Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems

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

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

Lukas Netz
Chair of Software Engineering, netz@se-rwth.de

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

Simon Varga
RWTH Aachen University, varga@se-rwth.de

Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems

Completed Research

Arkadii Gerasimov
RWTH Aachen University
gerasimov@se-rwth.de

Judith Michael
RWTH Aachen University
michael@se-rwth.de

Lukas Netz
RWTH Aachen University
netz@se-rwth.de

Bernhard Rumpe
RWTH Aachen University
rumpe@se-rwth.de

Simon Varga
RWTH Aachen University
varga@se-rwth.de

Abstract

This paper presents our approach to create an executable prototype of an enterprise information system based only on a data structure model. This prototype, which is still easily adaptable and extendable, can be used for analysis exploration and builds a solid foundation for the final system. The presented approach transforms a data structure model to changeable and extendable graphical user interface models**.** In a second step, the data structure model and the GUI models are used to generate the resulting system. This approach allows the developer to generate (a) persistence, (b) basic application logic, (c) transportation layers, and (d) a variety of possible graphical representations for the prototype based only on a data structure model. Extensions and changes of the GUI are still possible on model and code level. This is possible by synthetization of GUI models and change operations defined in the same domain-specific language.

Keywords

Domain-Specific Languages, Generative Software Engineering, Graphical User Interfaces, Model-Driven Software Engineering, MontiGEM

Introduction

In general, the use of model-driven approaches increases the adaptability and maintainability of systems (Jun et al. 2005). The use of generative approaches for the creation of information systems has increased in recent years (Hoyos et al. 2017). Enterprise Information Systems (EIS), either as stand-alone applications or accessible via a web-interface, are a prominent class of such information systems.

Research gap

EIS development either (A) focuses on underlying processes and the related data model (Daniel et al. 2016), or (B) includes an intensive design phase for graphical user interfaces (GUIs), interaction and navigation, which results in additional models for describing this (Falzone et al. 2018, Meixner et al. 2011, Schewe et al. 2019). Nevertheless, both approaches need to evaluate user interface concepts with future users to ensure usability and acceptability. A typical approach is the development of prototypes (Sommerville 2007) as shown in Figure 1. A demonstrator is developed and repeatedly refined, in order to work out the desired appearance of the GUI. In a second iteration, a dynamic prototype is implemented and again repeatedly optimized, to

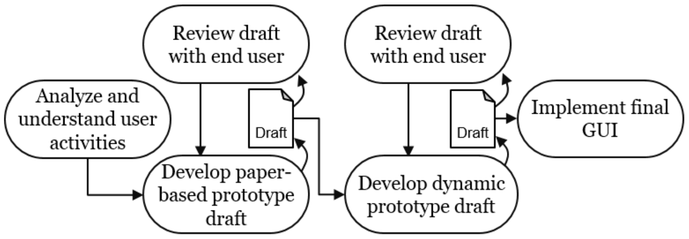

Review draft with end user
Review draft with end user
Analyze and understand user activities
Draft
Draft
Implement final GUI
Develop paper-based prototype draft
Develop dynamic prototype draft

Figure 1. Typical Prototyping approach used for user interface development

showcase possible behaviors of the user interface. The prototypes serve both developer and client to agree upon a common concept but are later replaced. Having a GUI in the early development stages yields a great opportunity to communicate the development process with the client (Wilkinson et al. 2014). To provide these GUIs for future users, even generative approaches need in case (A) GUI mock-ups or intermediate solutions for evaluation purposes and in case (B) a set of GUI-models additional to the data model as well as means to adapt the generated GUIs or add handwritten code. This leads to several challenges:

1. Drafted user interfaces from (A) are hard to continuously maintain during the development process and get outdated very quickly.
2. GUIs created in the late stages of the development process (as in (A) needed) can reveal problems with the user experience that lead to changes in the data model, as specific aspects of the user experience were not considered during the early development stages.
3. Both approaches need additional time: (A) because the GUIs must be replaced by a proper GUI in later development stages and (B) because developers have to define the different models.
4. Having several models in (B), it is important to ensure that they are consistent. This leads to additional effort to check the relations between models and means to inform the developer if there are inconsistencies.

Regarding the challenges, (B) is still a better solution than (A), as a generation of similar code parts in combination with continuous re-generation shortens the development time. Nevertheless, further improvements have to be considered as developers still have to ensure consistency among used models and need additional effort to edit or extend the generated code especially when supporting continuous re-generation and iterative development.

Relevance for practice and research

In practice, there is a big demand to shorten development time and speed up time to market. Shortened iterations of software updates require flexibility in platform development, which we attempt to provide via continuous Model-Driven Software Engineering (MDSE).
As a research project, we attempt to maximize the ratio of model-based generated code, while still providing an extendable flexible code base for real-world applications. We strive to define the software with few models and provide a generator. This work investigates model-to-model transformations and its implications to reduce complexity in the development process.

Research question

How can model-driven software engineering support agile evolutionary development for enterprise information systems?

Contribution

In this paper we focus on a generator-based approach with two main aspects. First: Capability to generate a viable prototype for an EIS based only on a small number of models. Second: Models that are generated themselves within the approach must be editable and extendable.
We use a data model first approach: we generate persistence, basic application logic, transportation layers, and a variety of possible graphical representations (represented as GUI-models) for the system based only on a data structure model. This allows creating a prototype of an EIS for analysis exploration quick and easy. The presented approach transforms a data structure model to a set of changeable and extendable GUI-models using the domain-specific language (DSL) GuiDSL. In a second step, the data structure model and the GUI-models are used to generate the resulting system. Extensions and changes of the GUI are still feasible on model and code level. This is possible by synthetization of GUI-models and change operations defined within the same DSL, creation of additional GUI-models by hand, and addition of handwritten code towards the final system. Our approach allows continuous agile evolution from a prototype towards a full-size real-world information system as additional models are integrated into the generation process and handwritten code is never overwritten by the generator. There is no need to discard existing prototypes.

Overview

This paper is structured as follows: In the section (*Foundations*) we introduce code-generation from models with MontiGEM and the DSLs that are involved in our use case. In section 3 (*Approach*) we present our approach and highlight the specific requirements for the creation of the DSL and the generator. Hereafter, we discuss the evolution from a prototype to a productive system in section 4 (*Transitioning from Prototype to Productive System*). In section 5 (*Discussion and Related work*) we provide a discussion, pointing out benefits and current limitations concerning other approaches and finally conclude.

Foundations

In this chapter, we explain the basic principles of the generator in use and how to add handwritten models. Moreover, we introduce the two DSLs used to realize our approach, namely UML/P class diagrams, a class diagram version optimized for code generation, and GuiDSL for user interfaces.

MontiGEM

Our approach uses the generator framework MontiGEM, a Generator for Enterprise Management (Adam et al. 2018, Adam et al. 2019, Gerasimov et al. 2020) which is able to generate an EIS. An EIS is an application that provides a centralized and organized data view for enterprise processes to different user groups. Therefore, it is fundamental to be able to create, read, update, and delete the underlying data (CRUD). Within the scope of this work we focus on web application EIS, which have the advantage to run on a multitude of different environments.
The Java-based framework MontiGEM (Figure 2) is based on MontiCore (Haber et al. 2015, Hölldobler et al. 2017), a workbench for modeling language development features the agile and compositional development of DSLs (Völter et al. 2013) by providing useful tooling. MontiGEM must be configured by a software developer, it is not yet intended to be calibrated by the end-user of the web application. There are three major aspects (see Figure 2): (1) A set of models used as input, (2,4) the generator itself consisting of a parser, transformer and template engine, (3,5) a set of intermediate models and configurations for the generators and (5) the target as the output files for the generator.

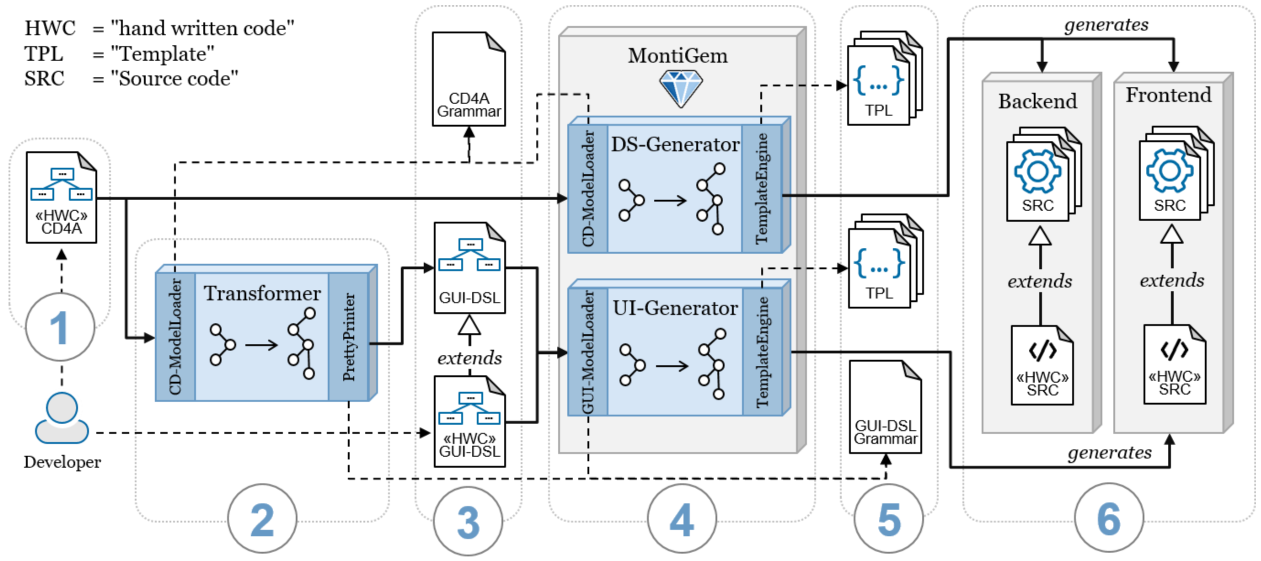


Figure 2. Generator environment.
(1): Input models, (2, 4): Generator, (3, 5): Configuration (6): Output source code.

Input: Models. A developer provides textual models of her corresponding domain (1). In our use case, a developer provides a class diagram for the data structure in the DSL Class Diagram for Analysis (CD4A). Further models for different aspects can be defined and added, e.g. models for validation logic or user access management, as long as the generator is provided with corresponding tooling to process given models. Note that in this approach the class diagram is the only mandatory model.

Generator. Each generator is based on the grammar for each DSL, for which MontiCore creates the basic environment, e.g., a parser, abstract syntax and symbol table. The generator parses each provided model with the ModelLoader and produces for each input a DSL-specific abstract representation of the model, the abstract syntax tree (AST). The transformer (2,4) converts the ASTs into target ASTs. Traversing each AST, the template engine provides target code using target language-specific templates provided by the developer. Within our approach, we use two sets of generators: Model-to-Model (2) is a generator used to transform input models from one DSL to output models of a different DSL. Model-to-Code generators (4) are generators that transform models into target code.

Configuration. MontiCore generator can be configured using DSL grammars. The output is additionally based on a set of templates using the template engine to configure the output format. The templates (5) can be changed if the target programming language changes.

Output: Source Code. Depending on the configuration and the used templates, the generator generates source code for the frontend and the backend of the application (6). Additionally, MontiGEM automatically provides a large amount of boilerplate code, reducing the workload from the developer. Due to the combination of templates and models, the code is consistent by construction and reacts well to changes in the models. The generated output should not be edited directly as any changes would be overwritten in the next generation-cycle. Before creating the output files, the generator detects handwritten classes. In consequence, it adjusts the output for those classes to a super class that can be extended by the according handwritten class (see TOP-mechanism, Hölldobler 2017). Thus, generated code can be extended without having to adapt it directly and handwritten code is never overwritten by the generator.

CD4A

```
1  classdiagram Example {
2   class Account {
3    String name;
4    int balance;
5   }
6
7   class Person {
8    String name;
9   }
10
11   class Accountant extends Person {
12    String bank;
13   }
14
15   association [1] Accountant ->
     (account) Account [*];
16 }
```

Table 1. Example of a CD4A Model

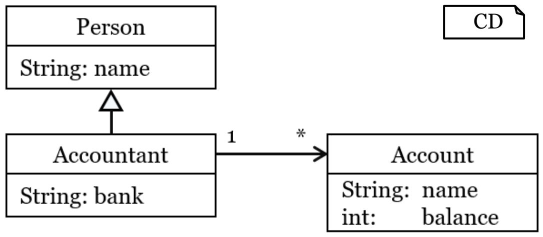


Figure 3. Example class diagram for Table 1

Class Diagram for Analysis

The DSL Class Diagram for Analysis (CD4A) is a textual language, which allows defining class diagrams (OMG 2017) in a machine-readable manner for analysis (language family UML/P). It is based on the UML standard for class diagrams and was adapted to have a more Java-like syntax (Rumpe 2016). Table 1 shows such a CD4A model and Figure 3 displays the visual representation of the given model. The first keyword class diagram marks the start of the class diagram itself. Within a class diagram, we can define e.g. classes interfaces, enumerations, and associations. An example for a simple class is shown in line 2-5 (class Account). The class Account has two attributes: name and balance. Moreover, CD4A provides constructs to define inheritance, by using the keyword extends (12) and associations (16) between the data classes (Figure 4). Multiple Context Conditions (CoCo) ensure unique class- and attribute names and test if used types are either predefined Java types (String, int, long, Date, ...), types defined in the class diagram or imported types from other sources.

Graphical user interfaces

A GUI is used to provide the end-user with a useful interface to interact with the application. Such GUIs can present different Views on the data. An important part is the connection between the View itself and the interaction with the user and the application. A GUI consists of basic components (GUI elements) i.e., `text`, `tables`, and `buttons`, which are oftentimes combined in a GUI framework to a complete view. They provide their own basic functionality such as displaying values or execute a method call on click. To provide a well-structured application, where the logic is separated from the GUI, the Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) patterns have been established (Figure 4). Following these patterns, it is easily possible to exchange the GUI framework, application logic, or parts of the persistence. The data provided for the GUI is called ViewModel and contains the information needed to display all relevant information for a specific view. The introduction of GUI-models and usage of a generator can further improve the configuration possibilities and simplify the coding process for GUIs. A generator is able to generate all connections to the data models and provide boilerplate code, such as data retrieval and communication.

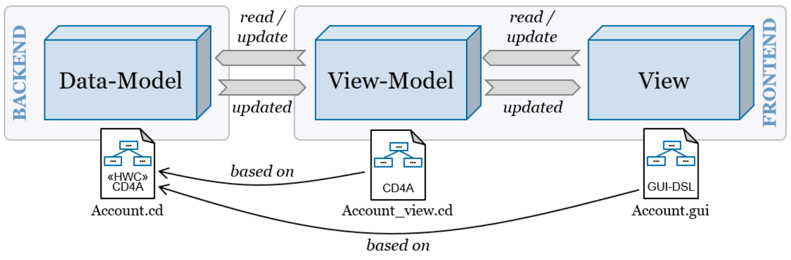


Figure 4. The structure of the Model–View–ViewModel (MVVM) pattern used by the application.

Adding handwritten GUI-models

In this work, we focus mainly on the adaptability of the generated models, but also have to consider handwritten models. A user interface consists of a ViewModel and a View. In our approach both can be customized: The developer can add handwritten Views (GUI-Models) and define custom ViewModels. Based on the ViewModel (Figure 4) the generator provides the infrastructure. In the backend, a ViewLoader contains the logic defining how the data from the original data model is gathered and processed. Based on the ViewModel, the communication for the frontend and the backend is generated, granting the frontend access to the aggregated data. The generated ViewModel can be referenced in the GUI-model. This enables the developer to display custom views, based on the previously defined view model.

GuiDSL

The Graphical User Interface Domain-Specific Language, GuiDSL, is a textual DSL. It can be used to define the appearance of a web application, following an aspect-oriented modeling (AOM) approach (Wimmer et al. 2011). A web application normally includes a variety of different views that are shown in web pages. With the GuiDSL it is possible to describe each view as a separate model (GUI-model).

Structure

GuiDSL models were developed for web applications and follow basic concepts of web design. Elements are nested within each other resulting in a tree-like structure with the `Page` as root (Figure 5). Branching off from themselves a `Page` and `PageElements` can contain further nested elements. These `PageElements` are the basic building blocks for the GUI-model. Examples are *buttons*, *tables,* and *charts*, but can also be layout defining elements such as *containers*, *rows,* and *columns*. Next to the basic `PageElements` there are input elements. They are used in PageElements and can be components such as *textareas*, *checkboxes* or *dropdown menus*. To give an example for the GuiDSL: The web page shown in Figure 6 can be deconstructed into a tree structure, as shown in Figure 5. Note that the same GUI can be represented with multiple different tree structures.

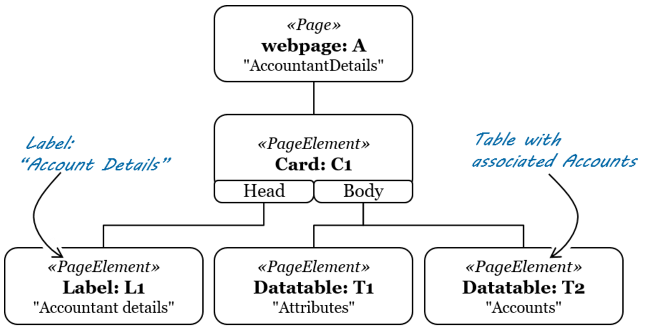
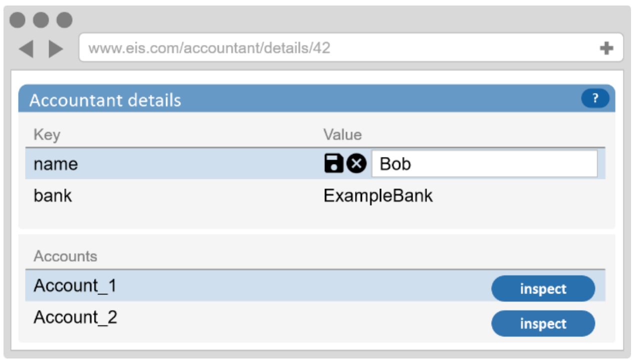


Figure 5. Tree structure of GUI-Model in Figure 6



Key	Value
name	Bob
bank	ExampleBank

Accounts
Account_1
Account_2

Figure 6. Example GUI (Details Page)

Domain-specific language

The GuiDSL defines a multitude of page elements that can be nested in multiple ways. To give an example: The web page shown in Figure 6 can be defined by the textual model in Table 2. Starting in line (1), the keyword `webpage` denotes the beginning of the GUI-model. `Account acc` (1) denotes the data provided for the GUI. The keyword `byId` indicates that data is gathered for one specific object, the keyword `all` indicates the gathering of the complete list of an object class. The box with the title "Accountant details" is defined with the `card` (2) keyword. A card consists of a `head` (3) and a `body` (8). In this example, the head contains a `row` (4) with a `label` (5). In Figure 6 we have two tables, therefore we define two `datatables` (9,15) within the `body` (8). Within the data table, we assign the data transfer object (DTO) of the Accountant `acc` (1,10) as input for the columns. Therefore, we define what data to display. The data tables presented in this paper do not only display data but also provide functions to edit the shown values and save changes in the database. Line (11) defines `attr_name` as key; in Figure 6 the column title is named according to that. The attribute `attr_value` is marked as `editable`, causing the generation of editing functionality for this attribute. We can add buttons to a row with the keyword `button` (18) followed by the button title. Additionally, an empty method body is created for the method `navigate()` (19), which is called after a click and can be completed with handwritten code. Alternatively, the method body can also be provided within the model. Note that it must be written in the programming language of the target code as it will be simply passed through. The GuiDSL provides a wide variety of components enabling the developer to model a wide range of web pages and generate a graphically consistent web application while reducing the required frontend implementation to a minimum.

GuiDSL

```
1 A: webpage AccountantDetails(byId
   Accountant acc) {
2 C1: card {
3  head {
4  R1: row (stretch) {
5   L1: label "Accountant details"
6  }
7  }
8  body {
9  T1: datatable "Accountant attributes" {
10  columns acc {
11   R2: row "Key", attr_name
12   R3: row "Value", attr_value(editable)
13  }
14  }
15  T2: datatable "Accountant accounts" {
16  columns acc {
17   R4: row "Accounts", name(editable)
18   R5: row "", B1: button "Inspect" {
19   click -> navigate()
20   }
21  }
22  }
23 }
24 }
25}
```

Table 2. Generated textual GUI-Model for class Accountant (Figure 6, Table 1)

Approach

Our approach allows to generate a viable prototype for an EIS based only on a small number of models. We show how to generate models for the GUI from data models. Moreover, we present the changes to the GUI generator itself to keep the models editable and extendable. For our approach, we follow a goal oriented experimental research methodology and validate it by example and discussions.

Prototyping user interfaces

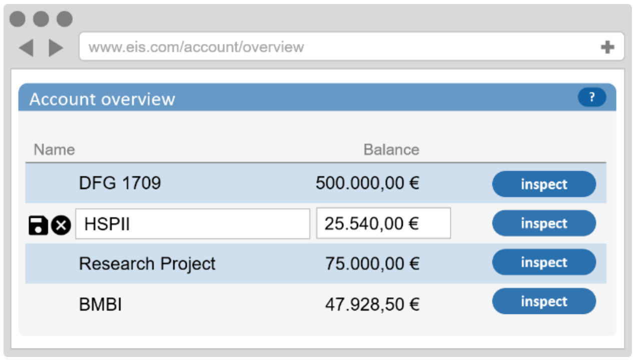
In order to be able to support the manual creation of GUI-models, an additional generator is used to create those based on the data model (Figure 2, (1)). The data model already contains all required information to create a set of default user interfaces (Views) for the end user, that provide:
1: (Details Page) Comprehensible CRUD functionality on all available data
2: (Overview Page) Simple overview of current data sets
3: (Navigation Page) Means to navigate between data sets

Creating the GUI-models

In order to build each a GUI-model for each View, we use a set of templates to create multiple GUI-models for each class. The EIS provides views of the domain data, therefore we create three types of models: (1) A *details page* for a single "class object" (see Figure 6). (2) An *overview page* for all instances of a class (Figure 7). (3) A *navigation page* providing an overview of all generated classes (see Figure 8). This approach provides web pages for all instantiable classes. There will be no page for interfaces or enumerations.
(1) Details Page: For each object, a *details page* as depicted in Figure 6 is generated. This page lists all attributes and associations of an object. The example shows the page of an `Accountant` object as defined

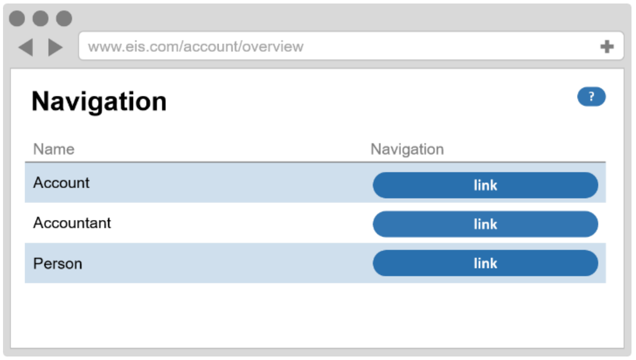
in Table 1. `Accountant` has the attribute `bank` as it inherits from `Person`, it also has the attribute `name`. One Accountant can manage multiple `Accounts` therefore a list is generated, showing all associated objects. Similar to the overview page this list also serves as a navigation to the details page of each object. The four different types of associations are handled as follows: *One to one:* Include a key-value table for the associated object with attributes. *One to many*: Include a list of associated objects with respective attributes. *Many to one*: Include a key value table for the associated object with attributes and link to the overview page of the associated class. *Many to many*: Include a list of associated objects with respective attributes and link to the overview page of the associated class. The mapping is based on the work of Reiß (Reiß 2016). The GUI will display all available attributes as default and could be reduced using the Tagging language (Greifenberg et al. 2015) or by hand if necessary.

(2) Overview Page: The overview page represents the instances of a class in a table. The class `Account` from Table 1 is represented as the page in Figure 7. The attribute values of each object are column-entries of the table. In our example `Account` has the two attributes `name` and `balance`. Each list entry also serves as a means to navigate to the details page for the specific object.



Name	Balance	
DFG 1709	500.000,00 €	inspect
HSPII	25.540,00 €	inspect
Research Project	75.000,00 €	inspect
BMBI	47.928,50 €	inspect

Figure 7 Card with a list of Account objects with their respective attributes defined in Table 1



Name	Navigation
Account	link
Accountant	link
Person	link

Figure 8 listing available Overview Pages and providing links

(3) Navigation Page: In the previous two page types, we already mentioned means to navigate between objects of the data structure, but we also have to consider navigation between objects that do not have any relation to each other within the class diagram serving as an input for the generator. Therefore, a navigation page is generated (see Figure 8). Providing a flat overview of all classes, linking to the overview pages for each class. Figure 8 displays the GUI resulting from the Class Diagram as shown in Table 1.

Creating routing

When providing handwritten GUI-models for a web application, the developer decides at which URL a specific page can be found. We configure the generator to place every page in the sub-URL followed by the class name and the page type and if needed the object-id. Thus, the overview for the class `Account` could be reachable at the URL `/generated/account/overview`. A detailed view of an object with the object id 42 of the class `Account` could be reachable at the URL `/generated/account/details/42`. The routing can be extended or adapted by adding additional routes for the pages within the handwritten code segments of the application.

Transitioning from prototype to productive system

In order to create a product, we need a transition from the generated prototype to a full-size real-world information system. The generated software product is formed by the input models and can be supplemented by additional handwritten models and code. Considering the data structure model, we can directly edit the model and use the TOP-mechanism to extend the generated code with custom code. In case of the generated GUI code, we can also use the TOP-mechanism and adapt the generated target code by extending it, but we should not directly change the generated GUI-model itself. As it is a generator output, any changes would be overwritten with each iteration of the generator. The TOP-mechanism is not applicable to the GUI-model, due to its nested tree structure. In the following, we present a set of operations to modify the GUI-models generated by MontiGEM.

Adapting a GUI-model

Typical adaptions that need to be possible are: (a) Adding and removing elements in the user interface, (b) changing the arrangement of the elements, and (c) changing the configuration of elements. Our goal is to provide a simple set of operations for the developer to configure the generated model to her needs, by reusing as much of the generated model as possible and only changing unwanted parts. The GuiDSL can be represented as a tree structure. Thus, adapting it will result in change operations on a tree.

Editing the tree structure

We extend the GuiDSL with a set of five keywords useable for the developer, to enable her to adapt the model and add unique identifiers to each page element. This provides the capability to write a new model containing only the changes upon the generated one. The handwritten GUI-Model-Adaptation itself is also a valid model. Its output overwrites the target code that is generated by default. Note that any changes to a node in the tree always affect its subtrees. Within a GUI this hierarchical approach is necessary, as the elements often rely on the context, they are displayed in e.g. removing a dialog, but keeping its buttons and its text in place leads to an invalid structure. Therefore, the entire subtree needs to be edited as well. The five change operations are *remove*, *replace*, *add_to*, *add_before*, *add_after* (see Figure 9), and the possibility to reference elements directly.
remove (Figure 9.1)**:** Removes a specific page element and its children elements from the tree.
replace (Figure 9.2)**:** Replaces a specific page element or subtree with another one or with a handwritten segment.
add_to (Figure 9.3)**:** Adds a page element or a handwritten element as a child node to a page element.
add_before (Figure 9.4)**:** Adds a page element or a handwritten segment on the same level of a specific page element on the left-hand side in the same subtree.
add_after (Figure 9.5)**:** Adds a page element or a handwritten segment on the same level of a specific page element on the right-hand side in the same subtree.
Upon parsing the generated GUI-model, the generator checks for handwritten GUI-model adaptations with the same file name. If an extension to a generated model is found, the changes will be executed on the tree structure (Figure 2.3). The adapted tree structure is used to generate the GUI.

GUI-model composition

The GuiDSL can be used to model a wide variety of GUI components and the developer might want to reuse parts of already existing models. Therefore, we introduce the symbol “@” to the GuiDSL to signal the inclusion of an external component within a model. A reference used in this manner has to uniquely identify the exact component, which could point to a subtree of a model or even the entire structure. The example in Table 3 adapts the GUI model in Table 2. It removes the second data table (line 1), adds a custom segment to the model (line 2-8), and reuses the label component “L1” (line 5).

Discussion and Related work

It is not practically feasible to compare this approach with a classic development process in the same project - which would have the strongest evidence. Therefore, no detailed evaluation can be carried out which is methodologically unassailable. The approach presented in this paper inspects the generation of models for the user interface instead of generating the user interface directly.

```
GuiDSL
1 remove @T2
2 add_before @C1 [
3 row (r, 50%) {
4  button 'B2' { click -> doSomething2() }
5  @L1
6  button 'B3' { click -> doSomething3() }
7 }
8 ]
```

Table 3 Handwritten adaptation for Table 2

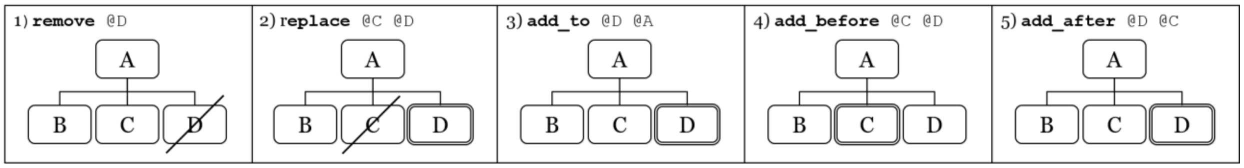


Figure 9 Actions upon the GUI-models

- *Development*: This approach delivers a functional basic prototype for a web-based EIS. The automatically generated user interfaces are based on the default configuration of the generator. MontiGEM supports the development of very generic functionality, it provides little support for specific and individual modifications.
- *Maintainability*: On the one hand side this approach automates large parts of the development process, but on the other side it can make debugging more difficult, as a combination of multiple models can now be responsible for one line of target code.
- *Model-to-Model-Transformation*: By transforming the data model into a GUI-model, we remain more flexible in choosing the target GPLs, as the transformation only takes place between two DSLs. Additional tooling can now use the resulting models and process them further.
- *Scalability*: In principle, our approach has no restrictions regarding the size of any model or the resulting system. However, an optimization for data queries might be needed.

Limitations

The generator provides default user interfaces for the data model. It most likely needs further configuration to suit the needs of the end-users. Thus, the *quality* of the EIS generated based only on the data model is *limited by the quality of the data model* and its default configuration. In its current state, the generator does not provide a dedicated *mobile view* for the generated websites. This is technically feasible as the frontend framework would allow this. In contrast to existing approaches, this framework facilitates the transition to a full-sized real-world system, due to the extensibility of its GUI-models.

Similar frameworks

Web development and prototyping are well-discussed topics. Therefore a few frameworks exist that target similar challenges as MontiGEM. Popular ones are Ruby on Rails, Django (Plekhanova 2009), OpenXava, Vaadin, or JHipster. *Ruby on Rails* (RoR) (Bächle et al. 2007, Viswanathan 2008) can be used to develop web systems in a fast and efficient way. In contrast to the generator-based approach, it faces problems like scalability and mandatory knowledge of the GPL Ruby. RoR enables to build a prototype for a web application whereas the developer needs basic knowledge of Ruby but she cannot generate a functioning EIS prototype based on a data model alone. RoR does not use a data model as input to generate its database but works on migration operations to manage the database, making it harder to scale up. *Django* is similar to RoR , except for the used GPL Python (Askins et al. 2006). In this case, the developer has to get familiar with Python, before being able to configure the framework. This binding to Python also results in slightly slower performance compared to other frameworks. Django is well suited for prototyping but cannot generate a prototype EIS only based on the data model, similar to RoR. *OpenXava* can be combined with “MOSKitt Code Generation Module” and Sketcher2UIM (Gjoni 2015) to generate enterprise web applications, with custom user interfaces. Although this combination can be used to also produce an EIS, it is less flexible in its development process. The separate models for separate aspects of the final target code must be kept compatible by the developer throughout changes. *Vaadin* is a framework for web applications that focuses on the simplification of GUI specifications in rapid web application development. Vaadin does not generate code based on UML models. *JHipster* is a similar framework as Vaadin. It provides a platform to rapidly develop Web applications based on Entity-Models.

To the best of our knowledge, our approach is the only one which keeps models during all iterations until the final product.

Conclusion

We have shown how to generate a viable prototype for an EIS based only on a small number of models. We have adjusted the GuiDSL generator and provide means to use and adapt GUI-models to speed up the development of an EIS prototype-based only on the data structure model. Although the EIS prototype serves as a demonstrator, it provides enough adaptability to be the foundation for further software development, reducing the need to rewrite already existing code, thus reducing the workload on the developer. Having a functional GUI in the early development stages supports the agile development process, iterative methods, and continuous re-generation. It also eases the required changes to the product in later development stages. By design, the input models can be modified and extended, resulting in iterative changes in the target code, while remaining an MDSE approach. For the *generator*, it would be interesting to provide multiple models and views the developer can choose from. Based on the type of data defined in the data model, the transformer ((2) in Figure 2) could provide an extended set of views, such as different diagrams or tables, similar to a 150% model (Grönniger et al. 2008). Additional handwritten GUI-models currently require a view model that is defined separately. Additionally, there exists the tagging language to define the visibility of elements.

REFERENCES

Adam K., Michael J., Netz L., Rumpe B., and Varga S. 2019. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. *Enterprise Modeling and Information Systems Architectures* (EMISA'19). (in press)

Adam K., Netz L., Varga S., Michael J., Rumpe B., Heuser P., Letmathe P. 2018. Model-Based Generation of Enterprise Information Systems. *Enterprise Modeling and Information Systems Architectures* (EMISA'18). 2097. CEUR-WS.org, 75-79.

Askins B., Gree A. 2006. A rails/django comparison. *The Open Source Developers' Conference Papers*.

Bächle M., Kirchberg P. 2007. Ruby on Rails. *IEEE Software* 24, 6 (Nov 2007), 105–108.

Daniel G., Sunyé G., Cabot J. 2016. UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases. *Conceptual Modeling*, LNCS 9974. Springer International, 430–444.

Falzone E., Bernaschina C. 2018. Model Based Rapid Prototyping and Evolution of Web Application. *Web Engineering*, LNCS 10845. Springer International, 496–500.

Gerasimov A., Heuser P., Ketteniß H., Letmathe P., Michael J., Netz L., Rumpe B., Varga S. 2020. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. *Modellierung 2020* Short, Workshop and Tools & Demo Papers. CEUR-WS.org.

Gjoni, O., 2015. Comparison of two model driven architecture approaches for automating business processes, Moskitt Framework and Bizagi Process Management Suite. *Mediterranean Journal of Social Sciences*, 6(2), 615.

Greifenberg T., Look M., Roidl S., Rumpe B. 2015. Engineering Tagging Languages for DSLs. In: *Conf. on Model Driven Engineering Languages and Systems* (MODELS'15), ACM/IEEE.

Grönniger H., et al. 2008. Modeling variants of automotive systems using views. Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF), *Informatik Bericht* 1, 76-89.

Haber A., Look M., Nazari P., Perez A., Rumpe B., Völkel S., Wortmann A. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. *Model-Driven Engineering and Software Development Conferenc*e. SciTePress, 19–31

Hölldobler K., Rumpe B. 2017. MontiCore 5 Language Workbench Edition 2017. Shaker Verlag.

Hoyos J.P., Restrepo-Calle F. 2017. Automatic Source Code Generation for Web-based Process-oriented Information Systems. In *ENASE 2017*, SCITEPRESS, 103–113.

Jun Y., Jarzabek S. 2005. Applying a Generative Technique for Enhanced Genericity and Maintainability on the J2EE Platform. *Generative programming and component engineering*, LNCS 3676.

Meixner G., Paternò F., Vanderdonckt J. 2011. Past, Present, and Future of Model-Based User Interface Development. *i-com* 10(3), 2–11.

OMG Object Management Group. 2017. OMG Unified Modeling Language (OMG UML).

Plekhanova J. 2009. Evaluating web development frameworks: Django, Ruby on Rails and CakePHP. Institute for Business and Information Technology (2009).

Reiß D. 2016. Modellgetriebene generative Entwicklung von Web-Informationssystemen. Shaker Verlag.

Rumpe B. 2016. Modeling with UML: Language, Concepts, Methods. Springer International.

Schewe K., Thalheim B. 2019. Design and Development of Web Information Systems. Springer.

Viswanathan V. 2008. Rapid Web Application Development: A Ruby on Rails Tutorial. IEEE Software 2

Völter M., Benz S., Dietrich C., Engelmann B., Helander M., Kats L., Visser E., Wachsmuth G. 2013. DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org.

Wilkinson C., Angeli A. 2014. Applying user centred and participatory design approaches to commercial product development. *Design Studies* 35(6), 614–631.

Sommerville I. 2007. Software Engineering, 8th Edition, Pearson Studium

Wimmer M., Schauerhuber A., Kappel G., Retschitzegger W., Schwinger W., Kapsammer E. 2011. A survey on UML-based aspect-oriented design modeling. *ACM Comput*. Surv. 43(4), 28:1-28:33