Applying a Self-Extension Mechanism to DSLs for Establishing Model Libraries

Arkadii Gerasimov
Software Engineering, RWTH Aachen University
Aachen, Germany

Nico Jansen
Software Engineering, RWTH Aachen University
Aachen, Germany

Judith Michael
Software Engineering, RWTH Aachen University
Aachen, Germany

Bernhard Rumpe
Software Engineering, RWTH Aachen University
Aachen, Germany

Abstract

When applying model-driven engineering in an agile environment, new requirements continuously expand the domain scope and trigger an extension of the concepts covered by a Domain-Specific Language (DSL). While programming languages streamline code extension and reuse through libraries, similar approaches for DSLs are more complex or target a specific language. We present and discuss an approach for designing DSLs with a self-extension mechanism to enable model library creation and seamless reuse. We use the self-extension mechanism to introduce concepts in models, gather reusable models into a library, and provide an infrastructure for its usage. We explain our language-specific realization of the self-extension mechanism using a DSL for graphical user interfaces and discuss its model libraries with a use case from practice. The approach provides more flexibility for agile model-driven engineering. It enables application modelers to introduce and reuse concepts via models without changing the DSL, reducing the communication overhead within the development team.

CCS Concepts: • **Software and its engineering** → **Model-driven software engineering**; **Domain specific languages**; Graphical user interface languages; **Source code generation**; *Unified Modeling Language (UML)*; **Reusability**.

Keywords: Model-Driven Software Engineering, Domain-Specific Languages, Software Language Engineering, Model Library, Graphical User Interfaces

ACM Reference Format:
Arkadii Gerasimov, Nico Jansen, Judith Michael, and Bernhard Rumpe. 2024. Applying a Self-Extension Mechanism to DSLs for Establishing Model Libraries. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24), October 21–22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3689484.3690732

1 Introduction

Model-driven engineering (MDE) [68, 77] is becoming increasingly prevalent in modern development endeavors over various domains. It is applied, for example, in automotive [7, 10], aviation [29, 30], robotics [17, 22, 79], production [28, 33, 34], smart homes [50, 58], civil engineering [55, 75], software engineering [42, 48, 77], and systems engineering [65, 66], using models as an abstraction to separate the problem domain from the technical solution domain and establish a seamless development process. In MDE, models are the primary development artifacts that steer the engineering process and constitute the single source of truth [31].

Models adhere to modeling languages that specify the concrete and abstract syntax, well-formedness rules, and semantics [18, 40]. These languages are often tailored for a specific purpose or application domain to enable domain experts to create solutions within their expertise, e.g., by using known terminology. Such languages are called Domain-Specific Languages (DSLs) [20, 54]. In contrast General Purpose Languages (GPLs), such as UML [61] or SysML [32, 41], are generally applicable across multiple domains.

As modeling languages continuously evolve, the maintenance and support effort for the languages and their tools is also growing, as with any software product [27, 56]. In recent years, the reusability of existing language components has become an essential part of their research and development [9, 39, 43]. Thus, we can see a trend that languages are not built from scratch but are based on incorporating modular, reusable building blocks. Multiple composition techniques [25, 72, 78] and design patterns [24] have been researched to seamlessly integrate language components provided in corresponding libraries [15] and, ultimately, leverage software language product lines [52].

While these advances are fundamental for sustainable language development and maintenance, the techniques that enable language extension via models have yet to be explored [67]. In this work, we refer to one such technique as a self-extension mechanism, i.e., extending the language's functionality within models without modifying its metamodel (the language provides means for its extension, thus "self"−extension). The mechanism has been previously described for some DSLs [67, 76]. Such DSLs often have referencing mechanisms for models and enable reuse of the models, e.g., via a model library. The library concept has already been established for prominent programming languages such as Java [2] and Python [74]. Therefore, *it is only logical that introducing a library concept is one of the next steps in the evolution of modeling languages.* The specifications of large GPLs in modeling, such as UML [61], SysML [62], or its successor, SysML v2 [64], already define concepts of model libraries. Their implementation in GPLs shows that a self-extension mechanism and model libraries have great potential in modeling languages. While an extension mechanism requires language-specific implementation, its advantage is that modelers can provide extensions themselves and do not necessarily require the services or resources of a language engineer. Furthermore, domain-specific extensions do not demand their language variant but can be provided directly via libraries by proficient modelers. This improves cross-disciplinary applicability by establishing domain-specific model libraries of a base language to be incorporated into different application areas.

Although model libraries are already considered in individual languages and language families [67], this aspect of language extensibility still needs to be explored. While extensive contributions to software language engineering practices [51] and ideas for model and meta-model evolution exist, a general, reusable self-extension concept is crucial for developing smaller DSLs, as updating a DSL without extension points in place is time-consuming and costly to re-implement. Occasionally, a few languages provide a language-specific solution. However, a general technique and requirements for realizing a self-extension mechanism for modeling languages and using it to build model libraries must be explored.

We investigate and specify *how to establish a model library using a self-extension mechanism for DSLs.* Our contributions:

- A conceptual framework for building DSLs with model library support.
- A description and implementation of the components necessary to build such a DSL using a self-extension mechanism that enables seamless integration of new language concepts and maintenance of a model library.
- A comparison of our approach using the self-extension mechanism to a more straightforward alternative that introduces new concepts directly in the language.
- A validation of the comparison using a real-world information system and its DSL for defining Graphical User Interfaces (GUIs).

We list the components necessary to build a DSL with a model library that is easy to extend and use.

Our approach's general idea is to integrate and use the self-extension mechanism to enable the introduction of language concepts at the modeling level. The integration must allow describing semantics in the models or the handwritten artifacts supporting the generated code from model implementation. Such models can be transferred and reused as a library with the same tooling built for the language. Further, we use concepts from the Meta Object Facility (MOF) [59] to designate metamodel level (M2), model level (M1), and code level (M0). We use the older MOF specification since we limit our work to these three levels in this paper. Applying the approach to DSLs with arbitrary layers or applying the mechanism to higher abstraction levels is out of our work's scope. For simplicity, while describing a concrete syntax definition of a language and a code generator, we consider them to be on a metamodel level (language level).

Figure 1 illustrates the difference between introducing a new language concept for the two approaches. On the left-hand side, a DSL is defined (top) with the two features, B and C, which are used in particular models (bottom). The right-hand side shows the generalization of this language by, instead of introducing multiple specific language features, introducing a general feature D that is extendable and referencable on the model level. Since the language on the right is extended on the model level, the concepts can be grouped and packaged as a model library.

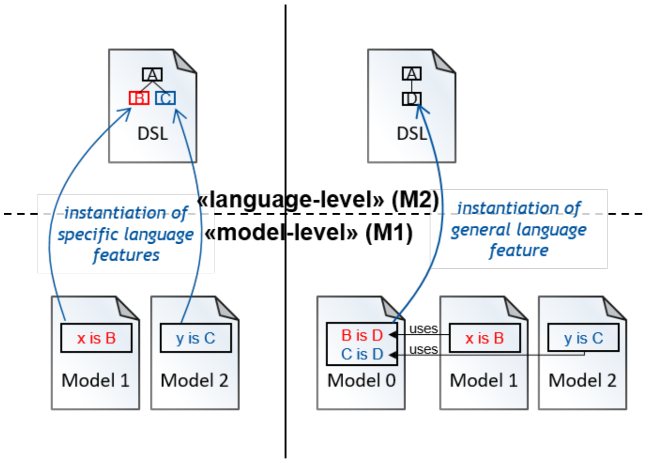


Figure 1. DSL extension techniques

We compare the approaches using the MaCoCo [35] information system as real-world example and GUI DSL that defines the system's GUI. We compare and discuss both strategies, highlighting the benefits and limitations of establishing model libraries using the self-extension mechanism.

The paper is structured as follows: The next section introduces preliminary work, including the technology and languages used. Section 3 investigates the requirements for a language's self-extension mechanism and parts of corresponding model libraries. Section 4 demonstrates an example of self-extension mechanism implementation. Section 5 presents a real-world use case in applying the implemented self-extension mechanism to a DSL for designing GUIs and establishing a model library. Section 6 compares the approach with its non-self-extendable predecessor. Section 7 evaluates the differences between the approach and its predecessor. Section 8 discusses the results, considering scenarios in which model libraries are beneficial and listing the approach's limitations. Section 9 presents related work contemplating associated techniques. The last section concludes.

2 Preliminaries

We introduce the technology stack and the real-world project used as an example application.

MontiCore. We develop textual DSLs using MontiCore - a workbench that produces infrastructure for the languages defined in a context-free grammar [44] in Extended Backus-Naur Form [80]. This includes a parser for the models of the language, which transforms the textual representation of the models into an Abstract Syntax Tree (AST), effectively defining the language metamodel (M2). AST classes are generated by MontiCore alongside a visitor infrastructure for their traversal. Components for creating, importing, and exporting symbol tables, a CLI tool, and other helpful utilities are derived automatically to support the development of a generator and other model processing tools. The code generated by MontiCore and its products can be extended with hand-written code using inheritance or hook points. A language engineer can also compose, extend, or aggregate MontiCore grammars [13, 15].

MaCoCo. We demonstrate the self-extension mechanism using the MaCoCo application [11, 35]. MaCoCo is an information system that supports managing and controlling faculty resources and projects [37]. User activities include budget management, planning staff on projects, recording working hours, etc. MaCoCo is actively used by about 50 institutes with around 190 daily users (Table 1).

Table 1. MaCoCo statistics

	Total
Modeled types (domain)	141
GUI-Models (web pages)	91
Lines of Code	11 mil
Active faculties	66
Users	1600
Logins per day	210

From a technical view, MaCoCo is a web application produced by the MontiGem generator with a Java backend and Angular frontend. It is a large project with over a million lines of code, where around 75% is generated.

MontiGem. MontiGem [1] is a framework that enables the creation of web applications. It includes a generator and a run-time environment to provide the frontend, backend, database, and infrastructure for communication between these components. A developer can quickly build a functional prototype by modeling the data structure as class diagrams and user interfaces with the GUI DSL and running the MontiGem generator. The frontend of a MontiGem product is an Angular application. MontiGem was applied to develop the MaCoCo application [35], process-aware information systems [23], low-code development platforms [21], digital twins [3], assistive systems [57], and IoT app stores [14].

First version of GUI DSL (GUI DSL 1). The first version of GUI DSL was created for describing user interfaces. A model of the language describes a web page by specifying domain objects to be loaded for the page and visual components to be displayed with the loaded data. Every component is defined in the grammar. A specific generator part transforms each component into the target code.

Second version of GUI DSL (GUI DSL 2). GUI DSL 2 is a rework of version 1. The DSL preserves general concepts from the first version and expands on them to support frequent changes and additions to the predefined component set. A model of the DSL describes a GUI component that can be a web page or a part of it. A component allows input parameters, e.g., strings or integers, in addition to the data source to enable the creation of configurable reusable components. Every specific component, e.g., a button or a table, is declared in a separate model. The language tooling handles arbitrary components defined by a modeler in the same way. This design is a foundation for realizing the self-extension mechanism, which leads to establishing model libraries.

3 Model Library and Self-Extension Mechanism

In an agile engineering process, the constantly changing domain scope requires an appropriate reaction in the implementation, including models and often their DSLs. This constant change requires a DSL user to introduce a new concept in some way, the direct solution being language extension, such as introducing a new grammar rule and adjusting the corresponding generator parts. An alternative is to work with more abstract concepts in the DSL and introduce new concepts in the models. Such models effectively form a library. Figure 2 shows the overall concept of establishing and employing such a library. Referenceable elements from a provided and precompiled model library are addressed and used in a source model. The same applies to the program artifacts synthesized from this source model. These artifacts import functionalities that were, in advance, generated from the library models or extended by hand. Thus, the library comes

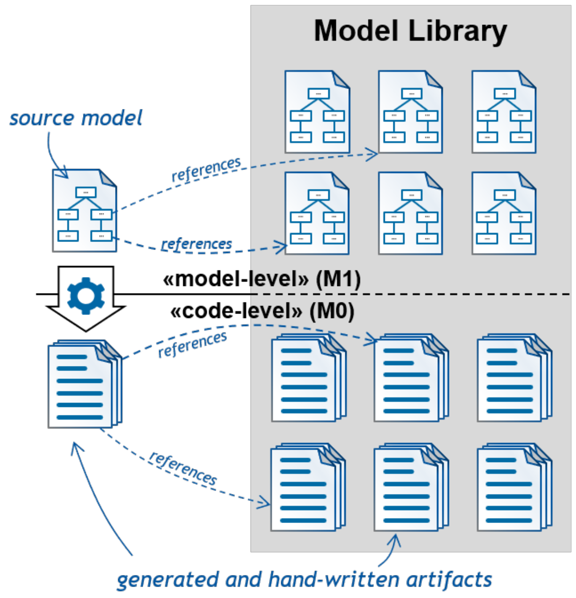


Figure 2. A model library concept where model elements refer to provided models, and the generated target code incorporates corresponding compiled artifacts

with compiled artifacts, similar to GPLs, so the generation process is only triggered for the models at hand.

The requirements for establishing and incorporating libraries are similar for different software languages, independent of their nature (i.e., programming or modeling). Thus, for a more intuitive understanding, we compare features required for a modeling library to similar concepts of Java:

- **Extendable concepts:** The language features abstract concepts for the lower-level concepts described in the models. Compared to Java, we have a similar notion, for instance, with the abstract concept of classes, which are implemented and can be provided externally.
- **Import mechanism:** Languages require a mechanism to import and use the models in another model. This includes a mechanism that allows for referencing particular model elements, regardless of whether they are provided in the same artifact or externally. Mechanisms such as references [71], or symbol tables [15] accomplish this task in modeling languages. Compared to Java, an implemented class can be imported and instantiated in another class.
- **Packaging mechanism:** The reusable models and associated artifacts must be suitably packaged into artifacts and exported. In Java, for instance, compiled files can be packed into a JAR file, which can be incorporated into other projects.

The packaging mechanism selects the elements to be incorporated, which is essential for reuse. For instance, if the models are backed by executable code, often generated from the models, the mechanism has to ensure that the code is correctly employed. Using a model from a library in another model produces the code that retains such a relation. This relationship is indicated in Figure 2, where the generated target code references artifacts delivered within the model library. Thus, a library of a DSL must precisely define deliverable packages that include models and associated artifacts. While the content of such packages depends on the corresponding language and its underlying language workbench [26], we can provide a general guideline for the required parts. The contents of a self-contained package are:

- **Model artifact:** The model contains the reusable elements and is at the core of a package. Depending on the implementation, the results from processing the models may be used instead of the model artifact that would serve as documentation. Such models can be compared to Java source files, which help a programmer review the original implementation.
- **Serialized symbol table:** Instead of the model, an artifact that exposes the referenceable model elements can be packaged. Such elements that can be referenced by their name are called symbols. These can be addressed intra- and inter-model-wide using a symbol table. Language workbenches such as MontiCore allow these tables to be stored and loaded efficiently. This provides an efficiency gain, especially for large models and massive model sets in libraries. These serialized symbol tables are comparable to the class files in Java libraries: A model is similar to a source file, and a serialized symbol table is similar to a compiled class.
- **Compiled target artifacts:** A package may contain generated from models artifacts or manually created extensions, such as executable GPL code. Since such artifacts are closely related to a model, they are also relevant for integrating library models into projects.
- **Additional artifacts:** Model components can be enriched with extra content. For example, they can include artifacts augmented with information synchronizing the generation process [46] to ensure the generated files match. Alternatively, application- or domain-specific artifacts, such as images for a GUI of an information system, may be relevant.

In the above list, the set of artifacts for the approach that relies on a direct DSL extension would include the language and generator extensions instead of the model and compiled target artifacts, respectively. For example, a DSL has a concept of an `Account`. If the `Account` is added to the DSL in the grammar with a mapping in the code generator, reusing it relies on the language and generator extensions. If the `Account` is introduced in a model, e.g., as a class in a class diagram, and the generator provides a general mapping for classes, the model (or equivalent concept description) and the mapped code are needed for their reuse.

4 The Self-Extension in GUI DSL 2

GUI DSL 2 implements a self-extension mechanism allowing application modelers and library developers to declare and implement new user interface components in the models of the DSL. The realization is similar to declaring new classes or functions in object-oriented languages. A model of the language contains a GUI component declaration with its (optional) implementation. The implementation consists of components imported from other models and instantiated as a part of the component.

MG

```
1 symbol GUIComponentDeclaration =
2     (["component"] | ["page"]) Name
3         "(" (GUIParam || ",")* ")" GUIBlock?;
4 symbol GUIParam implements Variable =
5     MCType Name ("=" default:Expression)?;
6
7 GUIComponent = "@" Name@GUIComponentDeclaration
8     "(" (GUINamedArg || ",")* ")";
9 GUINamedArg = Name@GUIParam "=" Expression;
```

Listing 1. GUI DSL 2 grammar excerpt. It specifies the concrete syntax for a component declaration and usage.

Listing 1 shows how MontiCore grammar (MG) of GUI DSL 2 specifies constructs necessary for realizing the self-extension mechanism. The declaration (lines 1-3) starts with a keyword `component` or `page` followed by an arbitrary component name, a list of input parameters (lines 4-5), and a block containing the component implementation. A component usage (lines 7-8) is indicated by an `@` sign followed by the name of the component and a list of arguments (line 9). The grammar additionally uses MontiCore features to specify a part of the abstract syntax connecting the name in the component usage (line 7) and the named arguments (line 9) to the symbols referencing a component declaration and parameters.

GUI DSL v2

```
1 package example;
2
3 import basic.GemText;
4
5 component TextExample() {
6   @GemText(value = "My text");
7 }
```

Listing 2. GUI model of a component displaying text.

Listing 2 demonstrates a component declaration `TextExample` that uses a component displaying `My text`. The component imports and instantiates `GemText`, which is used to display text specified in a parameter `value`. The `TextExample` component can further be imported and instantiated in other models, resulting in the text `My text` being displayed each time. The import mechanism is supported by the MontiCore tool. Using the imports and instantiation, components can be combined into new components, resulting in complete web pages. Further, the package and import declarations are omitted for brevity.

GUI DSL v2

```
1 package basic;
2
3 component GemText(String value)
```

Listing 3. GUI model of the `GemText`.

GUI DSL 2 also defines basic components, which differ from the combined components. Listing 3 shows the model for the basic component `GemText`. The model only defines the component's input parameters but does not specify the implementation. The component is implemented manually in the target language of a DSL generator, e.g., JavaScript, HTML, CSS, etc. As shown in Listing 2, using a component only requires the component's signature in the model. As a result, any new basic building block can be added without extending the metamodel.

If a model describes the implementation, the generator translates it to the target code. The component implementation is only used on the code level as compiled sources.

TypeScript

```
1  import { GemText } from "basic/GemText";
2  /*...*/
3  @Component({
4    template: `
5      <gem-text [value]="'My Text'"></gem-text>
6    `,
7    selector: `text-example`,
8    imports: [GemText]
9  })
10 export class TextExample { /*...*/ }
```

Listing 4. Mapping of a GUI model to the TypeScript (Angular) code.

The import and usage of a component translates to the import and usage of the component signature. The general idea can be seen in Figure 2, and an excerpt of the generated code for the `TextExample` is shown in Listing 4.

- The component import (Listing 2 line 5) is translated to Angular code imports (Listing 4 lines 1, 8).
- The name of a declared component (Listing 2 line 5) is converted to TypeScript class declaration (Listing 4 line 10) and HTML tags declaration (Listing 4 line 7).
- The name of a used component (Listing 2 line 6) is converted to HTML tag usage (Listing 4 line 5).
- The arguments (Listing 2 line 6) are translated to Angular parameter assignment (Listing 4 line 5).

The presented approach differs from the GUI DSL 1, where components are a part of the DSL grammar and models only describe complete web pages. Listing 5 shows how a text component is introduced in GUI DSL 1.

MG
```
Text = "textoutput" "{" Expression "}";
```
Listing 5. GUI DSL 1 grammar excerpt (simplified). It specifies the concrete syntax for a text component.

This and other components have a specific keyword and generation logic, which abstracts away their implementation details. However, if a new component has to be introduced, it must be added to the DSL grammar and usually the generator.

The key difference between the two approaches is the abstraction levels when (re-)defining components. The self-extension mechanism at the language (grammar) level eliminates the need to modify artifacts but makes the components indistinguishable. Further, we show how the self-extension mechanism supports establishing a model library and compare it to an alternative approach using GUI DSL 1 example.

5 Example

We use examples from the financial controlling domain to compare languages with and without the self-extension mechanism. The example models are simplified versions of the production code but have the same meaning and structure. The example workflow consists of three steps. The first two are valid for both versions of GUI DSL:

- **Step 1.** Create a staff overview page (see Figure 4). The page shows a list of persons with buttons to create a new person and to view a wiki page documenting the functionality.
- **Step 2.** Create an accounts overview page (see Figure 5). The page shows a list of accounts with buttons to create a new account and to view a wiki page. Compare the new page with the staff overview page and extract a common layout component.
- **Step 3.** Integrate the common layout component into a model library of GUI DSL 2.

We use the same data structure in the form of a class diagram shown in Figure 3 for both versions. Readers may find the full version in [36]. The GUI models reference the data structure to define the source of web pages' data.

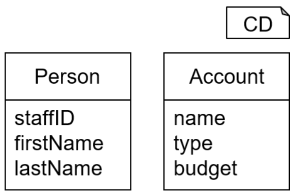
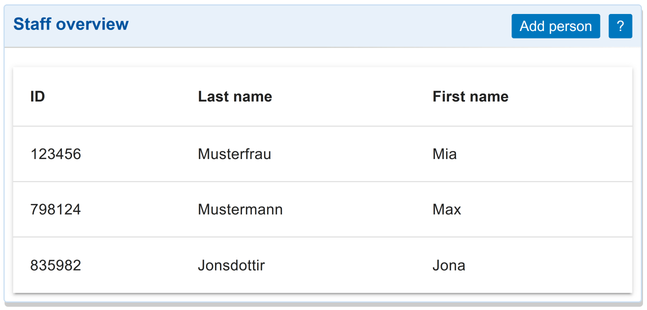


Figure 3. Data structure

In the example, we observe several roles responsible for the development on different abstraction levels:

- **Language engineer** modifies artifacts at the meta-model level (M2), such as a DSL grammar and the generator code.
- **Application modeler** (app modeler) modifies artifacts at the model level (M1), such as the models of the DSL that describe GUI components, including pages.
- **Component developer** modifies artifacts at the code level (M0), such as the code that implements or extends the functionality of the modeled GUI components.
- **Library developer** is a component developer responsible for maintaining a component library.

Step 1. The staff overview page is shown in Figure 4. Here and further, we demonstrate the GUI using only a single screenshot (despite having two versions of the pages) since the displayed result is practically the same. Differences are observed only in minor details, such as more rounded corners, slightly different colors, and other visual styles explained by the updated theme for the components in GUI DSL 2.



ID	Last name	First name
123456	Musterfrau	Mia
798124	Mustermann	Max
835982	Jonsdottir	Jona

Figure 4. Staff overview page

Models describing the page in GUI DSL 1 (Listing 6) and GUI DSL 2 (Listing 7) are structurally very similar. Both models specify the data source in the web page declaration (line 1) and define visual components to display.

GUI DSL v1
```
 1 webpage StaffOverview(all Person persons) {
 2   card {
 3     head {
 4       row(stretch) {
 5         textoutput { "Staff overview" }
 6         row (r) {
 7           button "Add person" { click->add() }
 8           button "?" { click->wiki() }
 9         }}}
10     body {
11       datatable "overview" {
12         rows <persons {
13           column "ID", <staffID;
14           column "Last Name", <lastName;
15           column "First name", <firstName;
16         }}}
17 }}
```
Listing 6. GUI DSL 1 model of the staff overview page.

GUI DSL v2

```
1  page StaffOverview(List<Person> persons) {
2    @GemCard(
3      title = @GemRow(
4        hAlign = "stretch", components = [
5          @GemText(value = "Staff overview"),
6          @GemRow(hAlign = "right", components=[
7            @GemButton(label = "Add person",
8              click = add),
9            @GemButton(label = "?", click = wiki)
10         ])]),
11     component = @GemTable(
12       rows = persons,
13       columns = [
14         TableColumn("staffID", "ID"),
15         TableColumn("lastName", "Last name"),
16         TableColumn("firstName", "First name")
17       ]));}
```

Listing 7. GUI DSL 2 model of the staff overview page.

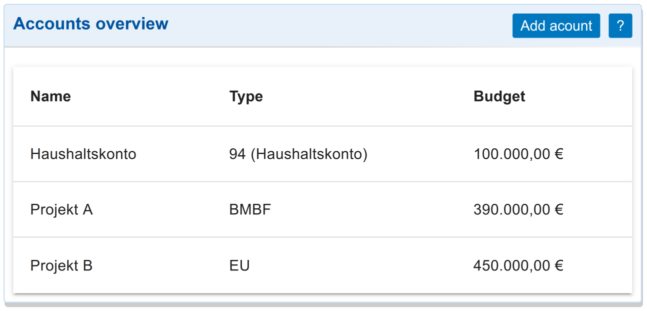
In a GUI DSL 1 model, the web page declaration only specifies the data source and what items should be retrieved (all or single items). In a GUI DSL 2 model, the declaration specifies a list of input parameters, and the data is loaded automatically whenever possible. Otherwise, the input parameter has to be set when a component is instantiated. For example, a `GemText` value is a `String` input whose value is set to `"Staff overview"` (Listing 7, line 5).

A GUI DSL 1 model body (Listing 6, lines 2-17) instantiates visual components using component-specific syntax defined in the language grammar. For example, a `row` aligns components listed in braces from left to right and may control the position and size of the components using a keyword in brackets, such as `stretch` (line 4) to stretch the items for filling the available space or `r` (line 6) to align all of the items to the right side. A button has a different syntax, specifying the button's text after the `button` keyword and a function to call on click in braces (lines 7-8). The non-uniform syntax allows a more specialized and compact description.

The GUI DSL 2 grammar defines a single pattern for all visual components and is more verbose. The values of component arguments can be literals (Listing 7, line 5), expressions (line 12), lists (line 13), and other components (line 3), providing sufficient expressive power to describe a web page. As a result, a single GUI DSL 2 model is typically bigger than a GUI DSL 1 model.

Despite the language differences, an application modeler would not notice a big difference between the workflow for creating either of the models. The application modeler specifies the data source and the page content using the available data structure and the same set of predefined components.

Step 2. A modeler creates an account overview page. The same layout pattern should be used since views with the same purpose must look similar according to the consistency property of design principles [70]. As a result, the page's visual appearance is similar to the staff overview page, except for the data and other textual information (see Figure 5).



Name	Type	Budget
Haushaltskonto	94 (Haushaltskonto)	100.000,00 €
Projekt A	BMBF	390.000,00 €
Projekt B	EU	450.000,00 €

Figure 5. Account overview page

Following the Don't Repeat Yourself (DRY) principle [73], the models are refactored to derive a visual component specifying a layout template for the overview pages.

MG

```
Overview = "overview" title:StringLiteral "{"
    PageElement "}";
```

Listing 8. GUI DSL 1 extension for the overview.

GUI DSL 1 implements the new concept via language extension, including a new grammar rule (see Listing 8) and a generator extension for a simple mapping from the model to the target language. The `overview` component hides details such as the usage of a `card` and a `textoutput` component.

GUI DSL v1

```
webpage AccountsOverview(all Account accounts){
  overview "Account overview" {
    head {
      button "Add account" { click->add() }
      button "?" { click->wiki() }
    }
    body {
      datatable "overview" {
        rows <accounts {
          column "Name", <name;
          column "Type", <type;
          column "Budget", euro(<budget);
        }}}}}
```

Listing 9. GUI DSL 1 model of the `account overview`.

The account overview page model uses the new component to define the same layout as the staff overview more concisely (see Listing 9). In the GUI DSL 2 case (Listing 10), a new `Overview` model is created. The model defines a page skeleton consisting of a card with a title, buttons, and an arbitrary element as a card body with a default layout. The account overview page model transforms where the layout details become hidden (Listing 11).

GUI DSL v2

```
page Overview(
  String title,
  List<GUIViewElement> buttons,
  GUIViewElement main
) {
  @GemCard(
    title = @GemRow(hAlign = "stretch",
      components = [
        @GemText(value = title),
        @GemRow(hAlign = "right",
          components = buttons)
    ]),
    component = main
  );}
```

Listing 10. GUI DSL 2 model for an overview page.

GUI DSL v2

```
page AccountsOverview(List<Account> accounts) {
  @Overview(title = "Accounts overview",
    buttons = [
      @GemButton(label = "Add account",
        click = add),
      @GemButton(label = "?", click = wiki)
    ],
    main = @GemTable(
      rows = accounts,
      columns = [
        TableColumn("name", "Name"),
        TableColumn("type", "Type"),
        TableColumn("budget", "Budget")
      ]));}
```

Listing 11. GUI DSL 2 model of the account overview.

Step 3. The component is recognized as useful for different projects, so it should become available for reuse. A library developer integrates the `Overview` component into an existing or a newly created model library. For GUI DSL 1 users, the component is available since it was added to the language in the previous step.

In GUI DSL 2, the library developer transfers artifacts into a library that consists of

- a set of models, each describing a GUI component, and
- a set of hand-written extensions of the components.

In the example, the `Overview` component is fully defined by its model. The library uses the same (or extended) tooling as the application and produces the same code. If extensions for the component are required, the library developer adjusts the model, adds hand-written extensions, or replaces the generated implementation with a hand-written one. The library developer pushes the update, which triggers a continuous integration and deployment pipeline.

In the pipeline (Figure 6), MontiCore processes the models and produces artifacts containing the component signatures without their implementation, i.e., exported symbols of the models. The component implementations are compiled with

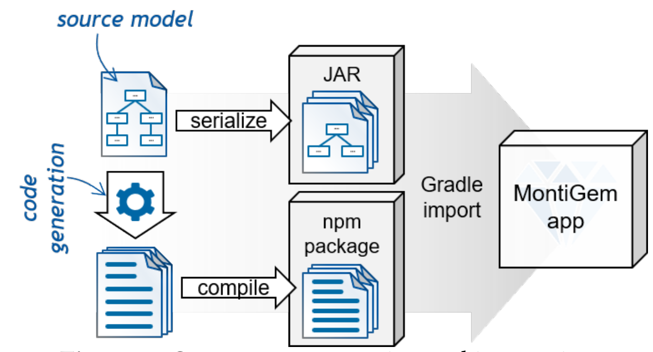


Figure 6. Component processing and integration

the target code compiler, in the example, Angular and TypeScript compilers. The artifacts produced from models are packaged into `JAR` files. The compiled target code sources are packaged into `npm` packages. We use different packaging systems since MontiCore is a Java-based tool, while the target code is JavaScript, HTML, and CSS. The packages are imported into a MontiGem application. To simplify the process, MontiGem applications use the Gradle build tool and a setup provided by a MontiGem Gradle plugin to import the library automatically and set up the build environment.

Technically, it is possible to derive package imports from model imports using the same principle of the self-extension mechanism, which is to implement consistent mapping from a set of models into a library package declaration and usage. However, this is outside this paper's scope.

6 Comparison

Although the result is similar, the workflow for creating a page and a layout template differs for an application modeler.

Using the GUI DSL 1, the workflow consists of the following steps (see Figure 7a):

1. The app modeler creates an account overview page with the same layout as the staff overview.
2. The app modeler communicates with a language engineer to add the layout component to the language.
3. The language engineer adds the component to the DSL and, if required, communicates with a component developer to extend it with custom logic or appearance.
4. After the component is added to the language, the app modeler integrates it into the models.
5. The app modeler may provide feedback and set additional requirements to the language or component developer, depending on the result.

If a new component is simple, the component developer is not involved. In our example, a language engineer may add a preprocessing step that expands the `Overview` component into a card, text, and other parts previously present in the model. However, if the app modeler requires special features for the overview, such as additional search fields or a custom style, the expertise of a component developer is needed.

Using the GUI DSL 2, the workflow consists of the following steps (see Figure 7b):

1. The app modeler creates an account overview page with the same layout as the staff overview.
2. The app modeler moves the common layout to a new model specifying the component.
3. The app modeler directly integrates the new component into the models and, if required, communicates with a component/library developer to add custom logic or appearance for the component.
4. The app modeler reviews the result and may decide to redefine the component or provide additional requirements for the component/library developer.

In the example, a library developer replaces a component developer during the third step when the component is integrated into a library (see Figure 7c). Since the library developer becomes responsible for deploying the component, the application modeler must communicate with them, and the workflow becomes closer to the GUI DSL 1 workflow. The component extension step remains optional. In our example, the `Overview` component (Listing 10) is equivalent to the initial layout setup since it does not require an extension.

The derived `Overview` model is domain-independent and can be reused for other overview pages in different projects. With time, project development results in a set of such components that form a library. Similarly, later projects expand an existing or create a new model library.

The workflows show that an extra step is necessary at the initial application development stages if the self-extension mechanism is absent. In the GUI DSL 1 workflow, an app modeler passes a component definition to a language engineer that integrates the component into the DSL. This corresponds to an integration delay of at least the DSL redeployment time. The workflow of GUI DSL 2 has a similar delay, but only after the component is integrated into the model library, where the delay is equal to the redeployment time of the library. However, the delay only applies to the application modelers' activities. The library modeler can continue working on the library component after its integration. In the GUI DSL 1 workflow, the component developer experiences the integration delay regardless. Since the described workflows are executed continuously during the agile engineering process and repeatedly for each component, the delay accumulates with each new component and update.

7 Evaluation

Using the presented example and the MaCoCo application, we compare DSLs with and without the self-extension mechanism. The criteria are workflow difference and the amount of code. Since the MaCoCo application still uses GUI DSL 1 and equivalent library components, we can compare models of different DSL versions.

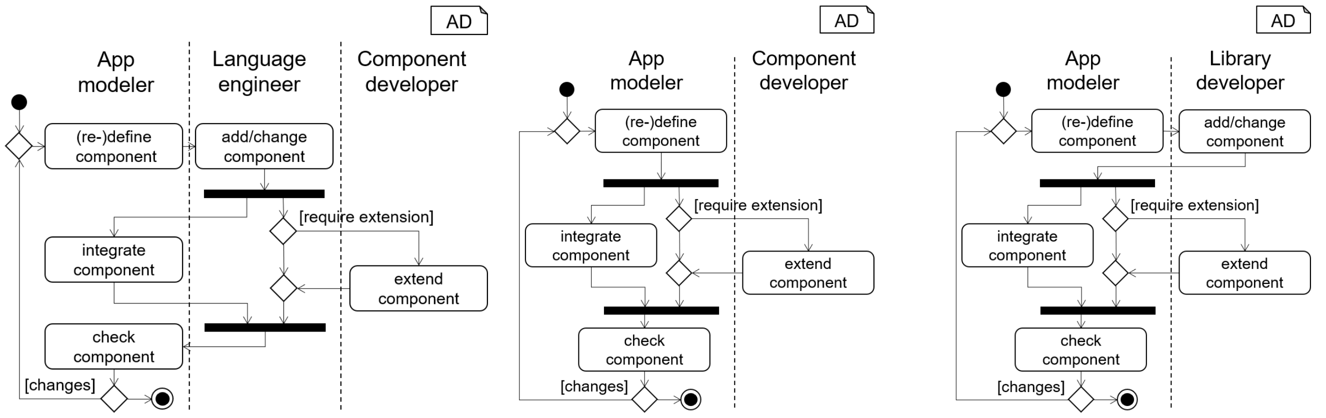
We compare the difference in library components, such as `Overview` in GUI DSL 1 (Listing 8) and GUI DSL 2 (Listing 10). Application models, such as staff and accounts overview pages, are not considered. These models represent web pages constructed from library components, and their implementation is equivalent, including their hand-written code extensions. We also do not consider the abstract concept implementation in GUI DSL 2 since this code remains unchanged when introducing a library component. However, this plays an important role in language engineering (see Section 8).

Table 2 demonstrates the statistics on the amount of code used to integrate components into the projects for different DSL versions. The components include our `Overview` example and portions of layout and chart libraries created in MaCoCo and updated to new versions from GUI DSL 1 to GUI DSL 2, i.e., they represent equivalent components from an app modeler perspective. The layout library portion contains card, row, and column components. The chart library portion includes pie, bar, and line charts.

Table 2. Lines of code (LoC) for GUI components

	GUI DSL 1	GUI DSL 2
Overview component		
Grammar	1	0
Generator	35	0
Model	0	32
Total	36	32
Layout library		
Grammar	118	0
Generator	307	0
Model	0	158
Hand-written	126	291
Total	551	449
Charts library		
Grammar	40	0
Generator	243	0
Model	0	106
Hand-written	1245	1036
Total	1528	1272

The components have similar implementation since we did not change languages and technologies besides the DSL. However, the code is distributed along the artifacts differently for the GUI DSL versions. For example, the overview component is implemented in the grammar and generator of GUI DSL 1, whereas it is fully implemented in a model using GUI DSL 2. The total amount of code is similar (36 vs. 32 LoC), with a slight difference that can be explained by a more complex Java syntax in the generator implementation compared to a specialized GUI DSL 2 syntax. As a rule, a component's interface is fully defined in a grammar (GUI DSL 1) or a model (GUI DSL 2). However, the GUI DSL 2 model also contains parts of the component's implementation, which makes a



(a) GUI DSL 1 **(b)** GUI DSL 2 before library integration **(c)** GUI DSL 2 after library integration

Figure 7. Adding/changing a component: Workflows

model always bigger than a grammar extension (32 vs. 1, 158 vs. 118, and 106 vs. 40 LoC for `Overview component`, layout, and chart library, respectively). The rest of the code defines the component's implementation, i.e., the generator code (GUI DSL 1), the model (GUI DSL 2), or a hand-written extension. The contribution of each part varies, but the total LoC for a GUI DSL 1 implementation is always higher and stays within 20% difference, which, besides GUI DSL 2 syntax being more compact than Java in the generator, is created by component updates reducing the code. Since the amount of code remains within the same bounds for both GUI DSL versions and the code does not change its meaning, the effort spent on the component implementation is similar.

However, as we demonstrate in the workflows in Figure 7a and Figure 7b, the development process for GUI DSL 2 avoids some delays during the component development. The GUI DSL 1 workflow includes adding or changing the component by a language engineer, which causes delays in the activities of other roles. In our experience working with GUI DSL 1, the delay varied from several minutes to a few working days, depending on the response time of a language engineer. Such a *delay in the application modeler's and component developer's activities* quickly accumulates with continuous requirement changes in agile development processes.

The self-extension mechanism simplifies the workflow for introducing a new domain concept, such as a GUI component in our example. We observe the benefits when dealing with isolated cases of component creation. However, the mechanism has disadvantages and various tradeoffs (see Section 8).

8 Discussion

We discuss the differences between deciding for or against using the self-extension mechanism to create a model library.

Development costs. The self-extension mechanism poses challenges during the early stages of the language development. A DSL requires a construct that abstracts domain-specific parts.

In our example, text, buttons, and other components with their configuration are represented by a general component and input parameter concepts. The application-specific information is given to the components as arguments. For example, a text component receives input such as a user name or an address depending on a domain and a use case. It took us several development iterations to find a solution that fits different use cases for various applications.

Furthermore, the tooling needs to support the self-extension mechanism. A code generator (M2) has to work with an abstract concept, such as a GUI component. At the same time, the generator needs to ensure that relations between models (M1) representing specific component types, such as a button and a page using it, are mapped to the target code (M0) without relying on the component type. In other words, the GUI DSL 2 tooling does not distinguish, e.g., a button from a table or a web page. Our solution is to describe a component's signature declaration and usage in the model and translate them uniformly into the declaration and usage in the target code for every component, bridging the gap between the M2 and M0 levels in the tooling without relying on component- or application-specific information.

Developing, deploying, integrating, and maintaining a model library's importing and packaging mechanisms also creates additional work since it requires additional configuration and scripts.

Shift of the skill sets. With the self-extension mechanism, a modeler is responsible for introducing a new concept. The modeler does not only decide what concept they need but also how it works. In our example, the overview component is implemented by a language developer (GUI DSL 1) or an app modeler (GUI DSL 2). GUI DSL 2 that has the self-extension mechanism demands a higher level of expertise in the GUI domain from app modelers since they have to know how to define a layout for an overview.

Syntactic variety. A prerequisite of a self-extension mechanism is an abstract concept in a DSL, which forces the lower-level concepts to have identical syntactical structures. On the one hand, the language syntax is easier to memorize since the same pattern is guaranteed. On the other, adding specialized syntax related to domain concepts becomes challenging. In our examples, the GUI DSL 2 concrete syntax ended up bulkier than its predecessor, and the models are harder to understand without tooling support.

Mixed approaches. Our examples only include GUI DSL 1 and GUI DSL 2. However, there are approaches based on mixed usage of libraries and the self-extension mechanism. For example, a library can be established for GUI DSL 1 by creating reusable language extensions and mappings. However, the development workflow from Section 6 would not change and have similar shortcomings of GUI DSL 1.

Alternatively, GUI DSL 1 could keep a set of basic components in the metamodel and add a self-extension mechanism on top of the predefined set. Compared to GUI DSL 2, the benefit is that basic components have a specific syntax and generation. The language loses uniformity as a drawback, and the DSL tooling has additional complexity.

Different concepts in the same language may also either be a part of a self-extension mechanism or not. For example, GUI DSL 2 includes concepts, such as loops, that allow a modeler to repeat a GUI part, for example, a button for each person in a list. Although creating a loop component is possible, we define a different semantic and syntax for a loop. A loop specifies repeated GUI parts and has no visual properties like components. Since introducing new kinds of loops is not required in our projects, the loop is defined in the metamodel.

Approach selection. We summarize the tradeoffs of integrating and using a self-extension mechanism and model libraries to determine when they should be used.

Integrating a self-extension mechanism with a model library is beneficial if the scope of a DSL continuously grows via newly introduced concepts. Our example demonstrates a new GUI component being created for a web page. Since our tool is used for several projects [11], new components are continuously being created to satisfy new requirements. We have created 38 library components so far. Those components implement an abstract "component" concept represented by four main parts: component package, import, declaration, and usage. The parts also consist of 15 minor parts, such as component input parameters, arguments, and loops, that are necessary to construct and use a component. GUI DSL 2 also uses over a hundred basic concepts, such as expressions and literals, from a MontiCore grammar library.

A DSL tailored for a few specific use cases has no use for a self-extension mechanism since it would only introduce additional setup costs. For example, the MaCoCo project was the only one using GUI DSL at the beginning. At that point, the first version of the DSL was created quickly and could be easily used. After years, we decided to reuse the DSL for other projects, so we had extra costs in adjusting the language. It took around 12 revisions to reach a stable version of the DSL, enabling its reuse in other application domains. A developer has to plan for such cases carefully. However, changing requirements in agile processes might not make such planning feasible from the start of the development.

A self-extension mechanism introduces a challenge if a DSL is built for modelers with narrow expertise since it forces semantics to a model and code levels. For example, GUI DSL 1 builds the overview component into the language, and an app modeler does not concern themselves with its implementation. With GUI DSL 2, the modeler must understand how to define a new component to utilize the self-extension mechanism efficiently.

Limitations. Our work explores using the self-extension mechanism to support model libraries and focuses on a three-layered language architecture. Generalizing the approach to architectures with an arbitrary amount of layers and applying the mechanism to higher abstraction levels could be possible. For example, an abstract construct at a metametamodel level could enable a metamodel to introduce constructs for building languages in different ways. However, the research of such cases is out of our work's scope.

9 Related Work

Several approaches mention the idea of a self-extension mechanism and model libraries, especially in programming. This trend is gaining popularity in the modeling community, as seen in major languages such as UML and SysML. Modern language workbenches such as Xtext [5], MPS [16], and MontiCore [44] support different mechanisms for referencing and importing additional model artifacts. Our contribution adds a general approach to realizing model libraries.

Seidewitz elaborates on a metasemantic protocol that enables language users to extend the language within itself [67]. Based on the general idea proposed for programming languages [49], the concept is refined and tailored towards modeling. Seidewitz describes that such a protocol requires *formal semantics* as well as the corresponding *abstract* and *concrete syntax* for realizing extension on the model level. He positions this strategy in the context of SysML v2 and its metalanguage, the Kernel Modeling Language (KerML) [63], providing a model library based on abstract extension points similar to a type-instance relation. The work points out how model elements are referred to in different scenarios for UML, SysML (v2), and KerML. The approach envisions mapping each specific element to KerML base elements, which already provide basic language self-extension capabilities. Furthermore, Seidewitz highlights the importance of realizing the prerequisites in one seamless technological scope or ecosystem. Our approach can be described as a realization within the MontiCore ecosystem. Reusing MontiCore's vast library of language components [12] directly provides the concrete and abstract syntax for establishing referenceable elements intra- and inter-artifact-wide. The generated symbol table infrastructure, facilitating loading a model's symbol table on demand [15], provides the required technical capabilities. Additionally, our approach gives a practical example of how to evolve a language toward self-extensibility.

Visser presents a case study on engineering modeling languages with WebDSL [76], a modeling language for designing web applications. The study examines the engineering process of a DSL, including identifying domain concepts, translating these into abstract syntax enriched with concrete syntax, parsing, code generation, and potential language extension. In this regard, the work considers different forms of language extensibility. It describes a trade-off between generative extension (at the language level) and non-generative extension (at the model level). In this regard, establishing non-generative extension points is discussed to create a library of referenceable artifacts incorporating the corresponding code snippets that modelers can use. The work uses a mixed approach with a restricted set of basic building blocks defined only at the language level that can be used to create more complex components. While this extension mechanism is not the paper's main focus, it abstractly describes the technique we are exploring in our work. Even the use case for creating web applications is similar, so our approach can be understood as an intellectual successor with a more detailed view of the self-extension aspects.

Creating a model library instead of defining domain concepts in a language has been mentioned in Karsai et al. [47] as a guideline suggesting limiting the number of language elements. The authors also point out that such language has to introduce more elaborate concepts that enable reusability for different domains without going into further details. Our work specifies conditions and creates a model library.

Selic [69] discusses model libraries via UML profiles. The work suggests that a model library can be created by refining a modeling language, whereas our approach is to create a new DSL or rework an existing one. The result is similar to how a model library represents specific domain concepts. The tooling, however, has information about what profile a library belongs to, thus allowing it to derive the semantics of the library elements. In our work, we define the semantics on the level of target code, specifying the exact meaning within a model and hand-written code extending the generated one.

IFML and WebRatio [8] utilize said profiling mechanism to provide a model library built with an extension mechanism different from the one presented in our work. An IFML extension requires several artifacts representing an implementation of a `ViewElement`, where a `ViewElement` is similar to our GUI component concept. The artifacts include templates and implementation code that define the signature, behavior, and design of the `ViewElements`. Such design is closer to GUI DSL 1 if it had generator plugins that specify code generation for individual components.

Bierhoff et al. [6] present the incremental development of a DSL. Their setup is similar to ours and uses an example to create an initial version of the language further adjusted for a different aspect of the same domain. Initially, the authors built a DSL to describe a to-do list application. Their DSL extension was required when adding a new feature showing incomplete items in the to-do list. It is reported as a low-effort modification, which was resolved by extending the grammar and generator of the language. We describe a mechanism that supports a model library without modifying the DSL.

A similar approach is described by Oberortner et al. [60] where they extend their DSLs reacting to user requirements. Language changes primarily include adding new concepts in a meta-model without considering alternative solutions.

Gerasimov et al. [38] propose an idea to solve the need to add new visual components to a GUI language quickly. Our work realizes the proposed mechanism and presents the results we analyze and compare with the approach where the concepts are directly introduced in a DSL.

Our topic focuses on bringing new concepts to a DSL, which relates to works on model and meta-model co-evolution. Hölldobler et al. [45] describes a DSL for language transformation, whose models are used to derive a new language alongside the tooling for migration of models. Cicchetti et al. [19] and Bell [4] similarly identify different types of changes, such as generalizing properties of a meta-model, and demonstrate their approach on Petri Nets. Mengerink et al. [53] evaluate to what extent existing solutions handle automatic model and meta-model co-evolution. These works tackle the problem of changing models alongside the meta-model encountered in our work. Although closely related, such works do not consider shifting domain concepts from a meta-model to the models.

10 Conclusion

This paper describes the concept of a model library and its prerequisites, including self-extension, import, and packaging mechanisms. Using the MaCoCo application as an example, we explain the self-extension mechanism and compare it to a more traditional solution, where domain concepts are defined in the DSL. We highlight the benefits of using the mechanism to improve the development workflow for an individual case and propose model libraries as the expansion of the idea. The results are validated by comparing two versions of the same DSL and equivalent concept implementations.

Model libraries facilitate the reuse of domain concepts across projects. The self-extension mechanism makes the model libraries easy to extend, improving workflow in an agile model-driven engineering environment.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de.

References

[1] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19) (LNI, Vol. P-304)*. GI, 59–66.

[2] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java Programming Language*. Addison Wesley Professional.

[3] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. 2022. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)* 70 (2022). https://doi.org/10.1016/j.cola.2022.101121

[4] Peter Bell. 2007. Automated Transformation of Statements within Evolving Domain Specific Languages. In *7th OOPSLA Workshop on Domain-Specific Modeling*.

[5] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

[6] Kevin Bierhoff, Edy S Liongosari, and Kishore S Swaminathan. 2006. Incremental development of a domain-specific language that supports multiple application styles. In *OOPSLA 6th Workshop on Domain Specific Modeling*. 67–78.

[7] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabo, Sandra Torchiaro, Sara Tucci, et al. 2013. EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design* (2013), 456. https://doi.org/10.4018/978-1-4666-3922-5.ch023

[8] Marco Brambilla and Piero Fraternali. 2014. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann.

[9] Barrett Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. 2015. *Globalized Domain Specific Language Engineering*. Springer, 43–69. https://doi.org/10.1007/978-3-319-26172-0_4

[10] Alessio Bucaioni, Vlatko Dimic, Mattias Gålnander, Henrik Lönn, and John Lundbäck. 2021. Transferring a model-based development methodology to the automotive industry. In *2021 22nd IEEE Int. Conf. on Industrial Technology (ICIT)*, Vol. 1. 762–767. https://doi.org/10.1109/ICIT46573.2021.9453680

[11] Constantin Buschhaus, Arkadii Gerasimov, Jörg Christian Kirchhof, Judith Michael, Lukas Netz, Bernhard Rumpe, and Sebastian Stüber. 2024. Lessons Learned from Applying Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator Framework. *Science of Computer Programming* 232 (2024), 103033. https://doi.org/10.1016/j.scico.2023.103033

[12] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)* 19, 3 (2020), 3:1–16. https://doi.org/10.5381/jot.2020.19.3.a4

[13] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2021. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In *Composing Model-Based Analysis Tools*. Springer, 217–234. https://doi.org/10.1007/978-3-030-81915-6_10

[14] Arvid Butting, Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Radoslav Orlov, and Bernhard Rumpe. 2022. Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices. In *21th ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE 22)*. ACM, 108–121. https://doi.org/10.1145/3564719.3568689

[15] Arvid Butting, Judith Michael, and Bernhard Rumpe. 2022. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology (JOT)* 21 (2022), 4:1–13. https://doi.org/10.5381/jot.2022.21.4.a5

[16] Fabien Campagne. 2014. *The MPS Language Workbench: Volume I.* Vol. 1. Fabien Campagne.

[17] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, Ivano Malavolta, Andreas Wortmann, and Patrizio Pelliccione. 2022. Model-driven engineering for mobile robotic systems: a systematic mapping study. *Software and Systems Modeling* 21, 1 (2022), 19–49. https://doi.org/10.1007/s10270-021-00908-8

[18] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'09) (LNCS 5795)*. Springer, 670–684. https://doi.org/10.1007/978-3-642-04425-0_54

[19] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating Co-evolution in Model-Driven Engineering. In *12th Int. IEEE Enterprise Distributed Object Computing Conference*. 222–231. https://doi.org/10.1109/EDOC.2008.44

[20] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. https://doi.org/10.1201/b21841

[21] 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 (2022). https://doi.org/10.1016/j.cola.2022.101117

[22] Edson de Araújo Silva, Eduardo Valentin, Jose Reginaldo Hughes Carvalho, and Raimundo da Silva Barreto. 2021. A survey of Model Driven Engineering in robotics. *Journal of Computer Languages* 62 (2021), 101021. https://doi.org/10.1016/j.cola.2020.101021

[23] Imke Drave, Judith Michael, Erik Müller, Bernhard Rumpe, and Simon Varga. 2022. Model-Driven Engineering of Process-Aware Information Systems. *Springer Nature Computer Science Journal* 3 (2022). https://doi.org/10.1007/s42979-022-01334-3

[24] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology (JOT)* 21, 4 (2022), 4:1–13. https://doi.org/10.5381/jot.2022.21.4.a4

[25] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *12th Workshop on Language Descriptions, Tools, and Applications*. https://doi.org/10.1145/2427048

[26] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, and Jimi Woning. 2013. The State of the Art in Language Workbenches. In *Int. Conf. on Software Language Engineering*. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[27] J-M Favre. 2005. Languages evolve too! Changing the Software Time Scale. In *8th Int. Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 33–42. https://doi.org/10.1109/IWPSE.2005.22

[28] Kevin Feichtinger, Kristof Meixner, Felix Rinker, István Koren, Holger Eichelberger, Tonja Heinemann, Jörg Holtmann, Marco Konersmann, Judith Michael, Eva-Maria Neumann, Jérôme Pfeiffer, Rick Rabiser, Matthias Riebisch, and Klaus Schmid. 2022. Industry Voices on Software Engineering Challenges in Cyber-Physical Production Systems Engineering. In *2022 IEEE 27th Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*. IEEE. https://doi.org/10.1109/ETFA52439.2022.9921568

[29] Peter H Feiler and David P Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.

[30] Damien Foures, Mathieu Acher, Olivier Barais, Benoit Combemale, Jean-Marc Jézéquel, and Jörg Kienzle. 2023. Experience in Specializing a Generic Realization Language for SPL Engineering at Airbus. In *ACM/IEEE 26th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*. 319–330. https://doi.org/10.1109/MODELS58315.2023.00035

[31] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)* (2007), 37–54. https://doi.org/10.1109/FOSE.2007.14

[32] Sanford Friedenthal, Alan Moore, and Rick Steiner. 2014. *A Practical Guide to SysML: The Systems Modeling Language.* Morgan Kaufmann.

[33] Shan Fur, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2023. Sustainable Digital Twin Engineering for the Internet of Production. In *Digital Twin Driven Intelligent Systems and Emerging Metaverse.* Springer Nature Singapore, 101–121. https://doi.org/10.1007/978-981-99-0252-1_4

[34] Antonio Garmendia, Manuel Wimmer, Alexandra Mazak-Huemer, Esther Guerra, and Juan de Lara. 2020. Modelling Production System Families with AutomationML. In *2020 25th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. 1057–1060. https://doi.org/10.1109/ETFA46521.2020.9211894

[35] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In *Comp. Proc. of Modellierung 2020 Short, Workshop and Tools & Demo Papers.* CEUR-WS.org, 22–30.

[36] Arkadii Gerasimov, Patricia Heuser, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, Simon Varga, and Galina Volkova. 2022. *Domain Modelling of Financial, Project and Staff Management.* https://doi.org/10.5281/zenodo.6422355

[37] Arkadii Gerasimov, Peter Letmathe, Judith Michael, Lukas Netz, and Bernhard Rumpe. 2024. Modeling Financial, Project and Staff Management: A Case Report from the MaCoCo Project. *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling* 19 (2024). https://doi.org/10.18417/emisa.19.3

[38] Arkadii Gerasimov, Judith Michael, Lukas Netz, and Bernhard Rumpe. 2021. Agile Generator-Based GUI Modeling for Information Systems. In *Modelling to Program (M2P)*. Springer, 113–126. https://doi.org/10.1007/978-3-030-72696-6_5

[39] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15).* SciTePress, 19–31. https://doi.org/10.5220/0005225000190031

[40] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (2004), 64–72. https://doi.org/10.1109/MC.2004.172

[41] Matthew Hause et al. 2006. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, Vol. 9. 1–12.

[42] Malte Heithoff, Alexander Hellwig, Judith Michael, and Bernhard Rumpe. 2023. Digital Twins for Sustainable Software Systems. In *IEEE/ACM 7th Int. Workshop on Green And Sustainable Software (GREENS).* IEEE, 19–23. https://doi.org/10.1109/GREENS59328.2023.00010

[43] Malte Heithoff, Nico Jansen, Jörg Christian Kirchhof, Judith Michael, Florian Rademacher, and Bernhard Rumpe. 2023. Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In *16th ACM SIGPLAN Int. Conf. on Software Language Engineering (SLE 2023).* ACM, 194–207. https://doi.org/10.1145/3623476.3623527

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

[45] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures* 54 (2018), 386–405. https://doi.org/10.1016/j.cl.2018.08.002

[46] Nico Jansen and Bernhard Rumpe. 2023. Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages. In *16th ACM SIGPLAN Int. Conf. on Software Language Engineering (SLE 2023).* ACM, 163–168. https://doi.org/10.1145/3623476.3623530

[47] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2009. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09) (Techreport B-108).* Helsinki School of Economics, 7–13.

[48] Djamel Eddine Khelladi, Benoit Combemale, Mathieu Acher, and Olivier Barais. 2020. On the power of abstraction: a model-driven co-evolution approach of software code. In *ACM/IEEE 42nd Int. Conf. on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20).* ACM, 85–88. https://doi.org/10.1145/3377816.3381727

[49] Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. 1991. *The Art of the Metaobject Protocol.* MIT press. https://doi.org/10.7551/mitpress/1405.001.0001

[50] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. 2020. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems.* ACM, 90–101. https://doi.org/10.1145/3365438.3410941

[51] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Pearson Education.

[52] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures* 46 (2016), 206–235. https://doi.org/10.1016/j.cl.2016.09.004

[53] Josh Mengerink, Alexander Serebrenik, Ramon Schiffelers, and M. Brand. 2016. A Complete Operator Library for DSL Evolution Specification. 144–154. https://doi.org/10.1109/ICSME.2016.32

[54] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* 37, 4 (dec 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[55] Judith Michael, Jörg Blankenbach, Jan Derksen, Berit Finklenburg, Raul Fuentes, Thomas Gries, Sepehr Hendiani, Stefan Herlé, Stefan Hesseler, Magdalena Kimm, Jörg Christian Kirchhof, Bernhard Rumpe, Holger Schüttrumpf, and Grit Walther. 2024. Integrating models of civil structures in digital twins: State-of-the-Art and challenges. *Journal of Infrastructure Intelligence and Resilience* 3, 3 (2024). https://doi.org/10.1016/j.iintel.2024.100100

[56] Judith Michael, Dominik Bork, Manuel Wimmer, and Heinrich C. Mayr. 2024. Quo Vadis Modeling? Findings of a Community Survey, an Adhoc Bibliometric Analysis, and Expert Interviews on Data, Process, and Software Modeling. *Software and Systems Modeling* 23, 1 (2024), 7–28. https://doi.org/10.1007/s10270-023-01128-y

[57] Judith Michael and Bernhard Rumpe. 2024. Software Languages for Assistive Systems. *SSRN* (2024). https://doi.org/10.2139/ssrn.4423849

[58] Judith Michael, Bernhard Rumpe, and Simon Varga. 2020. Human Behavior, Goals and Model-Driven Software Engineering for Assistive Systems. In *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, Vol. 2628. CEUR Workshop Proceedings, 11–18.

[59] OMG MOF. 2002. OMG Meta Object Facility (MOF) Specification v1. 4.

[60] Ernst Oberortner, Uwe Zdun, Schahram Dustdar, Agnieszka Betkowska Cavalcante, and Marek Tluczek. 2010. Supporting the evolution of model-driven service-oriented systems: A case study on QoS-aware process-driven SOAs. In *IEEE Int. Conf. on Service-Oriented Computing*

and Applications (SOCA). 1–4. https://doi.org/10.1109/SOCA.2010.5707172

[61] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML), Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF [Online; accessed 2024-06-05].

[62] Object Management Group. 2019. OMG Systems Modeling Language (OMG SysML), Version 1.6. https://www.omg.org/spec/SysML/1.6/PDF [Online; accessed 2024-06-05].

[63] Object Management Group. 2023. Kernel Modeling Language (KerML), Version 1.0 Beta 1. https://www.omg.org/spec/KerML/1.0/Beta1/PDF [Online; accessed 2024-06-05].

[64] Object Management Group. 2023. OMG Systems Modeling Language (OMG SysML), Version 2.0 Beta 1. https://www.omg.org/spec/SysML/2.0/Beta1/Language/PDF [Online; accessed 2024-06-05].

[65] Ana Luísa Ramos, José Vasconcelos Ferreira, and Jaume Barceló. 2011. Model-Based Systems Engineering: An Emerging Approach for Modern Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 1 (2011), 101–111. https://doi.org/10.1109/TSMCC.2011.2106495

[66] Cosmina-Cristina Raţiu, Wesley K. G. Assunção, Edvin Herac, Rainer Haas, Christophe Lauwerys, and Alexander Egyed. 2024. Using reactive links to propagate changes across engineering models. *Software and Systems Modeling* (2024). https://doi.org/10.1007/s10270-024-01186-w

[67] Ed Seidewitz. 2020. On a Metasemantic Protocol for Modeling Language Extension. In *MODELSWARD*. 465–472. https://doi.org/10.5220/0009181604650472

[68] Bran Selic. 2003. The Pragmatics of Model-Driven Development. *IEEE Software* 20, 5 (2003), 19–25. https://doi.org/10.1109/MS.2003.1231146

[69] Bran Selic. 2007. A Systematic Approach to Domain-Specific Language Design Using UML. *ISORC 2007*, 2–9. https://doi.org/10.1109/ISORC.2007.10

[70] Ben Shneiderman, Maxine Cohen, Steven Jacobs, Catherine Plaisant, Nicholas Diakopoulos, and Niklas Elmqvist. 2017. *Designing the User Interface Strategies for Effective Human-Computer Interaction, Global Edition*. Pearson Deutschland. 624 pages.

[71] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education.

[72] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. 2021. Composition of Languages, Models, and Analyses. In *Composing Model-Based Analysis Tools*, Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen (Ed.). Springer, 45–70. https://doi.org/10.1007/978-3-030-81915-6_4

[73] David Thomas and Andrew Hunt. 2019. *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional.

[74] Guido Van Rossum et al. 2007. Python Programming Language. In *USENIX annual technical conference*, Vol. 41. Santa Clara, CA, 1–36.

[75] Ennio Visconti, Christos Tsigkanos, Zhenjiang Hu, and Carlo Ghezzi. 2021. Model-driven engineering city spaces via bidirectional model transformations. *Software and Systems Modeling* 20, 6 (2021), 2003–2022. https://doi.org/10.1007/s10270-020-00851-0

[76] Eelco Visser. 2008. WebDSL: A Case Study in Domain-Specific Language Engineering. *Generative and Transformational Techniques in Software Engineering II: Int. Summer School (GTTSE 2007)* (2008), 291–373. https://doi.org/10.1007/978-3-540-88643-3_7

[77] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

[78] Markus Völter and Eelco Visser. 2010. Language Extension and Composition with Language Workbenches. In *ACM Int. Conf. companion on Object oriented programming systems languages and applications companion*. 301–304. https://doi.org/10.1145/1869542.1869623

[79] Dennis Leroy Wigand, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. 2017. Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics* (2017).

[80] Niklaus Wirth. 1996. Extended Backus-Naur Form (EBNF). *ISO/IEC* 14977, 2996 (1996), 2–21.

Received 2024-06-18; accepted 2024-08-15