[GMNR21] A. Gerasimov, J. Michael, L. Netz, B. Rumpe:
Agile Generator-Based GUI Modeling for Information Systems.
In: Modelling to Program (M2P), pp. 113-126, Springer, 2021.
www.se-rwth.de/publications/

Agile Generator-Based GUI Modeling for Information Systems

Arkadii Gerasimov, Judith Michael(✉), Lukas Netz, and Bernhard Rumpe

Software Engineering, RWTH Aachen, Aachen, Germany
{gerasimov,michael,netz,rumpe}@se-rwth.de
http://www.se-rwth.de

Abstract. We use two code generators for the model-based continuous development of information systems including its graphical user interfaces (GUIs). As our goal is to develop full-size real-world systems for different domains, the continuous and iterative model-based engineering of their GUIs comes along with challenges regarding their extension and modification. These challenges concern models, the languages they are written in and hand-written code. In this work we present four complementary approaches to allow extensions for GUIs that we encounter with the generator-based framework MontiGem to tackle these challenges. We discuss the four approaches in detail and present extensions of the framework in the grammar of the language, via atomic components, via hand-written amendments of generated models and by generating connections between the GUI and data structure models. These techniques can be used to create a flexible DSL for engineering information systems, adaptable for different domains and rapidly changing requirements.

Keywords: Information system · Modeling graphical user interfaces · Model-Based Software Engineering · Code generation · MontiGem

1 Introduction

Model-Based Software Engineering (MBSE) and code generators are well established technologies [30,31]. MBSE uses models as the basis for software engineering. They describe both problem and solution consistently and produce a solution that give comprehensive and verifiable answers to the system requirements posed by the problem [24]. These advantages provide significant support when seeking a solution to a software engineering problem.

MBSE and code generators can be used to create an Enterprise Information System (EIS) [2] using a multitude of Domain-Specific Languages (DSLs) [1]. Within this paper, we focus on the modeling of Graphical User Interfaces (GUIs) with a DSL and take a closer look on the needs for an agile iterative engineering process. In order to refine the target code, mechanisms to extend the generated code with hand-written amendments are introduced. This leaves us with two options to modify the application: Changing the model in order to generate new target code, or changing the hand-written code directly, for example

A. Dahanayake et al. (Eds.): M2P 2020, CCIS 1401, pp. 1–14, 2021.
https://doi.org/10.1007/978-3-030-72696-6_5

by extending the generated classes. In order to take advantage of the perks of MBSE and code generation, we intend to define as much as possible within the models. Therefore we have to consider a third option: Extending the grammar of a modeling language in order to facilitate the definition of further aspects.

Within this paper, we tackle three **challenges** regarding (1) the modification and extension of an EIS GUI (2) using an agile engineering approach (3) allowing for continuous re-generation (4) without adapting the hand-written code:
Challenge 1: Our GUI modeling language [1] is good enough for reuse in different generated EISs but *too generic and restrictive regarding addition of new components*. By extending the modeling capabilities of the language, we can integrate new components easily, generate more specific source code and reduce the amount of hand-written code in the resulting EISs.
Challenge 2: The current approach lacks support for *iterative, evolving GUI models* which can be reused. Similar to a pattern approach, a model should be usable as a template within another model of the same language.
Challenge 3: The current approach lacks support for *interweaving models from different languages*. Aspects already defined in a different model, *e.g.,* a data model, should be reusable to prevent redundancy in modeling. This requires composition of modeling languages.

To overcome these challenges, this paper addresses the **research question** *how we can extend and modify the GUI of an application created with MBSE methods without having to adapt the hand-written code for each particular generated EIS.* In this work we discuss *four approaches* to modify and extend the generated application and inspect options on how to adapt given models without having to adapt the hand-written code. The four (non-exclusive) options are:

(A1) Extension in the grammar: Introducing an approach to extend the grammar of the GUI-DSL enables the software engineer to define more problem-specific models and, thus, reduce the amount of hand-written source code (Challenge 1).
(A2) Extension via atomic components: A general GUI component in the grammar can be used to create and assemble new components and adds a new level of flexibility to the language (Challenge 1). This extension is similar to the first approach, but is more advantageous in particular cases.
(A3) Extension and modification via addition of hand-written GUI models: To modify GUI models by adding hand-written ones allows for model modifications despite continuous re-generation processes (Challenge 2).
(A4) Extension via data models and connections to the GUI models: Defining a connection between the two DSLs used to define the data structure and the GUI allows to generate view models from models of both types (Challenge 3).

Outline. The next section explains several prerequisites for agile and iterative GUI modeling of EISs. Section 3 presents our four approaches for GUI modeling. Section 4 discusses other GUI languages and how they handle the challenges described before. Section 5 concludes.

2 Prerequisites

MBSE relies on the use of models to reduce the complexity of developed applications and systems [17]. Engineers can use a variety of modeling languages for agile and model-based development of an EIS. In our case, these languages and their tooling are created with the modeling platform and language workbench MontiCore [18]. We use the generator framework MontiGem [2] and a newly developed DSL, called GUI-DSL, to create the resulting EIS [14].

2.1 The Language Workbench MontiCore

The MontiCore [18,22] language workbench facilitates the engineering of textual DSLs. It provides mechanisms to analyze, manipulate and transform the models of a developed DSL. The concrete syntax of a DSL is defined in extended context-free grammars and context conditions are programmed in Java for checking the well-formedness of models. MontiCore generates parsers, which are able to handle models of the DSL, and infrastructures for transforming the models into their Abstract Syntax Tree (AST) representation and for symbol table construction. The AST and symbol table infrastructures embody the abstract syntax of a modeling language. Once the parser gets a model as an input, it creates an AST for each model as well as a symbol table for further processing, *e.g.,* for code generation or analysis. The input AST can be transformed as needed. A template engine uses the output AST together with templates for the target language to create the resulting code [18]. MontiCore provides means for a *modular development* and *composition* of DSLs [18]. It supports language inheritance, embedding and aggregation [15].

Until now, a variety of languages, including a collection of UML/P [28] languages (variants of UML which fit better for programming) as well as the OCL, delta, tagging languages, SysML and architecture description languages are realized with MontiCore[1].

2.2 Creating Information Systems with MontiGem

MontiGem [1,2], the generator framework for enterprise management, creates an EIS out of a collection of input models and allows for hand-written additions. The parser, AST-infrastructure and the symbol table infrastructure of the framework are generated by MontiCore. Figure 1 shows the main generation process for an EIS using MontiGem.

MontiGem uses Class Diagrams (CDs) (domain and view models) and GUI-DSL models (views) as input (see A in Fig. 1). Additionally, it can handle OCL models, *e.g.,* for creating validation functions for input data, and tagging models, *e.g.,* for adding platform specific information to the domain model.

For code generation MontiGem uses two generators (B in Fig. 1): The data structure and the user interface generator. The data structure generator creates

[1] see http://www.monticore.de/languages/.

the database schema, back-end data structure and communication infrastructure for the back-end and front-end out of CD models and view models. The user interface generator creates TypeScript and HTML files for the front-end out of GUI models. The generator parsers check the input models for syntactical correctness and produce ASTs. The generators feed the ASTs and templates for the target languages (Java for the back-end and TypeScipt/HTML for the front-end) to a template engine, which generates the resulting code.

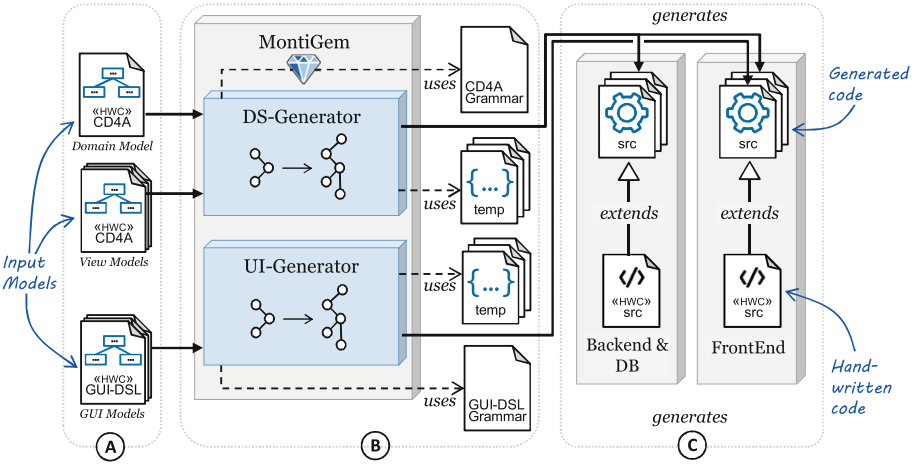


Fig. 1. The generation process using MontiGem

MontiGem can handle hand-written additions of the generated code. This is important for agile engineering processes and their need for continuous re-generation. For example, CSS classes are defined manually in separate files and can be referred to in the models. In other cases, the generated code needs to be extended by the hand-written code (HWC) directly, in which case the *TOP-mechanism* [18] is used. The main idea of the TOP-mechanism is to extend the generated code with the hand-written code using *inheritance*, while keeping artifacts separate (C in Fig. 1). This allows to extend the business logic of the information system and continuously re-generate code without losing the hand-written extension.

We have used the generator framework MontiGem for engineering of several applications, *e.g.,* a full-size real-world application for financial and staff control [13] and projects in the manufacturing [10], automotive and energy domain.

2.3 Roles for Language and Application Engineering

In practice, language and application engineering are separated processes, which involve people from different teams. Figure 2 shows roles related to the four approaches in Sec. 3. We distinguish six different roles that are involved in the

application and generator engineering process: In order to define a model we need a DSL. A **Language Engineer** defines the grammar (Fig. 1Ⓑ) of the DSL and maintains it. These DSLs are used by the **Application Modeler** to define models (Fig. 1Ⓐ) that represent different aspects of the modeled application. Dependent on the DSLs, the Application Modeler does not require a background in programming, but should have good knowledge of the domain. Once a model is defined, it is parsed, interpreted and transformed into target source code (Fig. 1Ⓒ) by the generator. The generator itself (Fig. 1Ⓑ) is maintained and configured by the **Generator Customizer**. In small development teams, Language Engineer and Generator Customizer might be the same person, whereas in larger teams this might even be people from different teams. Hand-written parts (Fig. 1Ⓒ) of the resulting application are implemented by the **Application Programmer** who has coding skills. Predefined, not generated and not application specific components (Fig. 1Ⓒ) are implemented by the **Component Provider**. Finally, the **Tool Provider** configures and maintains libraries and components (Fig. 1Ⓒ) in the run-time environment of the application and the generator (Fig. 1Ⓑ).

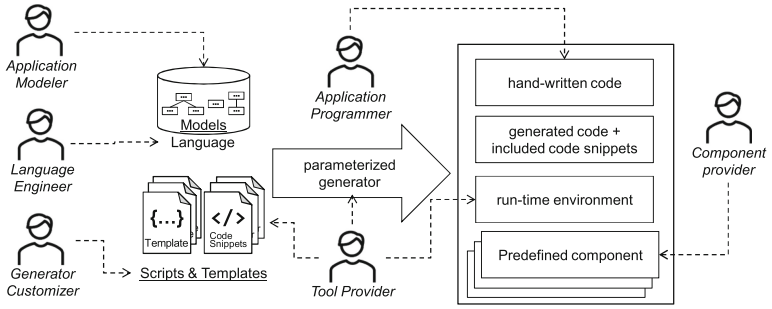


Fig. 2. Roles for language and application engineering

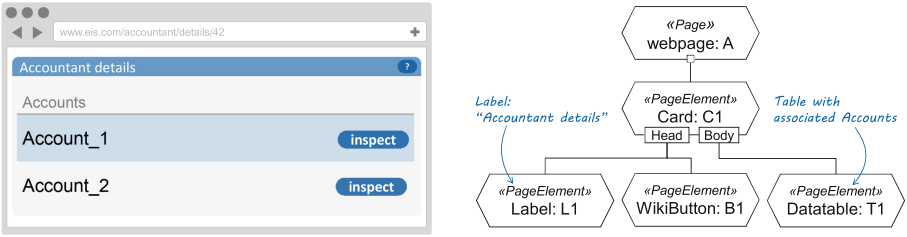
2.4 GUI-DSL, a Language for Defining User Interfaces

The GUI-DSL [14] is a textual modeling language aimed at describing an interactive user interface for data representation and management in an EIS. The DSL is part of a large collection of MontiCore languages and allows to create complex views by defining models, each of which is interpreted by the user interface generator and transformed into a web-page.

Grammar. As the web interface is a primary generation target, principles of web interface design are followed in the language and reflected in the structure of the grammar. The core concept of the grammar is built around the idea of constructing a view by combining and nesting GUI-components, which provide various data representations, *e.g.,* text, charts, tables, or are used to define the layout of the page. In the context of this work, a GUI-component is a software component with a graphical representation, defining a visual interaction unit

and/or providing graphical representation of information. Another aspect handled by the language is the definition of data sources for the view, which includes minimal information about how and in what format the data is delivered.

Model. Each model of the language describes a web page and has a structure similar to an HTML document. The page definition describes a hierarchical view: A tree, where leaves provide the graphical representation of the data and intermediate nodes define the layout of the page, *e.g.,* Fig. 3b depicts the structure of the web page on Fig. 3a. The GUI-components are part of a `Card` container, where the head (blue top area of Fig. 3a) has a `Label` (text "Accountant details") and a `Button` (with a question mark). The body contains a `Datatable` (table of accounts). A distinctive feature of a GUI-DSL model is the data handling. The page definition includes information about the data as variables, which are used as an input for GUI-components.



(a) Example GUI

(b) Tree structure of GUI

Fig. 3. Example of GUI Decomposition

Generator. The models are processed by the generator, which further produces the application code. The process of transformation is executed in several steps. First, the model is parsed and an AST of the model is created. The AST is further transformed into a form which closely resembles the structure of the generated code and is finally given to the template engine to construct the view and the logic of the page piece by piece. Optionally, the generated code is further formatted. This can be useful during the development as it makes the code more readable and helps to analyze the code.

In order to define a web-page with simple functionality, it is enough to create a GUI-DSL model to define the graphical representation and a data model to define the data being displayed on the page. We are also investigating approaches where only a CD is needed to create the first version of an application [14].

3 Approaches

During the agile development of an information system, several challenges arise for GUI modeling. We describe the challenge to be tackled in detail, propose a solution, define the roles involved in handling the challenge and analyze positive

and negative aspects of the solution. The overreaching objective is to support the application programmer with MBSE by increasing the amount of generated code and decreasing the amount of hand-written code.

3.1 GUI-Extension via Grammar (A1)

A GUI model consists of GUI-components describing parts of a user interface. If a new GUI-component is implemented and used, it has to be reflected in the GUI model. Additionally, it has to be determined how the GUI-DSL needs to be changed and how it affects models of the language.

Challenge. Depending on the specification of data presentation in the user interface, various GUI-components are needed and new ones have to be implemented and integrated into the application (Challenge 1). In GUI-DSL, a page is described by a model which uses GUI-components. The GUI-components have to be included in the definition of the DSL to be useable in the model. The DSL is unaware of new GUI-components and they cannot be used in the model until its interfaces are described as a part of the DSL. It is also necessary to implement the transformation process in the generator in order to properly map the usage of a GUI-component in the model to its usage in the target code.

Roles. The adjustment of the current language is handled by a Language Engineer, whereas the generator is adjusted by a Generator Customizer.

Solution. A direct solution is to make both DSL and the corresponding generator extendable. This can be achieved with various kinds of language compositions, such as aggregation or inheritance [19]. The idea is to separately define a new DSL with its own generator, which can be combined with the existing one to allow usage of both DSLs in one model and produce a combined result. For example, a calendar GUI-component needs to be added to the user interface. We have to create a separate extension of the DSL and corresponding generator, which describes the calendar GUI- component and generate the code for it. Such approach requires a common ground for different kinds of GUI-components, an interface to consistently handle them and a reusable generator structure to mitigate the effort spent on implementation of mapping for the new component.

Advantages and Disadvantages. The solution is straightforward for both development roles involved and allows to embed the logic necessary for a proper operation of a GUI-component on a page directly into the generator. This is especially useful if the added component has a complicated structure and is heavily configurable. Specific syntax can be defined to simplify its definition in the model and let the generator handle the specifics. On the other hand, it takes considerable effort to integrate even the simplest GUI-components, as a new language and different generator parts have to be created and integrated with the existing ones.

3.2 GUI-Extension via Atomic Components (A2)

The previous challenge can be addressed from different angles and although the previous solution is valid, it has a considerable disadvantage regarding the effort.

Challenge. An ability to use a hand-written GUI-component in a model is important, but it is also important to provide this ability without spending a lot of effort on creating a DSL and a generator, especially if the added component is very simple and does not require additional specific behavior definition to be operable on a page (see Challenge 1). In such case, it is desirable to enable GUI-component definition in models, where a new component is defined in a model and conforms to a common structure. We will call these GUI-components atomic, which signify a smallest unit of view composition on a modeling level.

Roles. Since the problem is solved in a different phase, the responsibility to define new components moves from the Generator Customizer and Language Engineer to the Application Modeler.

Solution. Defining a GUI-component in a model means that a rule for such general definition of atomic components needs to be present in the language. This rule specifies how to create a new GUI-component, *e.g.,* by using the name of the component and its interface. The interface consists of configurable properties of the component, which then have to be defined in a model to allow instantiation and usage of the GUI-component in other models. Taking the example from the previous section, a calendar needs to be added to the user interface. This solution suggests to describe the calendar GUI-component in a model, which can be referred to from other models.

Advantages and Disadvantages. Introducing a new GUI-component in a model saves a lot of effort compared to DSL extension. If a simple component is implemented, this approach allows to quickly integrate it into the application models, and generate the corresponding code without penalty. Additionally, a uniform rule for component creation ensures consistency of GUI-component representation in models. However, if a complex component is created which requires additional logic to be used on web pages, either a mechanism for indirect generator manipulation needs to be defined and attached to the language or the generated code always has to be completed by a hand-written amendment. This solution amplifies the development effort proportional to the complexity of an introduced GUI-component: The more complex the component is, the more additional effort is required for its integration, whereas the previous solution implies the similar amount of effort regardless of the component complexity.

3.3 GUI-Extension and Modification via Hand-Written Model (A3)

Models can be generated or hand-written. Independently from the origin of models, modifications can be necessary and should remain during re-generation.

Challenge. In [14] we have introduced an approach to generate GUI-models from CDs. In such case, developers would like to change the appearance of the application which results either in adding larger amounts of hand-written code, changing the generated GUIs or changing the generated models. Whereas the first approach requires manual effort, any changes would be undone by the next generation cycle for the latter two. If not only code but also models are generated,

changes in these models will get lost during re-generation processes. This requires an approach to handle changes for generated models.

Roles. The Application Modeler modifies models.

Solution. We can extend the GUI-DSL to allow for modification of a GUI model by adding another GUI model. This extension enables references between GUI models. The application modeler for example can be provided with a generated model, that displays data as a table. This approach enables him to replace the table with a bar chart by modifying the generated model with his handwritten one. As described in Fig. 3, a GUI model can be represented in a tree structure. With a simple set of tree manipulations, any GUI-model can be transformed into the one needed. Similar to the TOP-Mechanism [18], we use naming conventions to distinguish between and identify the generated and the hand-written model. The hand-written model itself is a valid model and replaces the generated one. This approach is discussed in [14] in more detail.

Advantages and Disadvantages. Extending models by further models, can lead to a lack of clarity considering where components are defined and configured. Hand-written models themselves can be modified multiple times leading to badly defined models. The alternative would be a modification within the generated source code, but depending on the size of the modification needed, an adaption of the model instead of the code might be easier to maintain and to comprehend for the developer while simultaneously generating consistent code.

3.4 GUI-Extension via Data-Models (A4)

Agile development requires additions and changes in the data structure. These changes will have effects on the graphical representation.

Challenge. The data structure of iteratively evolving software is very likely to be subject to change. The user interface presents and allows for interaction with the data, which is defined in the data structure. There is a strong dependency between GUI and data structure which should be reflected in a dependency between the GUI model and the domain model (Challenge 1).

Roles. The view is modeled by the *Application Modeler*, whereas the grammar of the GUI-DSL has to be adapted by the *Generator Customizer*.

Solution. In order to display data from the database in the GUI we use the MVVM pattern (Fig. 4). The `Data-Model` is defined by the domain model (a class diagram) and is used as an input for the generator. The `View` is defined by the GUI-model. We can use a combination of both to derive the `View-Model`. By referencing classes and attributes of the `Data-Model` in the `View`, we can derive the classes and attributes needed in the `View-Model`. An additional class diagram (`View-Model`) is generated for each GUI-model and provided to the data structure generator, which uses the `Data-Model` and all `View-Models` as an input. The dependency from `Data-Model` to `View-Model` can be used to load and transport the correct data to the `View`. For example, when defining a table in a GUI model, the data model could be referenced directly by the table entries.

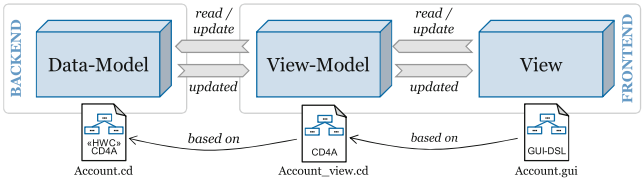


Fig. 4. MVVM pattern

Advantages and Disadvantages. Generated view-models can be less efficient than hand-written ones. This problem can be mitigated by allowing custom view-models, that are used instead of the generated ones. The advantage of generated view-models is that they always fit to the domain model. Changing the data structure will result in automatically changed view-models. Hand-written view-models become obsolete, which reduces developer effort.

4 Discussion and Related Work

MBSE for user interfaces has been addressed in different publications [5,20,23,25]. Different solutions address directly or indirectly the approaches discussed in this work, namely extensions (A1) in the grammar, (A2) via atomic components, (A3) via hand-written models and by generating models from models and (A4) by generating connections between the GUI and data-structure models. MontiGem in combination with MontiCore explicitly supports all four approaches.

GUI-Extension via Grammar (A1)**.** The problem of extending the language to allow more flexible definition of the user interface is often set to the background [3,4,16]. The set of elements in the view is considered to be static and focused purely on information delivery. This can result in a need for language extension when the models are required to be applied for another domain. For example, the modeling language IFML [5] had to be extended in order to fit a different domain [7,8,26]. Similar to GUI-DSL, the language introduces interfaces and means for abstractions, which serve as a base for new GUI-components and provide a general description for components of the same type, although the adaptability of the language to a different domain is not considered directly. Aside from adding new GUI- components into the language, the extension is also possible using stereotypes, *i.e.,* additional information is attached to existing components with keywords, which specify the interpretation of a GUI-component. Such solution also requires implementation of a separate language and a generator and can be used if an infrastructure for this approach is available. Diep et al. [11] suggest several GUI modeling levels, starting from an abstract representation of GUI-components in general, followed by more specific element type, *e.g.,* button, text box, *etc.*, and ending with platform-specific elements, such as `StackPanel` for Windows Phone. The abstract models are transformed into more specific ones up to the generation of the target code. Having several levels of abstraction allows to add not only specific elements, but also a new

category of GUI-components. However, adding a new category of components would still require the definition of the mapping to the more specific elements or target code.

GUI-Extension via Atomic Components (A2)**.** Most approaches for user interface modeling use predefined sets of GUI-components [3,4,16]. There exist, however, some approaches which provide the ability to extend the variety of GUI-components. One option is to combine components from a predefined set into a new component and use it in models of the user interface [9,21,23]. Although it does not affect the visual appearance of an interface, it helps to reuse presentational patterns. An interesting take on the problem is discussed in [20], which defines a DSL based on a Groovy programming language. The DSL uses function calls similar to the rules for construction of GUI-components described in this work to build the view, *i.e.,* the GUI-component definitions are functions and new components can potentially be added by defining new functions. The key feature is that the pages and GUI-components are both described on the same modeling level. The GUI-components are functions, which are called in the body of a page function, thus constructing the whole view. This technique is similar to the presented one, but it is not designed to simplify the process of defining a GUI-component on a modeling level and does not guarantee the simplicity of GUI-component integration.

GUI-Extension and Modification via Hand-Written Model (A3)**.** The concept of user interface derivation finds its place in [6], where the first step is to model the tasks with details such as task context, roles involved and manipulated objects. The user interface model is derived from the task model based on the dependencies between the tasks. As a result, the interface is updated when a task model is modified. [29] proposes model-to- model transformation in order to get models of user interface from models describing use cases, data structure and other models, which correspond to various aspects of an application. The approach considers only two options for modeling a user interface: it is either completely derived or completely modeled by hand. Our work goes one step further and enables flexible modification of the generated models.

GUI-Extension via Data-Models (A4)**.** The possibility to reference the data structure in a user interface model has been recognized in [12,27,32]. The overall approach regarding the connection between view and a data structure is similar to the one in this work: The publications propose the usage of several models to tackle different aspects of a web application, where a domain model and presentation model are defined separately, but at the same time the presentation model refers to the domain model in order to define the content of a user interface. Using such concept allows to describe parts of application focused on specific problem separately and to bring these parts together by integrating fragments of a domain model in a presentation model to define their content.

In general, a DSL designed for GUI description tends to have a predefined set of GUI-components used to build a user interface. This allows a modeler to easily create a new GUI by simply choosing a suitable data presentation from

a small set. However, this also implies that the language *restricts the variability* of the generated GUI, which is necessary if the language needs to be further reused and applied to a different problem domain. Making the language flexible by introducing extension mechanisms to integrate new GUI-components helps to cope with such challenge, but it still requires considerable effort to define a mapping between newly introduced component and the target code. If the language needs to be *heavily reusable and adaptable*, *e.g.,* to describe GUIs on different platforms or to target different users, it makes sense to introduce a creation rule for new GUI-components in the language to allow a modeler to create new building blocks for a user interface. Enhancements of a different kind, such as derivation of the GUI models, introduces the similar problem of restricting the variability of user interfaces by predefining the mapping from a non-GUI model to a GUI model. Major adjustments of a GUI on a modeling level would be confusing and difficult to maintain, while minor changes can be done manually and the problem can potentially be ignored. However, the benefits of a modeling approach would still be lost and it could be worth to create a simple set of operations to modify the generated GUI models. The last extension considered in this work is handling the connection between GUI and data. In case the data structure and GUI are modeled separately, establishing the connection between the models allows to generate a code for communication between a user interface and data provider, which enables building functional application prototypes.

The proposed solutions are designed to reduce the development time and to support the agile and iterative engineering of EIS. Typical engineering processes which consider MBSE and code generation include strong user involvement, agile reaction to changes and small product increments. The four approaches suggest adaptations on a language or a modeling level to delegate the task of changing huge parts hand-written code to a more abstract level, where the changes are minimal, thus reducing development time and effort.

5 Conclusion

MBSE and code generators allow for an agile development process of EIS. However, this comes along with the question where to make changes for changing requirements: In the models, in the grammar of the DSL the models are created with or in the hand-written code. The requirements can also imply different forms of changes, such as adding new code or adjusting the existing code. In this paper we have shown four approaches, which allow to extend and modify the GUI of an information system created with MBSE methods without having to adapt the hand-written code: (A1) in the grammar, (A2) via atomic components, (A3) via hand-written models and by generating models from models and (A4) generating connections between the GUI and data structure models. These techniques can be used to create a flexible DSL, adaptable for different domains and rapidly changing requirements with reduced manual effort.

It is important to enable extensions of GUIs using MBSE and code generation. The extendability is relevant not only for GUIs but on different levels

and also for other model-based aspects of the application. Thus, further considerations have to be done to enable extendability of other generators for the application as well as the run-time environment.

References

1. Adam, K., Michael, J., Netz, L., Rumpe, B., Varga, S.: 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 2019). LNI, vol. P-304, pp. 59–66. Gesellschaft für Informatik e.V (2020)
2. Adam, K., et al.: Model-based generation of enterprise information systems. In: Enterprise Modeling and Information Systems Architectures (EMISA 2018), vol. 2097, pp. 75–79. CEUR-WS.org (2018)
3. Bernardi, M., Cimitile, M., Maggi, F.: Automated development of constraint-driven web applications, pp. 1196–1203 (2016). https://doi.org/10.1145/2851613.2851665
4. Bernardi, M.L., Cimitile, M., Di Lucca, G.A., Maggi, F.M.: M3D: a tool for the model driven development of web applications. In: Proceedings of the Twelfth International Workshop on Web Information and Data Management, WIDM 2012, pp. 73–80. ACM (2012)
5. Bernaschina, C., Comai, S., Fraternali, P.: IFMLEdit.org: model driven rapid prototyping of mobile apps, pp. 207–208 (2017)
6. Berti, S., Correani, F., Mori, G., Paternó, F., Santoro, C.: TERESA: a transformation-based environment for designing and developing multi-device interfaces, pp. 793–794 (2004). https://doi.org/10.1145/985921.985939
7. Brambilla, M., Mauri, A., Franzago, M., Muccini, H.: A model-based method for seamless web and mobile experience, pp. 33–40 (2016). https://doi.org/10.1145/3001854.3001857
8. Brambilla, M., Mauri, A., Umuhoza, E.: Extending the interaction flow modeling language (IFML) for model driven development of mobile applications front end. In: Awan, I., Younas, M., Franch, X., Quer, C. (eds.) MobiWIS 2014. LNCS, vol. 8640, pp. 176–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10359-4_15
9. Costa Paiva, S., Oliveira, J., Loja, L., Graciano Neto, V.: A metamodel for automatic generation of enterprise information systems (2010)
10. Dalibor, M., Michael, J., Rumpe, B., Varga, S., Wortmann, A.: Towards a model-driven architecture for interactive digital twin cockpits. In: Dobbie, G., Frank, U., Kappel, G., Liddle, S.W., Mayr, H.C. (eds.) ER 2020. LNCS, vol. 12400, pp. 377–387. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62522-1_28
11. Diep, C.K., Tran, N., Tran, M.T.: Online model-driven IDE to design GUIs for cross-platform mobile applications, pp. 294–300 (2013). https://doi.org/10.1145/2542050.2542083
12. Dukaczewski, M., Reiss, D., Stein, M., Rumpe, B., Aachen, R.: MontiWeb - model based development of web information systems (2014)
13. Gerasimov, A., et al.: Generated enterprise information systems: MDSE for maintainable co-development of frontend and backend. In: Michael, J., Bork, D., (eds.) Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers, pp. 22–30. CEUR-WS.org (2020)

14. Gerasimov, A., Michael, J., Netz, L., Rumpe, B., Varga, S.: Continuous transition from model-driven prototype to full-size real-world enterprise information systems. In: Anderson, B., Thatcher, J., Meservy, R., (eds.) 25th Americas Conference on Information Systems (AMCIS 2020). AIS Electronic Library (AISeL), Association for Information Systems (AIS) (2020)
15. Haber, A., et al.: Composition of heterogeneous modeling languages. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) MODELSWARD 2015. CCIS, vol. 580, pp. 45–66. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27869-8_3
16. Heitkötter, H., Majchrzak, T.A., Kuchen, H.: Cross-platform model-driven development of mobile applications with md^2. In: SAC (2013)
17. Hölldobler, K., Michael, J., Ringert, J.O., Rumpe, B., Wortmann, A.: Innovations in model-based software and systems engineering. J. Obj. Technol. **18**(1), 1–60 (2019). https://doi.org/10.5381/jot.2019.18.1.r1
18. Hölldobler, K., Rumpe, B.: MontiCore 5 language workbench edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32, Shaker Verlag, December 2017
19. Hölldobler, K., Rumpe, B., Wortmann, A.: Software language engineering in the large: towards composing and deriving languages. Comput. Lang. Syst. Struct. **54**, 386–405 (2018)
20. Jia, X., Jones, C.: AXIOM: a model-driven approach to cross-platform application development. In: ICSOFT 2012 - Proceedings of the 7th International Conference on Software Paradigm Trends, pp. 24–33 (2012)
21. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-based web engineering: an approach based on standards, pp. 157–191 (2008)
22. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(5), 353–372 (2010)
23. Kraus, A., Knapp, A., Koch, N.: Model-driven generation of web applications in UWE. In: CEUR Workshop Proceedings, vol. 261 (2007)
24. Long, D., Scott, Z.: A primer for model-based systems engineering. Lulu. com (2011)
25. Marland, V., Kim, H.: Model-driven development of mobile applications allowing role-driven variants, pp. 14–26 (2019)
26. Morgan, R., Grossmann, G., Schrefl, M., Stumptner, M., Payne, T.: VizDSL: a visual DSL for interactive information visualization, pp. 440–455 (2018)
27. Ren, L., Tian, F., (Luke) Zhang, X., Zhang, L.: DaisyViz: a model-based user interface toolkit for interactive information visualization systems. J. Vis. Lang. Comput. **21**(4), 209–229 (2010). part Special Issue on Graph Visualization
28. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33933-7
29. Seixas, J., Ribeiro, A., Silva, A.: A model-driven approach for developing responsive web Apps, pp. 257–264 (2019). https://doi.org/10.5220/0007678302570264
30. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
31. Stahl, T., Völter, M., Efftinge, S., Haase, A.: Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management, vol. 2, pp. 64–71 (2007)
32. Valverde, F., Valderas, P., Fons, J., Pastor, O.: A MDA-based environment for web applications development: from conceptual models to code (2019)