A Model-Driven Approach to Design, Generation, and Deployment of GUI Component Libraries

Arkadii Gerasimov
gerasimov@se-rwth.de
Software Engineering, RWTH Aachen University
Aachen, Germany

Nico Jansen
jansen@se-rwth.de
Software Engineering, RWTH Aachen University
Aachen, Germany

Judith Michael
michael@se-rwth.de
Software Engineering, RWTH Aachen University
Aachen, Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering, RWTH Aachen University
Aachen, Germany

Sebastian Will
will@se-rwth.de
Software Engineering, RWTH Aachen University
Aachen, Germany

Abstract

The reusability of modular, embeddable components is a key determinant for the success of modern programming languages to ensure efficient and high-quality development. However, there is a gap in the field of domain-specific modeling regarding reusable components at the model level. While libraries are relatively common and a de facto standard for prominent programming languages, establishing model libraries is still in its infancy. This paper specifies building a model-driven component library that utilizes a self-extension mechanism. We demonstrate an approach to structure, build, and integrate such a library using a wide range of GUI components, from essential atomic elements and more complex composed components to specifically tailored ones for particular application domains. Employing such libraries at the model level further supports the goal of model-driven engineering to assist domain experts in efficiently building high-quality systems.

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, Bernhard Rumpe, and Sebastian Will. 2025. A Model-Driven Approach to Design, Generation, and Deployment of GUI Component Libraries. In *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering (SLE '25), June 12–13, 2025, Koblenz, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3732771.3742713

1 Introduction

Domain-Specific Languages (DSLs) are languages tailored for domains using concepts commonly known to domain experts [14, 34, 42, 68]. The use of DSLs in Model-Driven Software Engineering (MDSE) [66] enables developers to be less bound to the underlying implementation technology [48] and using domain-specific concepts, being much closer to the problem domain, when increasing the automation in software engineering processes. While MDSE is applied in practice [29, 49, 68], *e.g.*, because of its technical merit [37], it may become difficult to make changes to DSLs and their tooling over time [61] and modeling has to be fully integrable in code-centric environments [29]. Like any software product, the maintenance and support efforts for DSLs also increase as they evolve [28]. The success of DSLs also depends on finding a good balance between domain-specificity and generality and a proper scope when defining a DSL [62].

To mitigate some of these challenges, reusing existing language components has become a focus in DSL research and development [10]. Languages are no longer developed from scratch but constructed using modular, reusable blocks. Research and practice have suggested composition techniques [27, 55, 67] and design patterns [24] to facilitate the seamless integration of language components provided by libraries [13], enabling the adoption of software language product lines [21, 41]. These advancements at the language level are essential for language development and maintenance. In addition, extensibility at the model level becomes important [43], *e.g.,* Gerasimov et al. [32] have explored a self-extension mechanism for DSLs that allows the inclusion of model libraries within DSLs. However, such DSLs rely on model libraries to be as expressive as possible, *e.g.,* when generating web applications with the related MDSE tooling.

Graphical User Interfaces (GUIs) play a crucial role in web development, allowing users to interact with the system. As systems become more complex, the difficulty of creating a high-quality GUI increases [25]. To address this problem, MDSE tooling has been applied to support the process by using models to describe the GUI [26]. The approaches use techniques for modeling GUIs, which include specifying GUI components in their DSL and using those to build GUIs [2], using a flexible DSL that allows for the creation of custom GUI components [9, 64], or hybrid approaches with real-time modeling and rendering capabilies [46].

This paper aims to show a method to make it easier for modelers to create system specifications without using a platform-specific frontend development language. Our main contributions are an extensible GUI component library and a method to deploy and integrate a model library within the software engineering process up to system deployment. We discuss the results in several case studies, *e.g.,* for creating Digital Twin (DT) cockpits and several experimental applications for different domains.

The paper is structured as follows: The next section provides relevant background. Section 3 introduces the GUI model library, and Section 4 describes its developmnet, deployment, and integration. Section 5 presents case studies for applying the model library in different application domains. Section 6 discusses our approach and Section 7 discusses related works. The last section concludes.

2 Background

This section introduces the basics of MontiCore and Domain-Specific Language for Graphical User Interfaces (GUI DSL), a language used in MontiGem to model and generate GUIs.

2.1 DSLs & Language Workbench

DSLs [14, 42, 68] are an alternative or extension of general-purpose languages like Java. A DSL focuses on a domain, working on a higher abstraction level using a notation standard for the domain. This results in a software development process that performs better in productivity and quality, at the cost of a longer start-up time to create such a DSL [54]. For that reason, language workbenches such as EMF [52], GEMOC Studio [22], MetaEdit+ [57], MontiCore [36], Neverlang [59], Rascal [60], Spoofax [39], or Xtext [7] have been introduced to support and speed up language engineering.

Our approach uses MontiCore language workbench. MontiCore grammars define the context-free syntax of a DSL in an EBNF-like format that supports DSL composition, extension, and aggregation [36]. MontiCore is a meta-tool for generating infrastructure from a grammar file. This includes a model parser for the concrete syntax to be converted into an Abstract Syntax Tree (AST), and tooling such as visitors for AST traversal, assisting in developing generators.

2.2 Domain-Specific Language for Graphical User Interfaces

GUI DSL [33] is a language for modeling graphical user interfaces. It is used in the MontiGem framework [1, 11] - a tool built with MontiCore for generating web applications. The MontiGem framework has been applied to create web applications for different domains, such as resource management [31], process-aware information systems [23], low-code development platforms [18], digital twins [6], assistive systems [32], and IoT app stores [12].

GUI DSL
```
package montigem;
component Icon(String name)
```

Listing 1. GUI model: Icon model declaration

GUI DSL
```
package myapp;
import montigem.Icon;
component HomeIcon() {
  @Icon(name = "home");
}
```

Listing 2. GUI model: Home icon model declaration and icon component usage

Using MontiGem, system-specification models are transformed into Java, TypeScript, HTML, and SCSS code for an Angular application. Each GUI DSL model declares a single GUI component by specifying the qualified name of the component and its input parameters. The model optionally implements the component by importing and using other components. Otherwise, the hand-written implementation extends or overwrites the generated code. Listing 1 shows a component declaration without implementation, and Listing 2 shows its usage to create a `HomeIcon` component.

GUI DSL defines concepts for specifying and using components and their parameters. The language and the MontiGem generator do not distinguish between different components. Instead, a component's semantics are specified by its implementation. For example, an icon implementation may use, *e.g.,* Angular Material icons, or have a custom implementation. Such an approach to specifying GUI components has disadvantages in generating component-specific code, but is a better fit for building model-driven component libraries. This way of adding concepts using models without extending the DSL's grammar is called a self-extension mechanism [32].

To handle various use cases using a few syntactical constructs, GUI DSL provides variations for component declarations and different types of component parameters. For example, a component may be declared a regular or page component. The MontiGem generator creates routing configurations for pages in addition to component code. Input parameter may be declared with *e.g.,* built-in `Event` type for which the MontiGem creates event handlers and produces specialized syntax. Other types are discussed in Section 3.

All input parameters are optional; regardless of whether parameter values are specified, GUI is always functioning. This ensures a robust GUI following Postel's Law [70]. If a parameter expects some value, such as data for a chart, but the value is missing, a placeholder is shown without affecting the rest of the interface. Using observers, the MontiGem keeps values up-to-date and updates the GUI whenever a parameter value changes. For example, if the chart data is loaded later, the chart will be shown in the GUI automatically.

3 GUI Component Model Library

Building a library in a model-driven context using a self-extension mechanism handles stages typical to a general library lifecycle, such as creation, development, maintenance, deployment, integration, and usage. We describe the stages addressing the challenges of the model-driven aspect.

The stages involve the following roles:

- A **library developer** creates or derives reusable GUI components from applications.
- An **application developer** designs the GUI by creating application-specific GUI components using library components or from scratch.
- A **component developer** is either a library or an application developer.
- A **user** interacts with the GUI, for example, by clicking buttons, reading or typing in information, etc.

To establish a model-driven GUI component library, we consider the following points:

- Identifying reusable GUI components from use cases to create and grow a library. The library provides application developers the components necessary to build a system's GUI. The components vary from basic building blocks like buttons and text to complex domain-specific components like electric circuit diagrams or molecular structures.
- Making sure the integrated library does not bloat the size of the end product. During system deployment, the imported library may include unused components, negatively impacting user experience with the system's loading times. This problem can be partially mitigated by splitting a library into smaller chunks. This concern comes after a library is established, but we address it while describing the smaller library parts, further called *sub-libraries*, for simplicity.

We identify reusable GUI components, select and describe components with unique traits in an MDSE context, define a method for structuring, developing, deploying, and integrating the library, and identify trade-offs of the used approach.

Figure 1 shows an overview of the GUI component library explained in the following sub-sections. Reusable GUI components are identified from their use cases: whenever a component can be used in systems from different domains, it becomes a candidate for library integration. The library is split into sub-libraries of related components, where the structural complexity, functionality, and domain define which components are related. For example, text and image are atomic, basic, domain-independent components. An input field or checkbox is also an atomic and domain-independent component. Still, an input field's functionality is tied to manipulating an object state, like entering a person's name.

A sub-library may reuse components of other sub-libraries. For example, charts use buttons and text from the basic components sub-library. A component can reuse components of the same library, such as a card that consists of rows and columns. Some components are constructed from multiple sub-components, like the table, to form a cohesive structure.

Dividing a library into sub-libraries is unimportant for the system's modeling, but improves the application developer and user experience. The components are loaded in packages, which reduces the likelihood of unused components being loaded during application development and usage.

3.1 Basic Components

Basic components include domain-independent atomic GUI components for simple interaction, such as text, buttons, image, video, etc. "Atomic" means that, from an application developer's perspective, the component implementation does not require the composition of other components. For example, a video component has several buttons and an area for displaying a video. Still, a component developer may use a predefined HTML `video` element, which is perceived as one entity. "Simple" interaction is defined by component usage. The simplest GUIs are used for consuming information, provided by text, video, or audio, and most pages also have buttons and links for navigation.

GUI DSL

```
package montigem;
component Button(
  String height = "auto",
  String width = "auto",
  String label = "",
  List<GUIViewElement> content,
  boolean disabled = false,
  Event<Void> leftClick
)
```

Listing 3. GUI model: Button declaration

[GJM+25] A. Gerasimov, N. Jansen, J. Michael, B. Rumpe, S. Will:
A Model-Driven Approach to Design, Generation, and Deployment of GUI Component Libraries.
In: Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, pp. 57-70, SLE '25,
ISBN 979-8-4007-1884-7, DOI 10.1145/3732771.3742713, ACM, Jun. 2025.

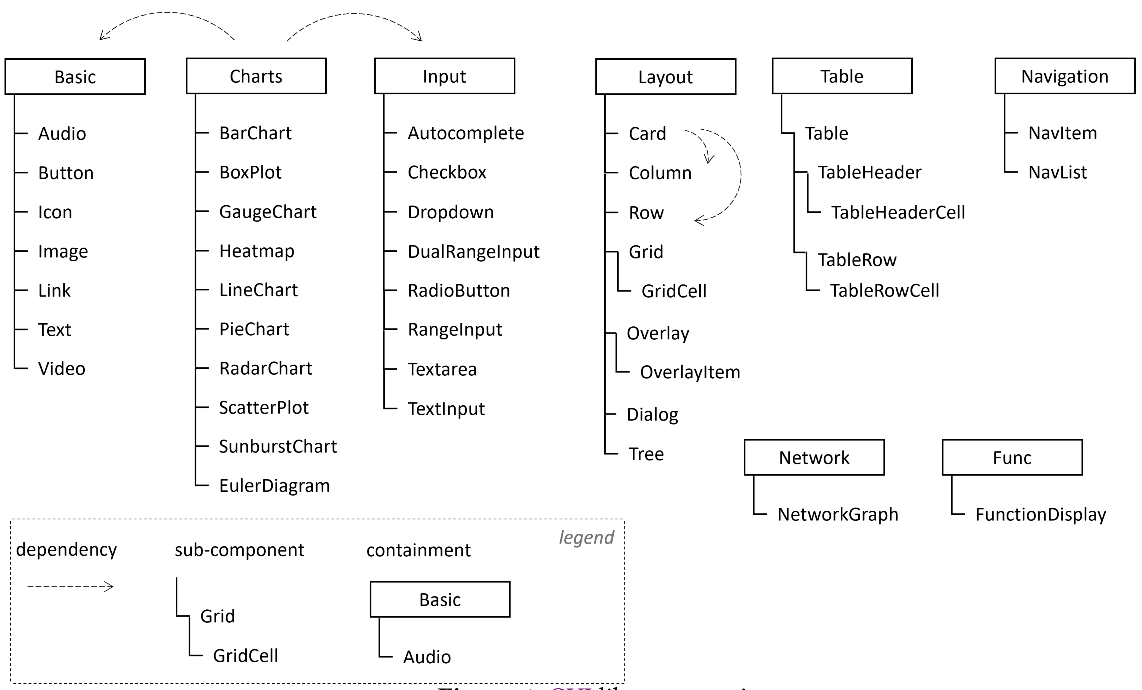


Figure 1. GUI library overview

An example of a basic component is a button. A button enables active user interactions through mouse clicks, key presses, or similar, depending on the system a user operates on (*e.g.*, taps on devices with a touch screen). The component specifies input parameters:

- The component style (Listing 3, l. 3-4). The width and height parameters specify the visual dimensions of the component. They are common for many components and will be omitted in the examples. The components use the width and height values to set corresponding CSS attributes, thus the values can be specified as a `String` in pixels, percents, with CSS keywords, etc.
- The button text (Listing 3, l. 5-6). If the label contains plain text, the `label` parameter is set. If multiple components, such as an icon and some text, should be shown, the `content` is set. It uses a type `GUIViewElement` that enables component nesting.
- The button state (Listing 3, l. 7). In this case, the `disabled` parameter indicates whether a user can interact with the button.
- A callback for an event (Listing 3, l. 8). The MontiGem recognizes `leftClick` parameter type `Event` and produces code for the event handling, for example, transmitting data on button click.

The sign-up form model in Listing 4 and its visual representation on Figure 2 show an example of GUI components' usage. The button (Listing 4, l. 16-19) specifies a function for signing a person up. The function is declared and implemented in a hand-written extension of the `SignUpForm`.

GUI DSL

```
package myapp;
import montigem.*;
component SignUpForm(Person p) {
  @Column(rowGap = "10px",
    components = [
      @TextInput(
        labelText = "Name:",
        placeholder = "Alice",
        entry = p.name
      ),
      @TextInput(
        inputType = "date",
        labelText = "Birth date:",
        entry = p.birthdate
      ),
      @Button(
        label = "Confirm",
        leftClick = signUp(p)
      )
    ]);}
```

Listing 4. GUI model: Sign-up form

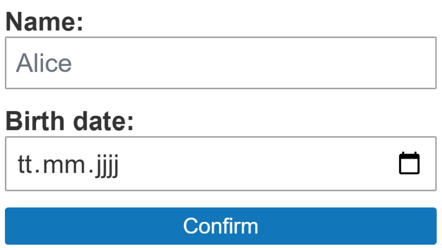

Name:
Alice
Birth date:
tt.mm.jjjj
Confirm

Figure 2. Rendered GUI: Sign-up form

3.2 Input Components

Input components are widely used in interactive systems where users must provide data, *e.g.*,, to create or update objects, form submissions, search functionalities, etc.

GUI DSL

```
1  package montigem;
2  component TextInput(
3    String labelText = "",
4    String placeholder = "",
5    String inputType = "text",
6    ?String entry,
7    TextInputValidity validity,
8    Event keyRelease,
9    Event keyPress
10 )
```

Listing 5. GUI model: Text input declaration

The text input example in Figure 2 enables users to input their name and birth date. Listing 5 shows the signature declaration of the `TextInput`. The component comprises an input field and a label to provide context for the expected input. The `inputType` changes the component's appearance and behavior. For example, the input type "date" shows an input in which a user can select a date using a calendar.

The main feature of input components is synchronizing an input parameter with a variable and displaying validation results. The parameter `entry` references the input field's value. The question mark before the type (Listing 5, l. 6) indicates that the parameter is linked to a variable. For example, variable `p.name` (Listing 4, l. 9) is updated whenever a user enters a new value into the "Name" field. The `validity` parameter can be set to validate user inputs. For example, if a user types an invalid date, the field is highlighted with a red color, an icon, and help text.

3.3 Layout Components

Layout components are domain-independent components serving as containers for other components. They ensure the components are shown in a column, grid, dialog window, etc. There are different ways to specify layout, such as using a grid, coordinates, etc. MontiGem provides a library based on CSS grid [17] and flexbox [16] layout.

GUI DSL

```
1  package montigem;
2  component Column(
3    String hAlign = "stretch",
4    String vAlign = "stretch",
5    String rowGap = "0",
6    String colGap = "0",
7    String wrap = "nowrap",
8    List<GUIViewElement> components
9  )
```

Listing 6. GUI model: Column declaration

A column is an example of a layout component (see Listing 6). The inputs of a layout component primarily consist of the layout configuration, such as horizontal and vertical alignment of nested elements (Listing 6, l. 3-4), gaps between components (Listing 6, l. 5-6), and whenever the components are wrapped to the next column when there is not enough space (Listing 6, l. 7). A `components` parameter specifies components to be arranged in a column. The parameters with types indicating a single or multiple `GUIViewElement` objects are handled by MontiGem separately to nest the components in the target code properly. Such parameters are the main feature of a layout sub-library component.

Listing 4 uses the `column` component to vertically stack input fields and the button. Any component can be used as input, which allows an application developer to nest layout components and create complex web pages. However, such a container may not affect components with a coordinate-based positioning, like context menus and dialog windows.

3.4 Charts

Charts are structurally complex components that represent statistical data. They are used in financial applications, marketing, cyber-physical systems, IoT data, and other domains handling large data volumes that require an overview. MontiGem's charts library is based on existing implementations of commonly used charts, such as pie, bar, and line charts.

GUI DSL

```
package montigem;
import montigem.LineChartTypes.LineChartEntry;
component LineChart(
  List<LineChartEntry> entries,
  boolean enableBackgroundColor,
  Integer maxValue,
  Integer minValue
)
```

Listing 7. GUI model: Line chart declaration

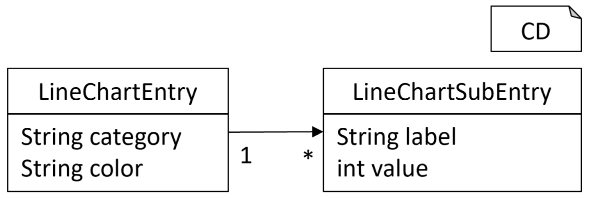


Figure 3. Class diagram: Line chart types declaration

An example is a line chart (Listing 7). Most input parameters are used to configure the chart, *e.g.,* setting minimum and maximum values of the y-axis. The `entries` parameter represents input statistical data and has a more complex structure. Its type is a list of `GemLineChartEntry` objects, each describing a line on a line chart. Figure 4 shows an example of a line chart. The blue line is represented by an entry specifying the blue color, category name, "Motor Temp.", and the list of points specifying values for different timestamps.

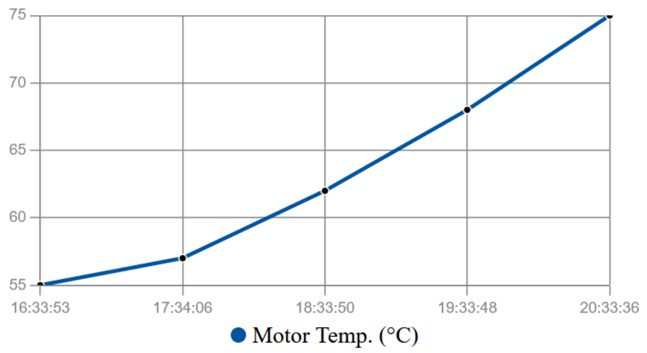
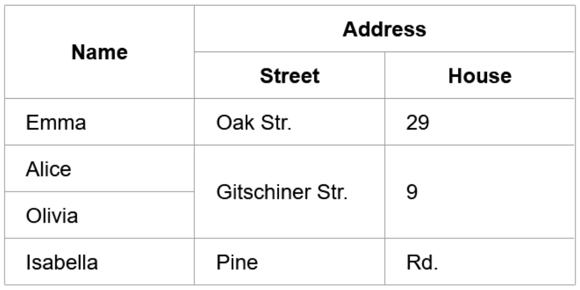


Figure 4. Rendered GUI: Line chart usage

Complex data types are a common trait of the charts since they visualize aggregated statistics on a dataset. To support complex types, GUI DSL enables importing and using types from compiled Java classes and interfaces or from MontiCore-specific class diagram definitions. In a MontiGem application, the data is aggregated and sent to GUI from the server.

3.5 Composed Components

MontiGem has a few GUI components that rely on the composition of several specific parts, such as the navigation and table components. These components are categorized in separate sub-libraries since they have different functionality. We use the table component to describe the composition trait.



Name	Address	
	Street	House
Emma	Oak Str.	29
Alice	Gitschiner Str.	9
Olivia		
Isabella	Pine	Rd.

Figure 5. Rendered GUI: Table usage

GUI DSL

```
package montigem;
component Table(
  List<GUIViewElement> headers,
  List<GUIViewElement> body
)
```

Listing 8. GUI model: Table declaration

The table component has several sub-components to specify a header and a body. A table organizes and displays text or other GUI elements in rows and columns. Listing 8 demonstrates the declaration of the table component in MontiGem. The header is defined by a list of other GUI elements (`headers`), same as the body. `Headers` are set to a list of table header components and body to a list of table rows. The header is visually distinguishable from the body and defines the width of the columns.

GUI DSL

```
package montigem;
component TableRow(
  List<String> cols,
  List<GUIViewElement> cells
)
```

Listing 9. GUI model: Table row declaration

Each row consists of as many cells as the table has columns. Listing 9 shows the declaration of a table row. If a table only displays text, the `cols` parameters can be used to specify the text for each column. For complex table elements, `cells` are used. The cells may span multiple columns and rows and contain GUI elements, such as buttons or text inputs.

Composed structures provide a high level of customization. The sub-components are not restricted to a specific type. For example, a modeler may specify custom sub-components to replace predefined `TableRow`. However, application developers are not prevented from making unintentional mistakes, such as providing unfitting components as input.

3.6 Domain-Specific Components

Some reusable domain-specific components were recognized while developing several projects in digital twins, chatbots, and artifact analysis domains. These include components:

- A status indicator like a traffic light.
- A network graph for viewing dependencies between objects. For example, it displays dependencies between software artifacts across git repositories.
- A function display component for showing component and connector architectures, currently used in teaching to visualize a factory simulation.
- A chat component built for communication between application users or between a user and a chatbot.

The chat and status indicator components are being integrated into the library at the time of writing. Each component is packaged in separate libraries since they are added to projects with specific use cases.

Status indicators are used in real-time monitoring and decision-making applications, such as digital twin dashboards for automated industrial systems. A traffic light indicates whether a service has stopped ("red"), is not functioning properly ("yellow"), or is operating ("green"). Status indicators give immediate feedback to the user, enabling an intuitive visualization of operational conditions.

Fundamentally, domain-specific components are similar to domain-independent components and utilize the same GUI DSL features. The status indicator is similar to a basic component, such as an icon, but it has additional features to switch between various states. The graph component is essentially a chart since it visualizes aggregated data. The function display is a composition of GUI components for displaying software, hardware components, and connectors.

The chat component combines input and layout components' features to enable user input to be entered and displayed.

4 Development, Deployment, and Integration of the Model Library

We describe the steps for building, deploying, and integrating a model library with the generated code into a system (see Figure 6). Further, we explain the process using the GUI component library.

4.1 Classification

The process starts with identifying and classifying a reusable GUI component. This step ensures the library's modular structure, simplifies component maintenance and deployment, and tracks dependencies by handling component groups instead of individual components. A reusable GUI component is distilled from use cases in which an application developer identifies repeated usage of the same GUI part. Given a reusable component, it is categorized into the library according to its traits: functionality, complexity, and application domain. The component is either assigned to an existing sub-library or put in a new one.

Consider adding an icon component to the library (Figure 7). The existing sub-library for simple components with basic interaction that fit an arbitrary domain contains text, image, and button components. Since the icon component does not possess new traits, a library developer puts it into the existing sub-library. When a `FunctionDisplay` component is integrated, a library developer creates a new library for the C&C architecture domain. The component consists of elements describing system parts, thus categorized as complex, and showing basic statistics.

The area assigned to a sub-library is chosen mainly according to the balance between maintenance complexity and package size. For example, each component could be managed in a separate sub-library, but that would require more careful dependency management. A separate sub-library for a component may also be insignificant regarding package size, for example, for tiny components like text or images.

New components are generally created according to research, experimental, and industrial project requirements. When library developers identify repeated use of a component, it is added to a domain-specific library corresponding to the project's domain or, if applicable, to a domain-independent sub-library. The extent of fitting the domain is derived from use cases for a component. For example, a status indicator for digital twins is currently used in two MontiGem applications in the digital twin domain out of numerous other applications. If a component is being more actively used in other domains, the component may be moved to a different sub-library fitting a more general domain.

4.2 Library Deployment and Integration Environment

The library must be set up before the reusable components can be integrated. The library provides an environment that manages GUI components' implementations in the form of models and corresponding code.

Figure 8 shows our library deployment and integration setup as an example.

- Models describe GUI component API and partial implementation processed by a generator producing Angular component code (TypeScript, HTML, CSS). Generally, any reusable models and corresponding target code could be library sources. For example, in JetBrains MPS, it could be one or several reusable solutions, possibly coupled with languages into plugins.
- When the library is deployed, the component code in the target language is compiled and packaged as `npm` packages for each sub-library. All models are processed by MontiGem and converted to JSON files describing exported symbols of GUI components, *e.g.,* the library's public API for application models. The symbols are packaged into a JAR file. Gradle manages the deployment process. It delegates target code packaging to Angular CLI and model conversion to MontiGem. In a different context, such as MPS, the process could be managed by a custom build with a plugin to compile and package target code and the corresponding solution, possibly with an extended Ant build script and plugins.
- On the side of a MontiGem application, a MontiGem gradle plugin includes the JAR file and a default subset of `npm` packages, which it provides for domain-independent sub-libraries. An application developer may override the default package set by providing a configuration file. Whenever an application's GUI model uses a library component, the generated code imports use a corresponding component in the target code. The MontiGem generator specifies how to map a library component import and usage in the model to the target code. In other words, the application side requires an integration infrastructure as a counterpart of the library deployment. For example, in MPS, it could be a plugin for integrating an existing solution.
- The library is published by a GitLab Continuous Integration (CI) pipeline.

The setup includes multiple configuration files required by the technology stack. For example, Angular JSON files configure libraries and applications, `package.json` files configure `npm` packages, Gradle scripts orchestrate library deployment and integration, etc. This environment is predefined for a MontiGem application. An application developer may adjust it by providing configuration files to override the defaults.

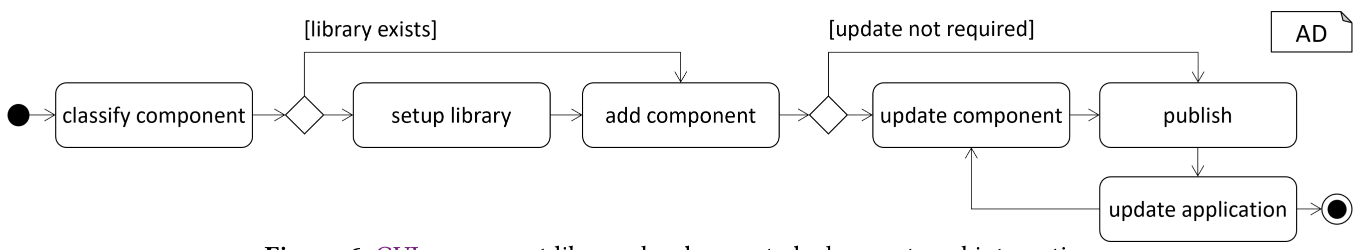


Figure 6. GUI component library development, deployment, and integration

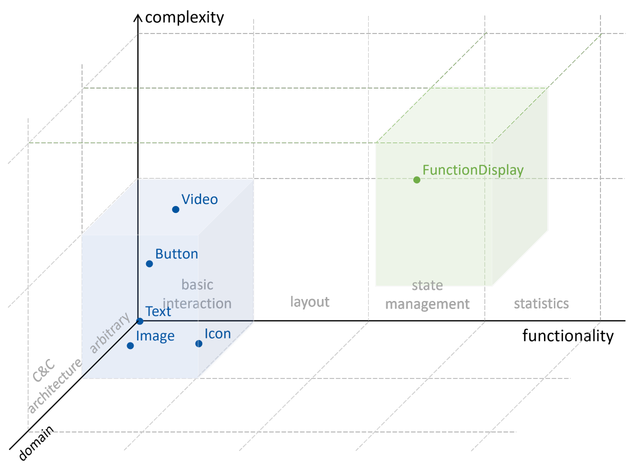


Figure 7. GUI component classification

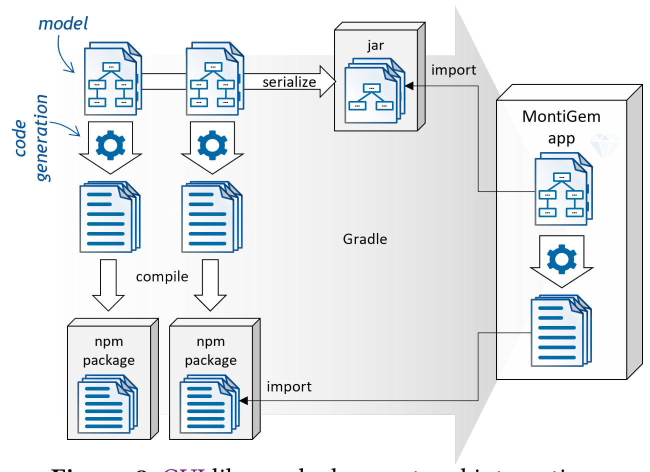


Figure 8. GUI library deployment and integration

Abstracting away from specific technologies, a model library based on a self-extension mechanism requires a realization of the following concepts:

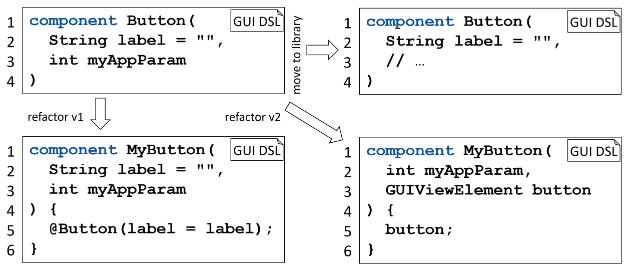
- Model packaging mechanism.
- Target code packaging mechanism.
- Mapping from model import and usage to the corresponding import and usage in the target code.
- A coordinator for integrating the model and target code packages into the end product.

Depending on the technology stack, the setup realization may vary. For example, models written in the internal DSL of a target code language could use the packaging mechanism of the target code. The coordinator may use a code generator to load dependencies based on imports in application models.

4.3 Integrating Components into the Library

Reusable components can be integrated after the library is set up (see Figure 6). Adding a component to an existing sub-library involves adding its model and code to the corresponding sub-library parts. Refactoring steps may be necessary if the library environment differs from the application. For example, generated MontiGem applications use path aliases for import paths, which should be avoided in an Angular library. The MontiGem generator realizes a library mode — the import statements are generated differently depending on the environment. However, hand-written code extensions may require similar manual adjustments.

A library developer must consider simplifying a component's model before publishing it, *e.g.*, reducing input parameters and naming them according to the library's standards. Since the model describes a component's public API, changes involving renaming or removing component parameters introduce breaking changes for library applications. Changes made to the API in the model should also be propagated to the target code. For example, if hand-written extensions of application code use input parameters of a hand-written library component extension that is missing in the component's model, refactoring hand-written code to the application models becomes difficult.



```
component Button(
  String label = "",
  int myAppParam
)
```

```
component Button(
  String label = "",
  // ...
)
```

```
component MyButton(
  String label = "",
  int myAppParam
) {
  @Button(label = label);
}
```

```
component MyButton(
  int myAppParam,
  GUIViewElement button
) {
  button;
}
```

Figure 9. Application GUI component refactoring

The (sub-)library version is incremented at the end, and the library can be published. The library component can then replace the original GUI component in applications from which it was derived. Generally, the replacement needs refactoring if changes were applied to the component during library integration. Figure 9 shows two refactoring strategies.

After moving a button to the GUI component library, the component loses the application-specific parameter `myAppParam`. The refactoring option `v1` suggests creating a wrapper GUI component with application-specific parameters. The component delegates the library component parameters if necessary. The refactoring option `v2` builds a wrapper that accepts a library component instance. Both solutions have disadvantages; the first requires duplicate parameter declarations, and the second allows arbitrary components. Moving a component to a different sub-library does not affect existing MontiGem applications since specific imports in the target code are abstracted away at the model level.

After the library component integration, an application developer may request new features, and the component development enters a loop. In case a breaking change is introduced, a migration mechanism should be supplied for models. The current MontiGem migration strategy is limited to simple scripts. More advanced techniques are possible, but outside this work's scope.

5 Case Studies

We applied MontiGem to create various web-based applications and conducted several experiments to evaluate the tooling. Gerasimov et al. [32] have already compared library-based development to traditional DSLs specifying GUI components on the language level. We analyze the effectiveness of built infrastructure based on a model library by comparing it to code-first development.

5.1 Digital Twin Cockpit

The presented GUI components library can be used to realize a GUI for a website by applying a model-driven approach. It takes the work of recreating GUI view elements from scratch and instead offers a large selection of predefined components to integrate into a system quickly. These range from essential components useful for almost every website, such as navigation, basic information display, and user input, to domain-specific components. This section describes engineering a GUI for the particular domain of digital twins, also called digital twin cockpit according to [6]. An example of such a GUI is shown in Figure 10.

In the digital twins domain, there are many sub-domains such as digital smart cities [8, 45], smart farming [4], manufacturing and production [35, 56, 65], energy systems and electricity networks [30, 44], oil and gas industry [20], supply chain management, clinical institutions [6, 19, 38], and more [47]. However, they all have common ideas. Therefore, these domains' applications can benefit from reusable GUI components. It also leverages the modularity of the GUI component library, only using components from sub-libraries relevant to the domain. Digital twin cockpits require a GUI capable of comprehensively visualizing large datasets, often capturing real-time information for monitoring. Using the table component is an intuitive approach for this task. Furthermore, displaying data in different charts supports data analysis, giving the user clues for predictions and planning.

A reference architecture for digital twins presented in [35] determines different parts of a digital twin using component and connector architecture models. Mapping these models onto the GUI DSL and class diagram models allows easy integration in a MontiGem project. It automatically generates services for backend communication and provides data access from the database. After the data layer is set up, we decide which web pages the application features. Creating a page with the keyword page will define a route for navigation. The types defined in the class diagram can be used as input parameters. A typical page is a content overview. It receives a list of services that the digital twin should monitor. Defining them as an input parameter of a page will automatically load the information from the database. After adding a dependency to the GUI component library, the desired components can be imported and used in the page, *e.g.*, `Table`, `LineChart`, `StatusIndicator`, or others. Furthermore, a page can be split into more granular components for either sub-parts of the page, *e.g.*, elements that should only be visible once their content is loaded, or complex elements like the table to display the observed values.

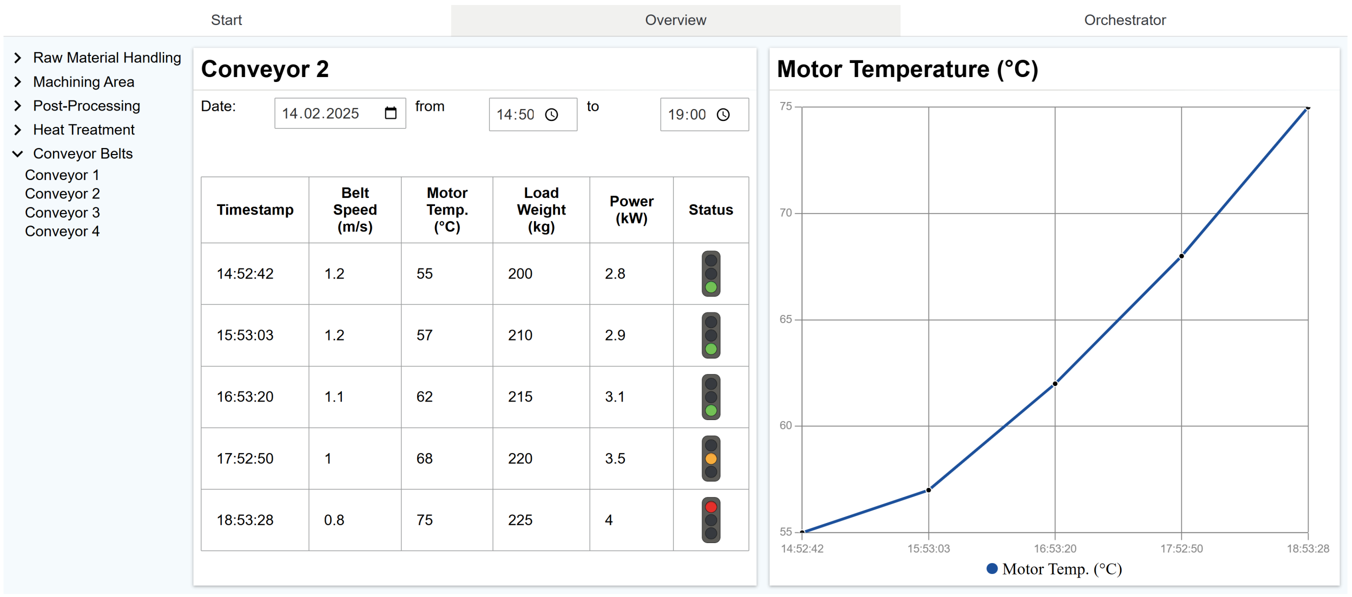
Figure 10 shows the described digital twin cockpit, reusing the domain-specific status indicator component demonstrated in Section 3. The table shows the evolution of the operational attributes of a machine, in this case, a conveyor belt, at different timestamps. The traffic light status changes once the "motor temperature" reaches a certain threshold. This is also visible in the line chart, allowing the machine's status to be monitored.

The digital twin cockpit application was written in 354 lines of code for the GUI models, with 267 additional lines of handwritten code to extend the GUI's functionality. The generated code in Angular, including TypeScript, HTML, and CSS, comprises 6967 lines of code. The generated Angular code represents an application without a model library; the model-driven approach requires about 90% less code.

5.2 Experimental Applications

We conducted an experiment in which 15 students organized into six groups (4 groups of 3, one group of 2, and one solo), selected a domain, and created two very similar applications for the domain: the first application was built using a technology stack of a group's choice, and the second was built with the MontiGem framework using the GUI component library (see Table 1). The developed applications:

- Inventory management application handling a list of wares and making purchases.
- Traffic management application handling train routes.
- Project management application handling issues in a software project, similar to GitHub.



Conveyor 2

Date: 14.02.2025 from 14:50 to 19:00

Timestamp	Belt Speed (m/s)	Motor Temp. (°C)	Load Weight (kg)	Power (kW)	Status
14:52:42	1.2	55	200	2.8	🟢
15:53:03	1.2	57	210	2.9	🟢
16:53:20	1.1	62	215	3.1	🟢
17:52:50	1	68	220	3.5	🟠
18:53:28	0.8	75	225	4	🔴

Figure 10. Digital twin cockpit

Table 1. Lines of code comparison: MontiGem vs. other frameworks

Technology	Statistic	Inventory M	Traffic M	Project M	Course M	Asset M	CRM
MontiGem	GUI LoC	1291	831	472	493	913	239
	HW LoC	1151	1652	435	404	985	279
	total LoC	2442	2483	907	897	1898	518
Own choice	**total LoC**	2898	3918	1246	2057	1578	1091
	GUI Tool	Django	Angular	React	React	Flutter	React

- Course management application handling creation and assignment to student courses.
- Asset management application handling cars in one application and devices, such as phones, in the other.
- Customer relationship management (CRM) application handling customer data, contacts, and deals.

The students had a week for planning the applications, *e.g.*, describing use cases and requirements, and a 4-week development time for each application. The quantitative analysis of the results is shown in Table 1. Developing a GUI for an application with MontiGem component library generally requires less code, for example, compared to `React` and `Angular` that require a moderate amount of boilerplate code. The amount of GUI model code and hand-written (HW) extensions within a MontiGem application was similar at the time of the experiment. However, as some participants reported, the MontiGem version of the application had fewer features. For example, the inventory management application's overview page did not have an update button that opens a 3-field form for updating product information. We surveyed the students (10 out of 15 participated) to analyze further the differences between using a model-driven approach and widely used frameworks.

Table 2. Survey participants' experience

Experience	<3m	3m-1y	1y-2y	>2y
Programming	1	1	1	7
Web dev	4	0	5	1
MDE	8	0	2	0

The participants' experience with technologies is shown in Table 2. Most participants had experience with programming (more than 2 years), more than half had moderate web development experience, and most were beginners in model-driven engineering (less than 3 months). The Table 3 shows questions related to GUI development. Development speed and experience are assessed in favor of the technology stack chosen by students. MontiGem's GUI development features were voted as slightly lacking compared to widely used frameworks. When asked the follow-up question, "What features did you miss most?" participants mentioned missing components: maps (Google Maps or similar) and drag-and-drop. Other missing features included out-of-the-box authentication and specific data request options for server-side data filtering. Such components will be added to the library.

Threats to validity The experiment started 10 months ago and ended 7 months before the writing of this paper.

Table 3. Comparing MontiGem and other frameworks. The numbers show the number of votes for the corresponding options.

It was possible to realize GUI within time limit		fully	mostly	partially	no
	MontiGem	1	5	4	0
	Own choice	3	5	2	0
GUI development with MontiGem was	longer	slightly longer	similar	slightly shorter	shorter
	4	1	4	1	0
Documentation was		sufficient		required support	insufficient
	MontiGem	0		1	9
	Own choice	4		6	0
How difficult was troubleshooting the errors		easy		manageable	difficult
	MontiGem	0		4	6
	Own choice	4		6	0

Since then, MontiGem and other frameworks have been updated, and MontiGem has addressed some issues. Its current version introduced major changes and was released shortly before the students used it. Thus, the documentation had to be created from scratch, and we are still improving it. The documentation provided to students had a short usage example and input parameters for each GUI component. This documentation will be generated from artifacts such as component declaration models and models describing more complex usage examples. A step-by-step tutorial and several example projects were added for students' training. Regarding IDE tooling, a language server protocol (LSP) was implemented for the GUI DSL and class diagram languages. The tooling is currently available and is being improved. Future support includes live updates of the application during model development, which is currently experimental. In addition, error logs are collected and analyzed by a centralized system. Error messages are being added and clarified to simplify the error resolution process for application developers.

The experiment started with all students working on an application using a technology stack of their choice first and MontiGem second. The students likely had a general idea of the application structure before building it with MontiGem, which could have given MontiGem an advantage.

The number of students and survey participants was low, and the students had similar backgrounds, favoring conventional tools. While some results, such as weaker documentation for MontiGem, are convincing, other results, such as GUI development time, could be different if the students had more experience with MDSE.

6 Discussion

The presented library creation, deployment, and integration method is similar to library management in traditional software systems. The model layer requires dependency handling through imports, packaging, and building tools parallel to code-level ones. Additionally, those tools must be coordinated to ensure intuitive library management. Providing such infrastructure enables model reuse across different projects. However, the infrastructure alone is not enough to provide a positive application developer experience.

6.1 Experimental Results

In general, applications created with MontiGem require less code than commonly used frameworks. Engineers with experience in model-driven engineering, as shown in Section 5.1, can build a GUI with less code using MontiGem and its component library. This suggests that the model-driven approach reduces the boilerplate code, accelerating development. However, the student survey does not entirely support this hypothesis. The lack of experience and knowledge about a model-driven framework, such as MontiGem, as well as the steep learning curve, plays a significant role in the development process. GUI models offer a good starting point for the development of web applications with the possibility of extending the functionality with hand-written code.

The survey results and interactions with the students indicate that the model-driven result with MontiGem, in its prior state, hardly competes with state-of-the-art GUI frameworks. However, the proposed approach could outperform such frameworks in domains that rely on components for specialized use cases. Although the knowledge distribution of the participants, which was more in favor of web development than MDSE, likely influences this result, it also indicates an important aspect: MDSE tools in themselves require a maturity comparable to modern programming frameworks. This includes comprehensive documentation, tooling support to guide the engineering process, and accessible tutorials considering different proficiency levels.

6.2 Limitations and Trade-offs

Our model-driven component library relies on target language supporting libraries, *e.g.,* providing the packaging and importing mechanisms, such as the Angular framework using its CLI tool, `webpack`, and `npm`. Additionally, a build tool must be able to use these mechanisms, in our example, Gradle delegates to the Angular CLI tool. While both conditions can be satisfied with custom-built tools when necessary, building such tools requires additional resources.

A generator may require customization for supporting libraries since the target language may have specific requirements for a library component implementation. For example, the MontiGem generator creates import statements with relative paths for library components and absolute paths for application components.

The current realization of the library integration uses a predefined list of sub-libraries with commonly used components. The information on how to import components from a sub-library is specified in the MontiGem generator. For example, a button component is a part of `@montigem/basic` sub-library, and the generator uses the library name to import the component. An alternative solution is importing sub-libraries based on the model's component usages. The library could provide the sub-library names that the generator would use. This way, the generator becomes independent of the libraries, and the imports can potentially be further optimized by importing individual GUI components.

7 Related Work

GUI component libraries implemented with general-purpose languages [15, 51, 58, 69] have well-defined tooling, processes, and documentation for their management from the library creation to integration and maintenance. Such libraries inspired our work, and our approach is particularly close to the Angular framework and its library system [3]. The Angular framework is built on web technologies like TypeScript, HTML, and CSS for component development and packaging systems, such as npm and webpack. It provides a custom HTML extension, which can be viewed as a DSL, and has a set of tools working together to provide and simplify component library management. Our work adapts this infrastructure, creating a conceptual framework and a method supporting libraries in the MDE context.

A common, straightforward model-driven approach to GUI creation is to specify a predefined set of GUI components directly in the DSL [2, 53]. The approach is simple, generally easier to implement, and provides a syntactic variety compared to a DSL supporting a library. However, as demonstrated by Gerasimov et al. [32], defining an abstract GUI component concept in the language and specific components in models, *e.g.,* using the self-extension mechanism, facilitates the library approach, enabling accelerated expansion of GUI component libraries. Although the self-extension mechanism is used in different forms in research [50, 63], component libraries and their lifecycle are rarely discussed.

Some prominent DSLs for modeling web applications utilize the library approach. Visser [64] presents WebDSL — a language for designing and generating web applications. The DSL uses abstract concepts, such as page fragments and templates, similar to the GUI components in this paper. WebDSL provides a module and import system that enables the reuse of concepts and opens a potential for model library support. The DSL has navigation, interaction, and layout component groups. Our work advances the research by describing the model library, including the different component types and principles behind library structuring. Furthermore, we explain in detail the library deployment and integration.

Brambilla and Fraternali [9] provide an extensive list of GUI components in IFML — a graphical UML standard language for describing the front end of software applications. They describe basic components and component composition patterns for various domains, such as mobile applications and business processes. The extension and distribution of components are done via the UML profiling mechanism. Our work specifies a different structuring for a GUI component library and provides a more detailed lower-level view of the library deployment and integration.

The UML profiling mechanism is similarly utilized by Link et al. [40]. The authors specify stereotypes for different groups of GUI components, such as `InputElement`, `ChoiceElement`, and `ContainerElement`, to provide extension points for adding new components. The work introduces enumerations for the stereotypes, for example, `TextField` and `PasswordField`, practically specifying a GUI component library. The paper focuses on the interaction between GUI and UML models, leaving out the details on the library.

Aschauer et al. [5] use the self-extension mechanism to specify the API for a UI using models and having abstract concepts for API definition in the language. The authors describe libraries for nested domains, giving an example of a domain concept library used by a manufacturer library and a customer library using both. Their work aligns with ours, but does not go into further details about libraries, such as how to structure, deploy, and integrate them into applications.

8 Conclusion

This paper introduces a method to create web-system specifications without using a platform-specific frontend development language based on an extensible GUI component library. We provide a method for realizing and managing different stages of GUI component library development in a model-driven context. We evaluate the approach by comparing it to traditional web development. Although the quantitative analysis shows positive results, the classic issues of model-driven technology, such as the steep learning curve and tools' immaturity, still pose significant challenges to the approach's widespread adoption and practical usability.

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.

[GJM+25] A. Gerasimov, N. Jansen, J. Michael, B. Rumpe, S. Will:
A Model-Driven Approach to Design, Generation, and Deployment of GUI Component Libraries.
In: Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, pp. 57-70, SLE '25,
ISBN 979-8-4007-1884-7, DOI 10.1145/3732771.3742713, ACM, Jun. 2025.

References

[1] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. 2018. Model-Based Generation of Enterprise Information Systems. In *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, Vol. 2097. CEUR-WS.org, 75–79. doi:10.18154/RWTH-2019-00745

[2] Arsalan Ali, Muhammad Rashid, Farooque Azam, Yawar Rasheed, and Muhammad Waseem Anwar. 2021. A Model-Driven Framework for Android Supporting Cross-Platform GUI Development. In *2021 National Computing Colleges Conference (NCCC)*. 1–6. doi:10.1109/NCCC49330.2021.9428869

[3] Angular. 2025. *Overview of Angular libraries.* Retrieved February 9, 2025 from https://angular.dev/tools/libraries

[4] Pascal Archambault, Houari Sahraoui, and Eugene Syriani. 2024. A Modeling Methodology for Crop Representation in Digital Twins for Smart Farming. In *ACM/IEEE 27th Int. Conf.on Model Driven Engineering Languages and Systems (MODELS Comp. '24)*. ACM, 342–352. doi:10.1145/3652620.3688247

[5] Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. 2010. A modeling language's evolution driven by tight interaction between academia and industry. In *32nd ACM/IEEE Int. Conf. on Software Engineering, V. 2 (ICSE '10)*. ACM, 49–58. doi:10.1145/1810295.1810304

[6] 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). doi:10.1016/j.cola.2022.101121

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

[8] Federico Bonetti, Antonio Bucchiarone, Judith Michael, Antonio Cicchetti, Annapaola Marconi, and Bernhard Rumpe. 2024. Digital Twins of Socio-Technical Ecosystems to Drive Societal Change. In *System Analysis and Modelling Conference, MODELS Companion '24: Int. Conf. on Model Driven Engineering Languages and Systems (SAM)*. ACM, 275–286. doi:10.1145/3652620.3686248

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

[10] 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. doi:10.1007/978-3-319-26172-0_4

[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. doi:10.1016/j.scico.2023.103033

[12] 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 Prog.: Concepts and Experiences (GPCE 22)*. ACM. doi:10.1145/3564719.3568689

[13] 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. doi:10.5381/jot.2022.21.4.a5

[14] 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 SE and Software Development Series.

[15] D3.js Community. 2011. D3.js – Data-Driven Documents. https://d3js.org/

[16] World Wide Web Consortium. 2018. *CSS Flexible Box Layout Module Level 1.* https://www.w3.org/TR/css-flexbox-1/

[17] World Wide Web Consortium. 2020. *CSS Grid Layout Module Level 2.* Retrieved February 4, 2025 from https://www.w3.org/TR/css-grid-2/

[18] 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). doi:10.1016/j.cola.2022.101117

[19] Giordano D'Aloisio, Alessandro Di Matteo, Alessia Cipriani, Daniele Lozzi, Enrico Mattei, Gennaro Zanfardino, Antinisca Di Marco, and Giuseppe Placidi. 2024. Engineering a Digital Twin for Diagnosis and Treatment of Multiple Sclerosis. In *ACM/IEEE 27th Int. Conf.on Model Driven Engineering Languages and Systems (MODELS Comp. '24)*. ACM, 364–369. doi:10.1145/3652620.3688249

[20] Vinicius Kreischer de Almeida et al. 2024. A Digital Twin System for Oil And Gas Industry: A Use Case on Mooring Lines Integrity Monitoring. In *ACM/IEEE 27th Int. Conf.on Model Driven Engineering Languages and Systems (MODELS Comp. '24)*. ACM, 322–331. doi:10.1145/3652620.3688244

[21] Juan de Lara, Esther Guerra, and Paolo Bottoni. 2024. Modular language product lines: concept, tool and analysis. *Software and Systems Modeling* (2024), 1–30. doi:10.1007/s10270-024-01179-9

[22] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th Int. Conf.on Software Language Engineering (SLE)*.

[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). doi: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). doi:10.5381/jot.2022.21.4.a4

[25] Lenin Erazo-Garzón, Steveen Suquisupa, Alexandra Bermeo, and Priscila Cedillo. 2023. Model-Driven Engineering Applied to User Interfaces. A Systematic Literature Review. In *Applied Technologies*. Communications in Computer and Information Science, Vol. 1755. Springer, Cham, 575–591. doi:10.1007/978-3-031-24985-3_42

[26] Lenin Erazo-Garzón, Steveen Suquisupa, Alexandra Bermeo, and Priscila Cedillo. 2023. Model-Driven Engineering Applied to User Interfaces. A Systematic Literature Review. In *Applied Technologies*. Springer Nature Switzerland, 575–591.

[27] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Twelfth Workshop on Language Descriptions, Tools, and Applications*. 1–8. doi:10.1145/2427048

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

[29] Andrew Forward and Timothy C. Lethbridge. 2008. Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals. In *Int. WS on Models in SE (MiSE '08)*. ACM, 27–32. doi:10.1145/1370731.1370738

[30] Francois Fouquet, Thomas Hartmann, Cyril Cecchinel, and Benoit Combemale. 2024. GreyCat: A Framework to Develop Digital Twins at Large Scale. In *ACM/IEEE 27th Int. Conf.on Model Driven Engineering Languages and Systems (MODELS Comp. '24)*. ACM, 492–495. doi:10.1145/3652620.3688265

[31] 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 *Modellierung 2020 Short, Workshop and Tools & Demo Papers*. CEUR-WS.org, 22–30.

[32] Arkadii Gerasimov, Nico Jansen, Judith Michael, and Bernhard Rumpe. 2024. Applying a Self-Extension Mechanism to DSLs for Establishing Model Libraries. In *23rd ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 29–43. doi:10.1145/3689484.3690732

[33] 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. doi:10.1007/978-3-030-72696-6_5

[34] Jeff Gray, Sandeep Neema, Juha-Pekka Tolvanen, Aniruddha S Gokhale, Steven Kelly, and Jonathan Sprinkle. 2007. Domain-Specific Modeling. In *Handbook of Dynamic System Modeling*.

[35] Malte Heithoff, Nico Jansen, Judith Michael, Florian Rademacher, and Bernhard Rumpe. 2024. *Model-Based Engineering of Multi-Purpose Digital Twins in Manufacturing*. Springer Nature Switzerland, Cham, 89–126. doi:10.1007/978-3-031-67778-6_5

[36] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag. doi:10.2370/9783844080100

[37] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *33rd Int. Conf.on Software Engineering*. ACM. doi:10.1145/1985793.1985858

[38] Abdallah Karakra, Franck Fontanili, Elyes Lamine, Jacques Lamothe, and Adel Taweel. 2018. Pervasive computing integrated discrete event simulation for a hospital digital twin. In *IEEE/ACS 15th Int. Conf.on Computer Systems and Applications (AICCSA)*. IEEE.

[39] Lennart CL Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*. ACM, 444–463. doi:10.1145/1932682.1869497

[40] Stefan Link, Thomas Schuster, Philip Hoyer, and Sebastian Abeck. 2008. Focusing Graphical User Interfaces in Model-Driven Software Development. In *First Int. Conf. on Advances in Computer-Human Interaction*. 3–8. doi:10.1109/ACHI.2008.16

[41] 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. doi:10.1016/j.cl.2016.09.004

[42] 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. doi:10.1145/1118890.1118892

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

[44] Judith Michael, Imke Nachmann, Lukas Netz, Bernhard Rumpe, and Sebastian Stüber. 2022. Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines. In *Modellierung 2022*. GI, 33–48. doi:10.18420/modellierung2022-012

[45] Neda Mohammadi and John E. Taylor. 2017. Smart city digital twins. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*.

[46] Pedro J. Molina. 2019. Quid: prototyping web components on the web. In *ACM SIGCHI Symp. on Engineering Interactive Computing Systems (EICS '19)*. ACM, Article 3. doi:10.1145/3319499.3330294

[47] Soheil Sabri, Kostas Alexandridis, and Newton Lee. 2024. *Digital Twin*. Springer Nature Switzerland. doi:10.1007/978-3-031-67778-6

[48] Bran Selic. 2003. The pragmatics of model-driven development. *IEEE Software* 20, 5 (2003), 19–25. doi:10.1109/MS.2003.1231146

[49] Bran Selic. 2017. Model-Based Software Engineering in Industry: Revolution, Evolution, or Smoke? https://www.youtube.com/watch?v=miPZyfRIcs8 Talk at the Monash University Dean's Seminar Series.

[50] Thiago Rocha Silva, Jean-Luc Hak, and Marco Winckler. 2017. A Behavior-Based Ontology for Supporting Automated Assessment of Interactive Systems. In *2017 IEEE 11th Int. Conf. on Semantic Computing (ICSC)*. 250–257. doi:10.1109/ICSC.2017.73

[51] Ivan Sopin and Felix G. Hamza-Lup. 2010. Extending the Web3D: design of conventional GUI libraries in X3D. In *15th Int. Conf. on Web 3D Technology (Web3D '10)*. ACM. doi:10.1145/1836049.1836070

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

[53] Harald Störrle. 2010. Model driven development of user interface prototypes: an integrated approach. In *Fourth European Conference on Software Architecture: Comp. Volume (ECSA '10)*. ACM, 261–268. doi:10.1145/1842752.1842802

[54] Ana Maria Şutîi, Mark den van Brand, and Tom Verhoeff. 2018. Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod. *Computer Languages, Systems & Structures* 51 (2018), 48–70. doi:10.1016/j.cl.2017.07.004

[55] 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*. Springer, 45–70. doi:10.1007/978-3-030-81915-6

[56] Angella Thomas, David A Guerra-Zubiaga, and John Cohran. 2018. Digital Factory: Simulation Enhancing Production and Engineering Process. In *ASME Int. Mechanical Engineering Congress and Exposition*, Vol. 52019. American Society of Mechanical Engineers, V002T02A077.

[57] Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *24th ACM SIGPLAN Conf. Comp. on Object Oriented Programming Systems Languages and Applications*. ACM, 819–820. doi:10.1145/1639950.1640031

[58] Vasileios Triglianos and Cesare Pautasso. 2015. Asqium: A JavaScript Plugin Framework for Extensible Client and Server-Side Components. In *Engineering the Web in the Big Data Era*. Springer, 81–98.

[59] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40. doi:10.1016/j.cl.2015.02.001

[60] Tijs van der Storm. 2011. *The Rascal Language Workbench*. CWI. Software Engineering [SEN].

[61] Arie van Deursen and Paul Klint. 1998. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice* 10, 2 (1998), 75–92. doi:10.1145/1869542.1869623

[62] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.* 35, 6 (June 2000), 26–36. doi:10.1145/352029.352035

[63] Jean Vanderdonckt. 2008. Model-Driven Engineering of User Interfaces: Promises, Successes, Failures, and Challenges. *Annual Romanian Conf. on Human-Computer Interaction* (01 2008).

[64] Eelco Visser. 2008. *WebDSL: A Case Study in Domain-Specific Language Engineering*. Springer, 291–373. doi:10.1007/978-3-540-88643-3_7

[65] Ryno Visser, Anton Basson, and Karel Kruger. 2024. An Architecture for the Integration of Product and Production Digital Twins in the Automotive Industry. In *ACM/IEEE 27th Int. Conf.on Model Driven Engineering Languages and Systems (MODELS Comp. '24)*. ACM, 431–441. doi:10.1145/3652620.3688257

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

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

[68] Andrzej Wąsowski and Thorsten Berger. 2023. *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*. Springer. doi:10.1007/978-3-031-23669-3

[69] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org

[70] Jon Yablonski. 2024. *Laws of UX: 10 praktische Grundprinzipien für intuitives, menschenzentriertes UX-Design*. O'Reilly and dpunkt-Verlag.