlighting and debugging.

A.1 Installation and First Step

We provide our LPL-driven IDE as a *virtual machine* (VM), because it is supposed to work in 10 years from now. It was prepared for the virtualization environment VirtualBox and is installed, as follows:

- Download and install `VirtualBox` from their website.[6]
- Download the NeverlangIDE image (**~1.3 GB**).[7]
- Open your `VirtualBox` and use **File ➥ Import Appliance** to select and import the downloaded file.
- Start the added NeverlangIDE virtual machine.

After the virtual machine is launched, double click on the NeverlangIDE icon on the desktop to start Eclipse with the NeverlangIDE plugin. You can use it to inspect the already opened JavaScript and RoleJava (`*.rolejava`) files or debug the *Mandelbrot* script (`mandelbrot.js`). Because the configuration and generation of a language variant is too complicated to be explained in one page, we opted to provide you with two products of such LPLs showcasing syntax highlighting and debugging, respectively. Henceforth, we focus on these two usage scenarios.

A.2 Debugging the JavaScript LPL

Once you have opened the editor `mandelbrot.js`, you can inspect the script computing the *Mandelbrot* set. It already contains *breakpoints* at Line 22 and 60, however, you can add and remove breakpoints by right-clicking on the line number and selecting *Toggle Breakpoint*. Currently, the `Debug` view and the `Variables` view are empty, they will be populated automatically during debugging.

(1) Click on the `Debug` icon (alternatively, use **Run ➥ Debug**) to start debugging.
(2) After a short while the `Debug` view is populated, with among others a "*Neverlang Debug Target*". Click on the triangles to unfold the "*Neverlang Thread*" and finally "*src/mandelbrot.js, line 60*".
(3) Click on "*src/mandelbrot.js, line 60*" to update the editor and populate the `Variables` view. **Note, that otherwise Step-Into and Step-Over will not affect the editor and the view!**
(4) Use the *Step-Into* icon until you jumped into the `mandelbrot` function (Line 20).
(5) Use *Step-Over* or *Step-Into* as you like to proceed with the stepwise execution.
(6) Continue the execution by clicking on the `Resume` icon, which halts the execution again on Line 22.
(7) To complete the execution simple right-click on breakpoints (Line 22) and click on *Toggle Breakpoint*. Afterwards, click on `Resume` to complete the execution. (Alternatively, you can always click on `Terminate` to kill the current debug session.)

A complete execution will emit the computed mandelbrot set as array and the required time on the `Console` view. A complete run inside the debugger will take a long time.

A.3 Context-Aware Syntax Highlightin

To illustrate the context-aware syntax highlighting, five files of the family of Java-based role-oriented programming languages have been included; four of these are already open when Eclipse starts. The file `example.rolejava` showcases context-aware syntax highlighting in action—i.e., it shows some language features from different languages (selected through the LPL) living together with their original syntax highlighting. Whereas the remaining `*.rolejava` files feature each different role-based programming language, e.g., Rava, PowerJava, ObjectTeams, from which the language features are selected. **Because the NeverlangIDE is running inside a VM and utilizes dynamic class loading, opening an editor takes a long time.**

As a usage scenario, we suggest you try copying `role` or `team` definitions from the various role-based programming languages into the combined `example.rolejava` file.

(1) Open the `example.rolejava` editor and browse through the source code. Note that some keywords, e.g., `class`, are highlighted differently depending on their position.
(2) Open the `rava.rolejava` editor (be patient). Select the role definition `Checking` (Lines 19–28) and copy it into Line 7 of the `example.rolejava` file. As a result, highlighting is retained, i.e., the keywords of Rava are highlighted in green.
(3) Open the `powerjava.rolejava` editor (be patient). Select the role definition `CA` at Line 8 and copy it into Line 18 of the `example.rolejava` file. Note that this definition can only occur as innerclass, hence including it outside of the `Transaction` team, would result in a syntax error (in turn, disabling context-aware syntax highlighting).
(4) Open the other editors by clicking on `»1` and selecting the corresponding file. Try copying other roles, teams or methods to the `example.rolejava` and explore the effects.

Be aware that once the program contains a syntax error, the context-aware syntax highlighting falls back to the default highlighting of Java. In that case use `Undo` to revert the file to the original state.

If you accidentally close one of the files, they can be opened through the **Project Explorer**. Just, open then "`RoleJava`" project and go to the "`src`" folder. It contains all `*.rolejava` files, which can be opened by double clicking on them (be patient).

A.4 More Information

Inside the virtual machine the folder `Artefact/Languages/` contains the two LPLs, there you can inspect the implementation, whereas the Neverlang files are gathered in the `nlg-src/` folder and auxiliary Java classes in the `src/` folder. More information on Neverlang, its development and its use can be found on its website.[8]

[6] https://www.virtualbox.org/
[7] https://adapt-lab.di.unimi.it/NeverlangIDE.ova
[8] https://neverlang2.di.unimi.it