A Low-Code Platform for Systematic Component-Oriented Language Composition[*]

Jérôme Pfeiffer
jerome.pfeiffer@isw.uni-stuttgart.de
University of Stuttgart
Germany

Andreas Wortmann
andreas.wortmann@isw.uni-stuttgart.de
University of Stuttgart
Germany

Abstract

Low-code platforms have gained popularity for accelerating complex software engineering tasks through visual interfaces and pre-built components. Software language engineering, specifically language composition, is such a complex task requiring expertise in composition mechanisms and language workbenches including multi-dimensional language constituents (syntax and semantics). This paper presents an extensible low-code platform with a graphical web-based interface for language composition. It enables composition by using language components, facilitating systematic composition within language families promoting reuse and streamlining the management, composition, and derivation of domain-specific languages.

CCS Concepts: • **Software and its engineering → Reusability**; **Software notations and tools**.

Keywords: Software language engineering, Language composition, Language components, Low-code

ACM Reference Format:
Jérôme Pfeiffer and Andreas Wortmann. 2023. A Low-Code Platform for Systematic Component-Oriented Language Composition. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623476.3623516

1 Introduction and Motivation

In recent years, the demand for efficient software development processes has led to the emergence and widespread adoption of low-code development platforms [1]. These platforms provide visual interfaces and pre-built components that accelerate the development of complex applications [10]. Among the intricate aspects of software development, software language engineering and composition play a vital role in achieving effective and customizable solutions, e.g., in the domain of digital twins [4]. Software language engineering involves designing and implementing domain-specific languages, while language composition combines languages. Understanding composition mechanisms and language workbenches is essential [7]. Although web-based language workbenches exist [11], limited reuse for multi-dimensional languages remains a challenge. To address this, we introduce a low-code platform based on a method for black-box language composition using language components that encompass syntax and semantics [3]. We refer to this method for systematic component-oriented language reuse as SCOLaR in the following. The platform allows language engineers to derive language components from existing projects and systematically compose them within a language family. By selecting and combining language features within the platform's language family, language engineers can create tailored languages that cater to specific application domains. Through this research, we aim to contribute to the advancement of low-code development methodologies and empower language engineers with a powerful tool to create and reuse languages in a more intuitive and efficient manner.

2 Systematic Component-Oriented Language Reuse

The low-code platform is grounded in the concepts of SCOLaR that uses language components encompassing the constituents of language definitions in the language workbench MontiCore [6]. Language components can be reused by their interface in language families.

2.1 Language components

Language components [3] provide the three essential language definition constituents: (1) syntax, (2) well-formedness rules, and (3) code generators, realizing the semantic mapping between the problem and the solution domain, that are exposed by extensions in their interface. SCOLaR differentiates between required and provided extensions. Provided

[*]The authors were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) [grant number 441207927].

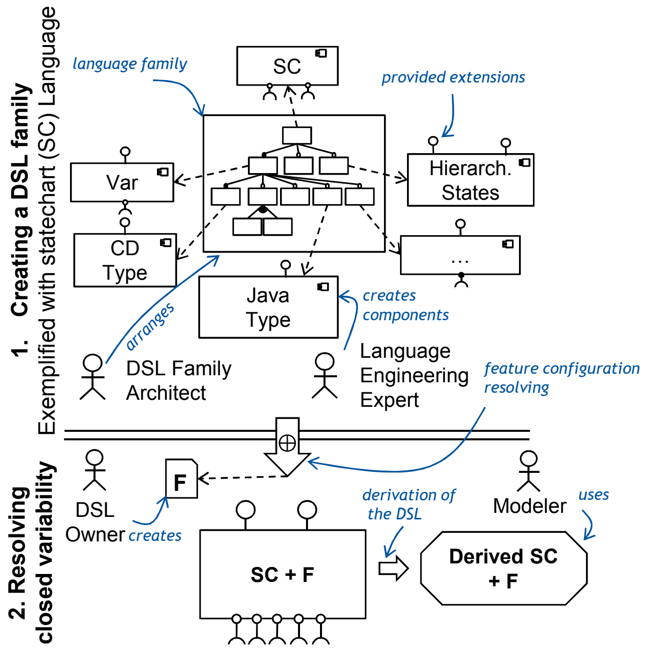


Figure 1. The systematic language composition process exemplified with a language family for statecharts.

extensions offer DSL functionality to be reused by other components. Required extensions make missing functionality of a DSL component explicit and can be either optional or mandatory. Provided and required extensions can reference productions of the grammar or a generator for a specific grammar production. Well-formedness rules are contained in sets that can act as both provided and required extensions at the same time.

2.2 Systematic Composition with Language Families

Language family architects arrange language components into a feature model representing a family of DSLs (cf. Figure 1). In this feature model, each feature either is related to a language component or is an abstract feature [9] for logical grouping. Through this relation, the language family architect decides how the components will be composed when their related features are selected. Once the language family architect completes the language family, DSL owners, who are experts of the application domains derive a suitable DSL for their application domain, by selecting appropriate features from the family in a feature configuration. The composition of two DSL components is the directed application of bindings between these components. Currently, SCOLaR supports the composition operators embedding and aggregation [8]. For the composition the provided extensions of one component are bound to required extensions of another component. The composition includes two main activities: (1) Composing the components' interfaces; and (2) Composition of the comprised language definition constituents (grammars, well-formedness rules, code generators).

3 The Low-Code Platform for Language Composition

The SCOLaR low-code platform is designed around the SCOLaR process (cf. Figure 1) and provides a graphical web-based environment to support it. This section outlines the essential requirements that the platform must meet and then delves into the workflow of the SCOLaR low-code platform.

3.1 Requirements

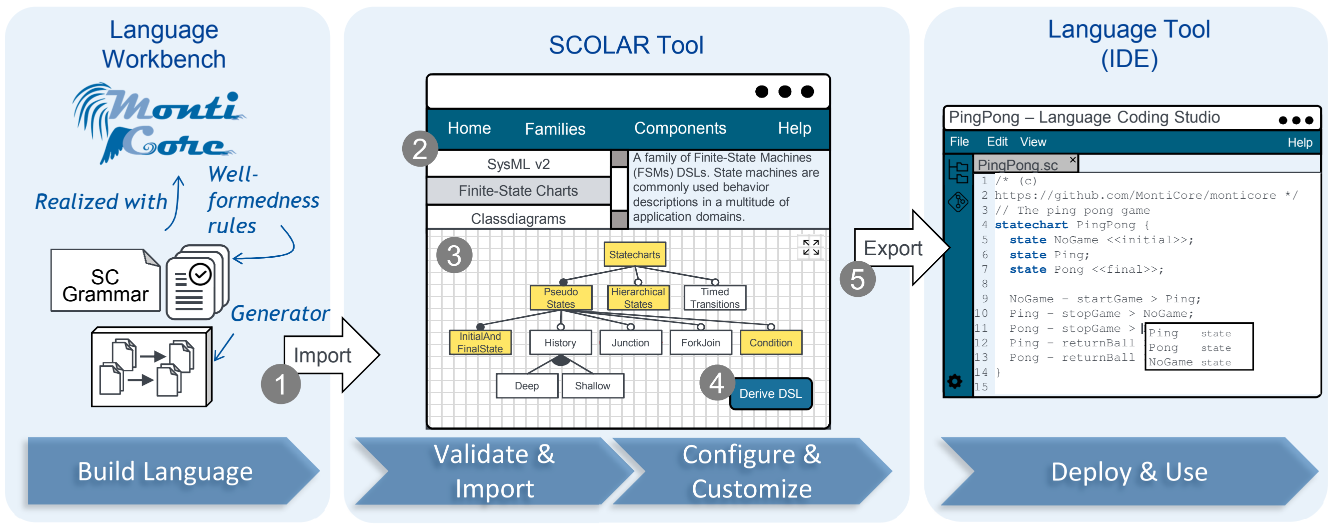
To transform the SCOLaR method from a conceptual method into a user-friendly low-code platform accessible to language architects, we have created an enhanced version of the SCOLaR process, illustrated in Figure 2. The numbers in Figure 2 refer to the requirements we derived for the low-code platform:

Req 1: Language engineers can continue developing languages in their language workbench together with the associated technology-specific artifacts. Hence, existing language projects can be imported and a language component representation is derived automatically.
Req 2: Imported language projects and created language families should be persisted, e.g., in a database.
Req 3: The low-code platform enables DSL Family Architects to create language families.
Req 4: DSL owners can configure existing language families and derive new languages by composition.
Req 5: The composed languages should be exportable for deployment.

3.2 Workflow of the Low-Code Platform

The workflow of the SCOLaR low-code platform can be divided into three steps (cf. Figure 2). Firstly, languages are constructed within a language workbench and subsequently imported into the platform. However, the only language workbench that is supported to this date is MontiCore [6]. Once imported, these languages can be reused within a language family. The configuration of the language family leads to the creation of a new language component, which includes a composed language project that can be downloaded.

3.2.1 Automatic Derivation of DSL components. Creating a language component model is tedious and error-prone. However, in pursuit of our low-code platform's objective to empower users to compose languages with minimal manual coding, we have developed a derivation function that automatically generates language components from existing MontiCore language projects. After the automatic derivation of a language component, the low-code platform enables the customization of this component to restrict the provided extensions for generator and grammar productions. After the



A family of Finite-State Machines (FSMs) DSLs. State machines are commonly used behavior descriptions in a multitude of application domains.

```
/* (c)
https://github.com/MontiCore/monticore */
// The ping pong game
statechart PingPong {
  state NoGame <<initial>>;
  state Ping;
  state Pong <<final>>;

  NoGame - startGame > Ping;
  Ping - stopGame > NoGame;
  Pong - stopGame > |Ping    state
  Ping - returnBall  Pong    state
  Pong - returnBall  NoGame  state
}
```

Figure 2. The systematic language composition process including the development of a language in a language workbench, the import into the SCOLaR tool, the selection of language features, and the derivation of the configured language variant.

import and customization step, the components are ready to be reused in language families. The following describes how the automatic derivation of different language component constituents is realized.

Grammar. The grammar reference is derived directly from the grammar name and its package specified in the project's grammar directory. Within this grammar, we identify that each production rule with a right-hand side serves as a provided grammar extension. Each production rule with the keyword `interface`, indicating that implementation is open for extension, becomes a required extension in the language component.

Well-formedness rule sets. The well-formedness rule sets are linked to particular grammar production rules. MontiCore provides infrastructure for validating model well-formedness, including checkers that allow developers to register specific well-formedness rules. In the language component, a set is defined for each checker, which precisely matches the rules registered within the corresponding checker.

Generator. In order to derive generator extensions, it is expected that generators are constructed in accordance with the concept described in [2]. This entails that generators should have explicit product and producer interfaces.

3.2.2 Creating a Language Family.

The low-code platform offers a dedicated language family workbench for composing language components. Within this workbench, users have the option to start with a blank canvas or modify an existing language family. Using a side menu, the user can add or remove features of the language family. Each feature is characterized by a name and a type, i.e., abstract or normal feature. Abstract features are used for grouping. Normal features make references to language components available in the platform's language component library. Connections between features are established and defined with types such as or, xor, and, optional, or mandatory. These connections establish bindings between the extensions of the referenced language components, linking child features to parent features. The resulting composition tree can be saved to the language family library and utilized for deriving composed languages through feature configuration in subsequent steps.

3.2.3 Configuring and Exporting Composed Languages.

To derive new languages, the initial step involves selecting a language family from the language family library. Each language family within the library comes with a comprehensive description and is visually represented as a tree structure. The user can interact with this tree by clicking on specific language features to select them. Once the desired language family configuration is established, the user can click on the "Derive DSL" button. This action triggers a validation process where the language family configuration is checked against the constraints of the feature tree. If the configuration is deemed valid, the SCOLaR framework in the backend proceeds to compose the referenced language components. Once the composition process is completed, the user is notified through a pop-up message and provided with the option to download the composed language project source files. To utilize the language, the project can be built using Maven and subsequently used with the tooling provided by the MontiCore language workbench.

4 Software Architecture

The SCOLaR low-code platform is realized as a classic three-tier architecture consisting of a persistency layer, a backend and a frontend. The software architecture is depicted in Figure 3. Our low-code platform is designed with deployability in mind, and every component of the software architecture is packaged as a Docker image. This Dockerization enables seamless and scalable deployment of our platform, allowing for efficient utilization of resources and easy management of the platform's infrastructure.

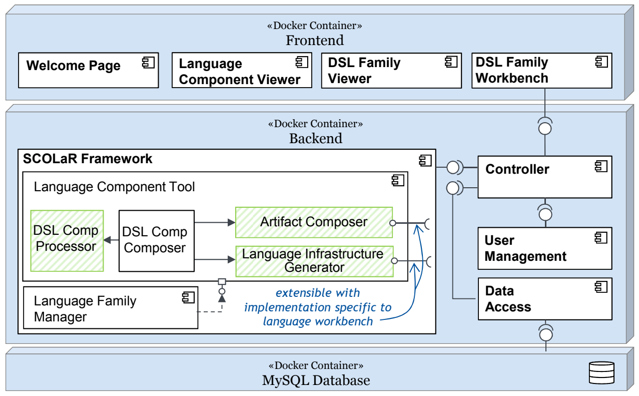


Figure 3. The 3-tier architecture of the low-code platform.

4.1 Frontend

The frontend of the SCOLaR low-code platform is developed using Vue.js and provides different components for user interaction. The `Welcome Page` enables to log into the platform and choose from the `Language Component Viewer`, the `Language Family Viewer`, and the `Language Family Workbench`, afterward. Utilizing the `Language Family Workbench` users can assemble new language families or reuse and extend existing ones following the process described in Section 3.2. With the `Language Component Viewer` language projects can be managed and imported into the platform. The `Language Family Viewer` shows available language families for configuration.

4.2 Backend

The backend of the platform is built on Java Spring Boot. The backend comprises a `Controller` for the overall workflow control of the platform, and to provide the REST API for interaction with the frontend. Additionally, the backend includes a `User Management`. Today, the SCOLaR platform supports the roles 1) engineer, that has rights to create, remove and modify language families and components, 2) and user, that can only view components, and families and configure and derive language products by selecting features of a language family already available in the language family library. For persisting imported language projects together with their associated language components, as well as created language families, the `Data Access` component persists them into a MySQL database. In addition, the existing tooling of SCOLaR is reused in the backend, to perform the processing of language component models, language families, language family configurations, and the composition of the language components and their comprised artifacts. Furthermore, there exists a language infrastructure generator that generates the composed language project and exports it as a zip file.

4.3 Database

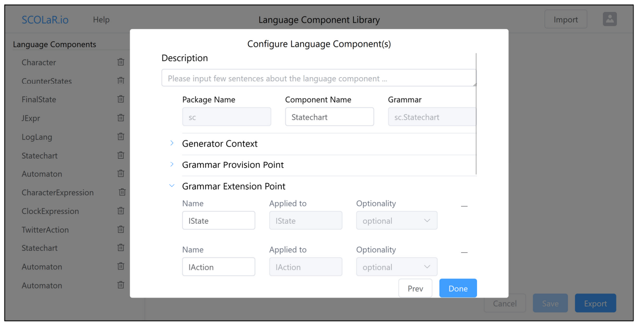
The persistency layer of the platform is implemented using MySQL. The database of the platform is used to persist all artifacts related to the SCOLaR process (cf. Figure 1), i.e., language components, their related language projects, the language family model, and users, together with roles and the associated permissions. This enables the platform to provide a library of language components and families for reuse that were imported and assembled before.

4.4 Extensibility

Since SCOLaR is subject to ongoing research, the software architecture should be refined accordingly. We see the following concepts being subject to changes that have to be taken into account in all three layers of our architecture. 1. The extension of constituents of language components and their composition according to bindings. When extending the constituents of language components, in the backend, the `DSL Comp Processor` (cf. Figure 3) has to be extended. In the frontend, the `Language Component Viewer` has to be updated. Furthermore, in the database the schema for language components has to be adapted to the changes to the language component constituents and to represent the project structures of new technological spaces. 2. When introducing new composition operators between language components, in the backend, the `Artifact Composer` for the specific artifacts of technological spaces and the `Language Infrastructure Generator` has to be implemented. Furthermore, the `Language Family Manager` has to be extended with the bindings specific to this new language composition operator. In the frontend, these changes have to be adopted by the `DSL Family Viewer` and `DSL Family Workbench`. Finally, the database schema for language families has to be updated. For all of these changes, the software architectures provides dedicated extension points or interfaces that can be extended and implemented, respectively.

5 Demonstration

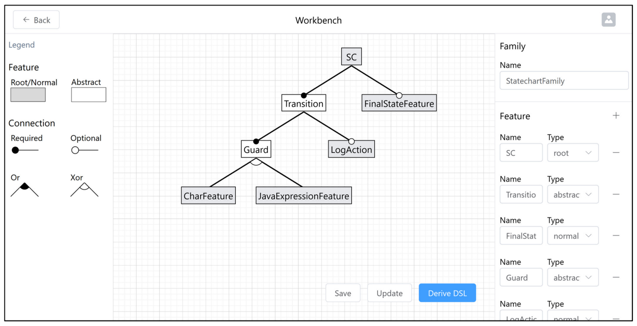
This section performs a walk through the presented platform by an example of a language family for statecharts. First, the language component library together with the import mechanism is shown. Afterwards, the language family workbench where imported languages can be reused as components and arranged in a language family is presented. And at last, the language family language family for statecharts is configured to derive a specific language variant.



Package Name: sc
Component Name: Statechart
Grammar: sc.Statechart

Name	Applied to	Optionality
IState	IState	optional
IAction	IAction	optional

Figure 4. Importing languages into the SCOLaR platform and configuring the derived language component.



Family Name: StatechartFamily

Name	Type
SC	root
Transitio	abstrac
FinalStat	normal
Guard	abstrac
LogActic	normal

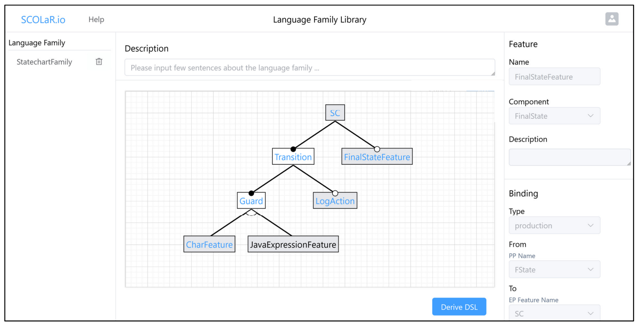
Figure 5. The workbench for creating language families.

5.1 Importing a Language Project

The first step towards reuse of existing language projects in SCOLaR is the import as language components. As mentioned in Section 3 our platform provides an automatic language component derivation mechanism. Importing language projects into the platform is possible via the language component library. By clicking the button *import*, a dialog for importing language projects opens. The selected project is uploaded to the platform, persisted, and a fitting language component is derived automatically. This language component can then be customized according to its provided and required extensions (cf. Figure 4). After the configuration, the component is persisted and available in the language component library, and the language family workbench for reuse.

5.2 Creating a Language Family

Figure 5 shows the language family workbench in the SCOLaR platform exemplified with a language family for statecharts. In the workbench, features can be removed and added, and features can be associated with language components from the library. In the view, abstract features are filled white and features backed with language components are filled grey. Editing the family is possible via the sidebar. The layout of the family is adjusted automatically whenever a feature is added or removed.



Name: FinalStateFeature
Component: FinalState
Type: production
PP Name: FState
EP Feature Name: SC

Figure 6. The systematic language composition process exemplified with a language family for automatons. Selected features are highlighted blue.

5.3 Configure and Export

To derive language variants from language families, the language family library enables choosing from a set of existing language families built in the workbench. To configure a language family, the user can simply select a family in the library and choose the variant of his choice by clicking on the features in the feature tree (cf. Figure 6). By clicking the button *Derive DSL* the configuration is applied, the selected language features are composed, and the language variant is downloaded as a zip archive. After that, the user can build the language project and utilize the language to define models in his language variant.

6 Conclusion

The SCOLaR low-code platform provides graphical means for language engineers to 1) import existing languages, 2) automatically derive a language component interface, that 3) enable reuse along a language family, 4) create and configure language families for various language variants, 5) and manage language components and families for reuse. The platform is still in its early development and since SCOLaR framework is built using MontiCore this is the only language workbench supported currently. However, in the future, we plan to extend our platform by supporting other language workbenches, e.g., XText[1] and to add other language composition operators besides embedding and aggregation [5] and even composition between different compatible technological spaces. Early user reports indicated that extending the platform with more detailed error reporting, and language component and family versioning would be helpful. For the exported language variants, we plan to add LSP[2] generation, providing languages with common editor features, e.g., syntax highlighting, folding, auto-completion etc..

[1]https://www.eclipse.org/Xtext/
[2]https://microsoft.github.io/language-server-protocol/

References

[1] Alexander C Bock and Ulrich Frank. 2021. Low-Code Platform. *Business & Information Systems Engineering* 63 (2021), 733–740.

[2] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, Gothenburg, Sweden, 65–75.

[3] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, Virtual Event, Canada, 35–46.

[4] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. 2022. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)* 70 (June 2022), 101117.

[5] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, Tallinn, Estonia, 1–8.

[6] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.

[7] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures* 54 (2018), 386–405.

[8] Jérôme Pfeiffer and Andreas Wortmann. 2021. Towards the Black-Box Aggregation of Language Components. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, ACM, Fukuoka, Japan, 576–585.

[9] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *2011 15th International Software Product Line Conference*. IEEE, Munich, Germany, 191–200.

[10] Massimo Tisi, Jean-Marie Mottu, Dimitrios S Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*. CEUR-WS.org, Eindhoven, Netherlands, 73–78.

[11] Jos Warmer and Anneke Kleppe. 2022. Freon: An Open Web Native Language Workbench. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Auckland, New Zealand, 30–35.

Received 2023-07-07; accepted 2023-09-01