



Lessons Learned from Applying Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator Framework

Constantin Buschhaus, Arkadii Gerasimov, Jörg Christian Kirchhof, Judith Michael, Lukas Netz, Bernhard Rumpe, Sebastian Stüber

Software Engineering, RWTH Aachen University, Germany

Abstract

We report on our success stories in developing and using Model-Driven Engineering (MDE) tools for information systems on real-world projects within different application domains. It is necessary that we ensure the extensibility and adaptability of code generators if we want to reuse them for different domains. Up to now, research on reusing software has been mainly conducted in the software product line community but rarely discussed in the context of code generators. This paper introduces the generation framework MontiGem and shows how it has been used and evolved within five different research and industry projects in the domains of financial management, IoT, energy management, privacy policy, and wind turbine engineering. We have developed the code generator within the first project and further refined it with each of the following projects. This paper describes the projects, shows how MDE helped us in the software engineering process, and discusses the lessons we learned. These examples show how MDE techniques can be successfully applied to the development of information systems in practice, although further requirements have been met over time.

Keywords: Model-Driven Software Engineering, Code Synthesis, Domain-Specific Languages, Financial Management, Internet of Things, Engineering, Wind Turbines, Energy, Privacy Policies

1. Introduction

While Model-Driven Engineering (MDE) [1] is a well-established method in academia [2, 3, 4], we have a specific interest in applying these methods to full-size real-world projects. MDE is often applied to develop web-based applications and information systems for different application domains [5,

6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Using models as core artifacts in the engineering process of complex, software-intensive systems allows us to (1) handle complexity by raising the level of abstraction, and (2) continuously evolve the models and the code according to changing requirements. Whereas some critique on the use of models in software development argues that it leads to heavy-weight, tedious development processes, research has already shown that it is well applicable for agile development methods in different domains [16, 17, 18, 19, 20, 21].

As the development of code generators needed in MDE approaches requires an additional effort, it is advisable to reuse the generators or at least parts of them for similar projects [22]. The architecture is a characteristic that such projects often have in common, as these aspects are not generated but defined in the Run-Time Environment (RTE) of an MDE application. In practice, this raises the question of to which extent code generators are reusable. As for other software projects, quality aspects such as extensibility and adaptability also apply to these software systems.

This article presents the generator framework MontiGem [23], which aims to generate data-centric web applications. We used the framework within five projects targeting different domains. Within the Management Cockpit for Controlling (MaCoCo) project [24], the first version of the generator was developed to create an application for the financial management of university Chairs. Within the Ford project about the tracking of IoT devices, we reused MontiGem for the first time and, e.g., provided multilingual support of the Graphical User Interface (GUI) and adapted the appearance of the frontend for mobile platforms. Within the energy management project, we use a specific version of MontiGem, which requires only a data model as input. We connect the generated application to an existing framework and add several visualization components. In the InviDas project, MontiGem is used to generate a platform for sovereign decision finding regarding privacy policies of smart wearables where external partners develop parts of the frontend. In the Agile Data Dev (ADD) project, we create a digital twin cockpit for parameter management in wind turbine engineering and connect it with existing tools. We present twenty lessons learned from applying the model-driven approach while using and further developing the MontiGem generator in these projects. We discuss the lessons in the context of the state-of-the-art in MDE and software engineering.

The article is structured as follows: The next section details the background, such as developing Domain-Specific Languages (DSLs) with the MontiCore language workbench and MontiGem, the generator for data-centric applications. The following sections describe the lessons learned in 5 projects we used MontiGem in, namely for the financial management of university

Chairs in Section 3, for tracking IoT devices in Section 4, for energy management in Section 5, for privacy policies in Section 6, and for the parameter management in wind turbine engineering in Section 7. We are discussing the lessons learned in Section 8 and comparing them with related work in Section 10. The last section concludes.

2. Background

Our projects use several DSLs, the language workbench MontiCore, and the generator framework MontiGem as tools for model-driven development. This section briefly explains the technologies and how they are used.

2.1. Domain-Specific Languages

DSLs are modeling languages devoted to a given application domain [25]. In contrast to a General Purpose Language (GPL), they are used to create models that fit the specific domain (thus also referred to as DSMLs, Domain Specific Modeling Languages). The advantage of modeling in a DSL (as opposed to Java, for example) is the benefit of modeling much closer to the actual domain, making them easier to understand and communicate the models, especially to people not familiar with software development [26, 27]. In the context of this paper, we will take a closer look at two DSLs used to define the target applications: Class Diagrams for Analysis (CD4A)—a modeling language for class diagrams, and GUI DSL—a language to define user interfaces.

CD4A is a textual DSL, based on UML [28] to define class diagrams in a Java-like syntax [29]. Aside from attribute- and class definitions, CD4A supports all common elements of class diagrams such as *associations*, *inheritance*, and *enumerations*. Complete documentation can be found at [30]. Listing 1 shows a small example of a class diagram consisting of one class `Person` with the two attributes `name` and `age`.

GUI DSL [31] simplifies the development of user interfaces for information systems. The GUI DSL defines models that describe how data should be displayed and manipulated on a screen. Each model defines one information system page with predefined GUI components. Listing 2 shows a simple example of a GUI model, composing a page of a card with the title 'Staff Overview' (label in line 3) and with a simple table (Line 5..8) consisting of attributes that are provided by the `Person` Object declared in Line 1. Figure 1 shows the resulting user interface.

```

1 classdiagram CD {
2   class Person {
3     String name;
4     Date birthday;
5   }
6 }

```

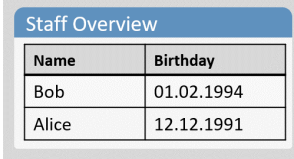
Listing 1: CD4A Class Diagram Defining Person Class

```

1 webpage Staff(all Person p){
2   card {
3     head {label "Staff Overview"}
4     body {
5       datatable { rows <p {
6         column "Name", name
7         column "Birthday", birthday
8       }}}}}

```

Listing 2: GUI-Model



Staff Overview	
Name	Birthday
Bob	01.02.1994
Alice	12.12.1991

Figure 1: Resulting GUI showing Person objects in a Table

2.2. MontiCore

The MontiCore language workbench [32, 33] is designed to facilitate the engineering of textual DSLs. It provides mechanisms for analysis, manipulation, and transformation of the models of a developed DSL. MontiCore generates parsers capable of handling the models of the specific DSLs and infrastructures for transforming the models into their Abstract Syntax Tree (AST) representation and symbol tables. A provided template engine generates resulting code from a source AST and target language templates [32]. To date, MontiCore has implemented a variety of languages [34], including a collection of UML/P languages [29] (variants of UML that are more suitable for programming), as well as the OCL, delta [35], tagging languages [36], SysML, and architectural description languages.

2.3. MontiGem

MontiGem is a generator for data-centric applications. It uses standard models from UML/P as sources, and GUI models to generate web applications (c.f. Figure 2). MontiGem parses each model and produces an abstract syntax tree before transforming it into a target syntax tree. The transformed ASTs are provided to template engines that transform it into the application's source code in the specified target GPL. The generator only creates the domain-dependent code. The framework provides a generic code, such as the run-time environment.

Over time multiple extensions were added to MontiGem. The CD2GUI-Extension provides an additional transformer that derives GUI-Models from Class diagrams. The OCL-Extension can process additional OCL-Models in order to generate additional validators and simple business logic. All extensions are optional and not mandatory for the generator to operate.

The generator framework MontiGem has been developed for several years and was used in different academia and real-world projects as well as teaching activities. We use the MontiGem generator framework as an academic demonstrator and extend it, e.g., to generate process-aware information systems [37], low-code development platforms for digital twins [38], digital twin

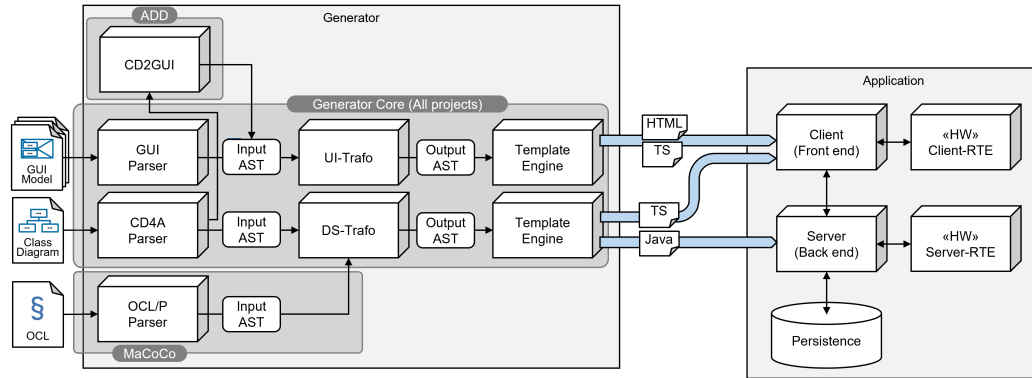


Figure 2: MontiGem generating a web application using a set of models. The framework was used ‘as is’ in Ford (c.f. Section 4), InviDas (c.f. Section 6), and EM (c.f. Section 5). Extensions were added for MaCoCo (c.f. Section 3) and ADD (c.f. Section 7)

cockpits [13, 39], process-aware digital twin cockpits from event logs [40], IoT App stores [41], information systems with privacy policies [42], assistive systems [43], and to integrate goal modeling approaches in information systems [44]. In the following chapters, we look closely at five information systems and describe our experiences applying the model-driven approach using our tool. Each project was developed by small teams (2-20 people), where each team included MDE experts as well as inexperienced developers. The development started with the MaCoCo project that laid a foundation for the MontiGem.

3. MaCoCo

MaCoCo [45] is a joint project of the Chair for Controlling and the Chair for Software Engineering at RWTH Aachen University. The project enables faculties, chairs, and institutes within the university to manage financial resources, staff, and projects and keep track of and document activities, e.g., taking vacations, logging working hours, and managing contracts. Figure 3 shows an example of a web page displaying accounts managed by a faculty (testing data is shown).

MaCoCo is a web-based information system currently used by over 200 institutes and faculties of varying sizes with a growing number of users (Table 1). The web application is mainly generated with numerous views displaying data structured from a CD4A domain model (Table 2). Currently, around 75% of the application code is generated.

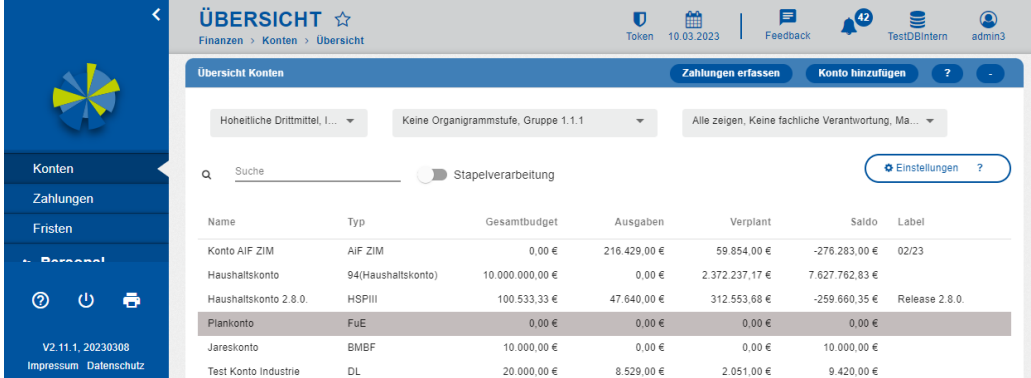
The MaCoCo project is being developed in an environment where requirements constantly change, and new features must be continuously implemented and released. Like in most other university projects, developers

	Total
Faculties	216
Users	1791
Logins per day	200

Table 1: MaCoCo usage statistics

	Total
Modeled classes (domain)	85
GUI-Models (web pages)	85
Lines of Code	1 mil

Table 2: MaCoCo code statistics



Name	Typ	Gesamtbudget	Ausgaben	Verplant	Saldo	Label
Konto AIF ZIM	AIF ZIM	0,00 €	216.429,00 €	59.854,00 €	-276.283,00 €	02/23
Haushaltskonto	94(Haushaltskonto)	10.000.000,00 €	0,00 €	2.372.237,17 €	7.627.762,83 €	
Haushaltskonto 2.8.0.	HSP/III	100.533,33 €	47.640,00 €	312.553,68 €	-259.660,35 €	Release 2.8.0.
Plankonto	FuE	0,00 €	0,00 €	0,00 €	0,00 €	
Jareskonto	BMBF	10.000,00 €	0,00 €	0,00 €	10.000,00 €	
Test Konto Industrie	DL	20.000,00 €	8.529,00 €	2.051,00 €	9.420,00 €	

Figure 3: Screenshot of the Account overview, listing all accounts of the current user.

often change because they are students and graduate at a certain point, with few people remaining as part of the core team. Most developers are inexperienced and have various programming habits, i.e., implement the solutions differently. Bug fixes and most necessary features are expected to be released on demand, while several less critical changes are delivered every 2-3 months.

The project started as a hand-written web application with frontend and backend parts, where the backend was a prototype with simple functionality for financial management. The code consisted of several classes, such as Account, AccountBuilder, and AccountDAO for the creation and access of account objects. At this point, a model-driven approach was adopted for the backend development soon after the prototype was built.

Lesson learned #1: *Model-driven approach pays off not only in the long run but also at the start of a project if basic generation tools and MDE experts are available.*

The pivot toward MDE prevented writing a lot of boilerplate code, such as domain object, builder, and access object classes by hand, thus decreasing the development time and keeping the high quality and consistency of the code. This has become increasingly important in the context of continuously changing requirements. The almost instant migration to the model-driven approach was reinforced by an existing generator for creating data structures, which eventually became one of the core components of the MontiGem frame-

work and is still being used. Although the generator was initially designed for web applications with a different technology stack, taking relevant parts and slightly rebuilding it did not take much effort for an MDE expert familiar with the generator tooling at the starting phase of the project development.

Lesson learned #2: *Features affecting the whole system can be introduced via generator extension but might require additional DSLs for advanced cases.*

The system realization continued for several years; all parts, including the generator, have continuously been refined. The generator provided significant support in the essential part of the application - data structure and surrounding boilerplate code, which was and still is derived from class diagrams. With time other aspects solidified as connected to, but not a part of, the data structure. When the concept of rights and roles was introduced, it was generally incorporated into the data structure generator, but only in a basic form of checking whether the current user has rights for an object of a class. Validation of the user input and subsequent checks in the backend and database had become a separately handled issue. OCL models were defined and served as the source for the generation of validation logic both in the frontend and backend of the application. The elements that can be generated are limited by the information contained within the input models, and additional features might require further models in new DSLs.

Lesson learned #3: *When a project is expected to be developed further, early integration of a generator is critical.*

Eventually, the frontend code grew to a point where maintainability became problematic. Due to the high dynamic in the development team, the code deviated from one web page implementation to another. Most of the work with TypeScript, HTML, and CSS was done by inexperienced students, and although the product was functional, diverse patterns could be seen in the code. As a result, assigning a different task to someone focused on a specific area came with the additional effort of understanding the code and possibly rewriting it. Had we recognized the issue earlier, such confusion and constant re-learning could have been avoided, as we learned after introducing a new DSL and a generator for the frontend development. The DSL hid patterns via abstract concepts and allowed developers to focus on the logical parts of the application frontend, such as navigation, loading data, formatting data, etc. In retrospect, we should have integrated the DSL in the earlier stage as we did with the backend but failed to recognize the necessity.

Lesson learned #4: *Retrofitting a DSL and a generator is difficult but pays off in the long run [46].*

The migration to a new DSL came with a great effort, as the amount of retrofitted code was considerable. Despite having numerous examples of the code to be generated, it was difficult to recognize patterns due to the diversity in the code. We have spent around two additional person months building the new solution to be functional, where the work was done mainly by students with some experience in the project. At the time, less than half of the current views and the data structure classes were present. After integrating a new DSL, the code has become consistent and easy to maintain. Additionally, developers without frontend expertise could now create simple, functional pages, thus allowing a more flexible distribution of programming tasks. Over the years, the generator was further refined, and creating new views is currently the least of our concerns; instead, we focus more on business logic and optimizations.

Lesson learned #5: *A separation of generic and use-case-specific generator code is necessary to derive a general-purpose generator framework from the current use case.*

After three years of developing the MaCoCo project, the generator included many features valid for the automated creation of information systems. We decided to extract the generator and infrastructure needed to build a basic web application into a separate project, which has become the MontiGem generator framework. We faced two major tasks; one was recognizing, removing, and reworking the parts of the generator which assumed the context of the MaCoCo project, namely components working specifically with financial or staff management. Despite avoiding such dependency in the generator, some code, such as creating visualization charts displaying financial information or rights and roles concepts, was generated for every page since it was generally used everywhere in the application. Another issue was to separate standard functionality, i.e., run-time environments, such as client-server communication infrastructure and GUI components, from domain-specific code into a library. Separating the run-time environment took around two person months, and the work was performed by experienced developers. In retrospect, the code could have been developed in its own sub-project without issues, as we experienced using the extracted framework in later projects.

4. Ford - Tracking IoT devices

In collaboration with Ford Research and Innovation Center Aachen, we have developed an IoT application that enables craftsmen to track their valuable tools. Based on MontiGem, we provided craftsmen with web interfaces for desktops, in-vehicle tablets, and smartphones that enabled them to track

which tools were in which vehicles or where tools were last seen. For this purpose, each tool was equipped with a heat and water-resistant Bluetooth Beacon, and the vehicles with a tracking module. This tracking module transmitted information about the tools in a vehicle to the MontiGem application. The MontiGem application displayed live information about the tools' locations in a tile view with photos of the tools and on an interactive map view. Figure 4 shows a worker viewing the application on a desktop. Compared to the previous project, we faced new requirements, such as enabling support for different languages (e.g., German and English), showing maps, and providing mobile application views.

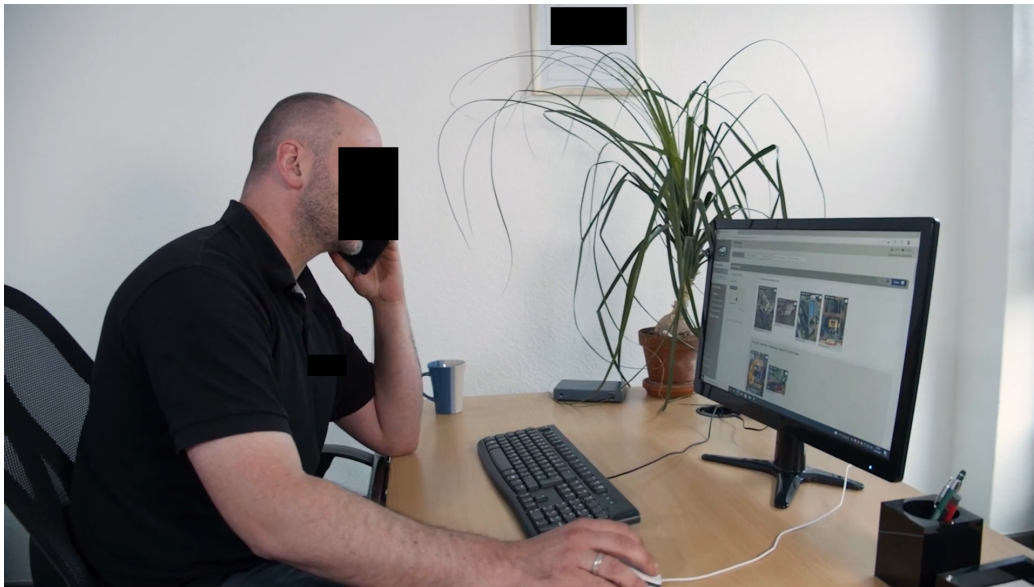


Figure 4: A craftsman at the office working with the web app generated by MontiGem.
Image Source: <https://www.youtube.com/watch/ODuvZ6AahzI>

The application was tested over a period of around 11 months at three craft enterprises of different sizes and from different sectors in North Rhine-Westphalia. Based on the feedback from the craftsmen and the findings from the log data of the application, the application was continuously iteratively developed.

Lesson learned #6: *Models like class diagrams can be valuable communication tools for developing applications in cross-disciplinary teams.*

The development of the tool tracking application has required expertise from a wide range of domains. These include marketing experts, electrical engineers, mechanical engineers, craftsmen, and designers. To create a shared understanding of how the application should behave in which situations, it

was necessary to make the application understandable to a wide range of experts. People without programming experience can easily be put off by general-purpose programming languages or database-specific DSLs like SQL. Class diagrams allowed us to discuss the application's data structures across disciplines. For easier accessibility, we used the models mostly in graphical syntax. We explained concepts like inheritance (A *is a* B) or associations (A *knows a* B) in a basic way.

Lesson learned #7: *If a project has non-standard user experience requirements, consider creating native apps instead of web apps.*

As part of our MontiGem application, we have delivered mobile versions for smartphones and a tablet permanently mounted in the vehicle, among other things. For the tablet version, in particular, delivering a significantly modified user interface was essential. Since the tablet was permanently installed in the car, we had to ensure that the display did not distract the driver while driving, as a tablet-sized version of our desktop version would have done. Therefore, among other things, we measured the vehicle's acceleration to reduce the amount of information displayed while driving. However, since our user group was not necessarily in regions with good mobile connectivity, the server could not reliably guarantee control of the device. In addition, the web application felt partially non-responsive and less natural than a native app. For these reasons, we retired the web version of our application in later versions in favor of a native Android app.

Lesson learned #8: *Highly application-specific parts should rely on handwritten code alone instead of mixing models and handwritten code.*

We could generate large parts of the desired GUI directly from GUI models for many of our websites. For some sites, the UX designers involved in the project created GUI templates that differed significantly from MontiGem's standard elements. These templates could be implemented by integrating handwritten HTML code into the GUI models. However, in the case of strongly deviating specifications, the GUI models were similarly challenging to understand as the resulting HTML code due to the mass of handwritten code. In particular, due to the lack of DSL tool support (e.g., syntax highlighting), we recommend not using a DSL at all in these cases.

Lesson learned #9: *The benefits of model-driven software engineering are sometimes underestimated by outsiders.*

In the early phases of the project, when we had to coordinate fundamental decisions about development methodology and frameworks to be used with Ford, the proposal to develop the application in a model-driven way was

met with some skepticism. In particular, mentioning code generators led to unintended associations for people unfamiliar with MDE. Concern was expressed that the website might not meet the visual specifications and look like a WordPress blog. Furthermore, it was feared that there would be no possibility to continue using the website as soon as our support would be discontinued at the end of the project. With the first versions of our application, we could dispel such concerns and impress with a high development speed. For the future, we recommend keeping demonstrators of the generator's results to be used.

Lesson learned #10: *Provide in-app editors to involve non-experts.*

To involve experts unfamiliar with the respective modeling languages and code generators, offering editors that can be operated directly within the generated web application is helpful. In our specific case, for example, the web application was offered in several languages to be understandable for German-speaking craftsmen and for English-speaking managers of Ford. Using MontiGem's code generator, we systematically generated the necessary code to edit website translations directly within the web application.

5. Energy Management - Visualizing Building Information

In the Energy Management project, the data from various building sensors is visualized on a web page. During the project's setup, each room in the customer's office building was equipped with temperature sensors. In the next phase, a two-step process is executed. First, the collected data is used to analyze the energy use of the building. Second, the heating elements are individually controlled to minimize energy use.

The customer had already built the infrastructure to collect and store measurements using the FIWARE framework. Our job was to create an application to visualize the temperature measurements for which we used the MontiGem framework.

A component displaying an SVG image is used to display the floor plan. An SVG element with a unique ID represents each room. The image is then modified at run-time when the data for the room is received. The current temperature is mapped to the text and color of the rooms on the image. Figure 5 shows an example of the floor plan with temperature measurements.

Besides the current temperature, meta- and historical information is also relevant. Meta information contains the sensor type or size of the room. Historical information shows previous measurements. The GUI-DSL already provided suitable components, as seen in Figure 6. New challenges for the

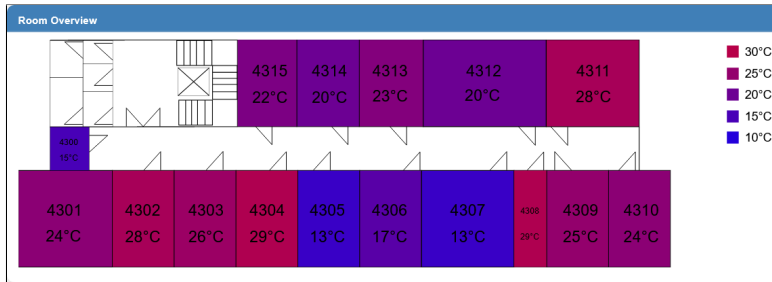


Figure 5: Floor plan with real-time temperature measurements

framework included integrating complex graphics for a floor plan with live updates and stripping out vast portions of the generated server infrastructure.



Figure 6: Meta information of sensors

Lesson learned #11: *Generated components should be modular and replaceable to enable integration with custom user-defined components.*

To tightly integrate the website into the customers existing tooling, the authentication service developed by the customer was used. This way, the user could continue using their existing account and password. MontiGem had tightly integrated its authentication infrastructure into frontend and backend, which was difficult to replace. The modularity also makes it easier to reuse components in other projects. This is especially important as hand-written logic often extends a complex generated component.

Lesson learned #12: *Code generators should be configurable so that no large unused modules are produced.*

The project was heavily focused on the frontend, while the backend only had to proxy requests to the customer infrastructure. Hence, much of the generated backend functionality was never used. For example, only read was operational from the default create, read, update, and delete commands. MontiGem application was not required to alter the data; hence Create/Update/Delete was not used. Unused code slowed the compile time, made the code harder to analyze, and introduced additional security concerns since extra communication endpoints were generated. We learned that the code generator should be configurable to generate only needed functionality.

6. InviDas Platform - Privacy Policies of Wearable Devices

InviDas¹ (Interactive visual Data rooms for sovereign decision finding regarding data protection) is a project about visualizing privacy policies of smart wearables and making them comparable with each other. The project is funded by the German government (BMBF) and is realized by a research association including software engineers, human-computer interaction scientists, and smart wearable manufacturers. This project started when MontiGem was already used by multiple other projects; thus MontiGem improvements were more related to bug fixes and less to implementing new features. The bugs were minor, e.g., automatic redirection of anonymous users to a login page, even though there was no need to be logged in to access the specific part of the website. These bugs did only concern MontiGem’s runtime environment and not the generator itself.

The main result of the InviDas project is a website platform on which smart wearable users can inform themselves about the privacy policies of different manufacturers. The application GUI excerpt is shown on Figure 7. The privacy policies are not displayed in the legal text but as data points of different data processing categories. These categories were identified by analyzing different privacy policies from smart wearable manufacturers. Ultimately, manufacturers should be able to enter their privacy policy into the platform.

A unique project-specific requirement that had not been addressed by the MontiGem before was the integration of privacy policies. Since privacy policies (legal texts in general) vary, much effort was put into finding a standard abstracted model to specify a privacy policy instance. Seven major smart wearable vendors’ privacy policies were analyzed with the GDPR [47] to create the underlying model. This model is called the privacy policy model and is the primary development artifact of the InviDas platform. It is a textual Class Diagram (CD) used to describe the backend data structure. From this CD, MontiGem generates the whole backend apart from custom data transfer object creations.

Lesson learned #13: *Model-driven approach accelerates early phases of the development when domain experts and developers have to communicate frequently.*

Describing unstructured data, e.g., legal text, using structures like class diagrams is difficult. Such a task requires a thorough analysis of the domain and additional knowledge about it, which leads to frequent communication

¹<https://invidas.gi.de/>

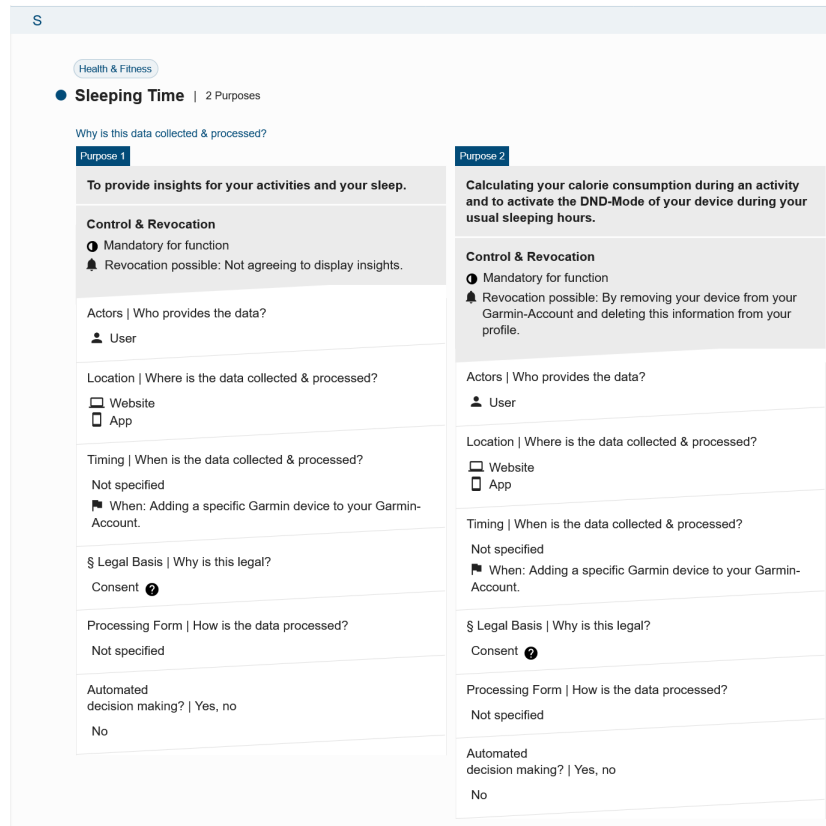


Figure 7: InviDas platform excerpt of the graphical user interface. This excerpt shows a translation of what is stated in the German privacy policy of Garmin Ltd. about the data collection and processing of users' sleeping time.

between stakeholders and changes in the system. Generators like MontiGem are a great way to implement changes to the data model quickly. This happens more often in projects like InviDas during the initial phases because of the complexity of the unstructured data and the additional knowledge about a domain that one builds up over time.

Lesson learned #14: *A code generator has to support hand-written code extensions.*

Similar to the lesson 8, the project required a specific custom graphical user interface design; thus, a hand-written implementation was used. Other parts of the frontend, e.g., the communication to the server, still benefit from the generator since the hand-written GUI can use the generated code and vice versa. MontiGem enables this by using the top mechanism, which is described in [48].

Lesson learned #15: *If different representations of models are used, they have to be synchronized.*

To decrease the entry barrier into modeling, a graphical modeling tool for CDs was used to create the first iteration of the privacy policy model. The textual CD was created simultaneously, but inconsistencies occurred because these models did not automatically update each other. The GUI was designed based on the graphical CD, while the platform was developed with MontiGem based on the textual CD. Because only the textual model was kept up to date, it had gone unnoticed that the design slowly branched off of the data model, and inconsistencies in requirements from the GUI design to the data structure originated. This impeded their integration with each other and slowed down the development of the platform. Literature suggests solutions to this problem by automating the synchronization of the different model representations (model views) [49]. Still, we have yet to address this further in future work.

Lesson learned #16: *Code generators enable building prototypes quickly, which helps to identify problems in different parts of a developed system.*

Simple rendered HTML form and table-based depictions of dummy data as instances of CDs are helpful to let website designers get a feeling of how the data structure looks like. Furthermore, it helps to understand the domain model better because a simple prototype provides insights into such details as correct or incorrect inputs. The domain experts can use it to validate the requirement fulfillment.

7. Agile Data Dev - Data Management for Wind-Turbine Engineering

The goal of the Agile Data Dev (ADD) project is a faster time-to-market and higher efficiency in the development of wind turbines. To achieve this goal, MontiGem was used to create a central point of information exchange [39]. Development artifacts like CAD files, parameter values, and simulation results are stored in the MontiGem database. Compared to the approach of storage on individual engineers' computers, MontiGem enables a faster exchange of current results between engineers. Furthermore, it is easier to archive relevant artifacts of the development process to comply with legal requirements. However, we faced a new challenge of making the storage replaceable. In this project, we also tried to simplify the process of building an application by automatically creating GUI models.

Lesson learned #17: *Data storage and retrieval should be easily integratable into existing tooling.*

Usually, there is already existing data and tooling, which has to be migrated to the new solution. This can significantly burden the development of the new tool, especially when consistency constraints on the data have to be enforced. In the ADD project, most existing tooling was written in Python since the design programs (e.g., CAD software) provide good Python integration. A Python library was developed to connect this tooling to the new MontiGem Server. This Python library replaces local reads and writes with API calls to the MontiGem server.

The screenshot shows a web application titled "Overview page for Pump". It includes a search bar labeled "Suche" and a button labeled "Einstellungen". Below this is a table with the following columns: "name", "idealPower", "minimalPower ^", "maximalPower", and "VolumnFlow". The table contains three rows of data for pumps: MK01, MK12_13, and MK2000. Each row has icons for editing and deleting. At the bottom, there is a pagination control showing "3 Einträge", "10" (selected), and "Einträge pro Seite".

	name	idealPower	minimalPower ^	maximalPower	VolumnFlow
	Name	Ideal Power	Minimal Power	Maximal Power	Volumn Flow
	MK01	30	28	32	20
	MK12_13	300	200	400	40
	MK2000	600	550	700	120

3 Einträge 10 Einträge pro Seite

Figure 8: Generated user interface, representing Pump objects in an overview. [39]

Lesson learned #18: *End users find visual representations for class diagrams more user-friendly than textual representations.*

In the ADD project, the data schema was not fixed. Instead, it was modified by the industry customer according to their needs. Handling the textual CD4A DSL proved unintuitive and cumbersome for the customer as the size of the data schema grew to more than 120 classes and more than 200 associations or inheritance relations. Visualization tools proved to be very useful in this situation. This be realized, e.g., by allowing editing in the textual representation of models for consistency reasons and visualizing the models in a graphical way for end users. Visualizations helped organize the presentation, e.g., by positioning classes, defining views hiding irrelevant information, or grouping elements.

Lesson learned #19: *Generated Graphical Interfaces are great for prototyping.*

Within the ADD, a large set of classes was processed. To rapidly reach an interactive web application, the model-to-model transformer CD2GUI was

developed. It derives GUI models for MontiGem from CD4A models, providing a set of interactive pages for each class as shown on Figure 8. A running application, complete with user interfaces for data entry, allowed stakeholders to evaluate requirements and data structures according to their needs at a very early stage of the project. These prototypes saved time and resources and were a great foundation to discuss further development.

Lesson learned #20: *Real-life projects are a driving force for the development of tools in MDE.*

While CD2GUI was great for prototyping, it was not usable for specific customer requirements, and neither was GUI DSL. Figure 9 demonstrates visual components requested to be integrated into the application. According to the customer's requirements, the display of simulation runs should have been a graph and not a table. Furthermore, the requirements from the customer about how the graph should be displayed were very specific. For example, graphs should be displayed side by side in an easy-to-compare way. Hence, the generated prototype was extended or replaced by more customer-centric components, the same as in lessons 8 and 14. However, it became clear that GUI DSL is not powerful enough to be used in different domains. Thus we have updated the language to enable easier integration of custom components.

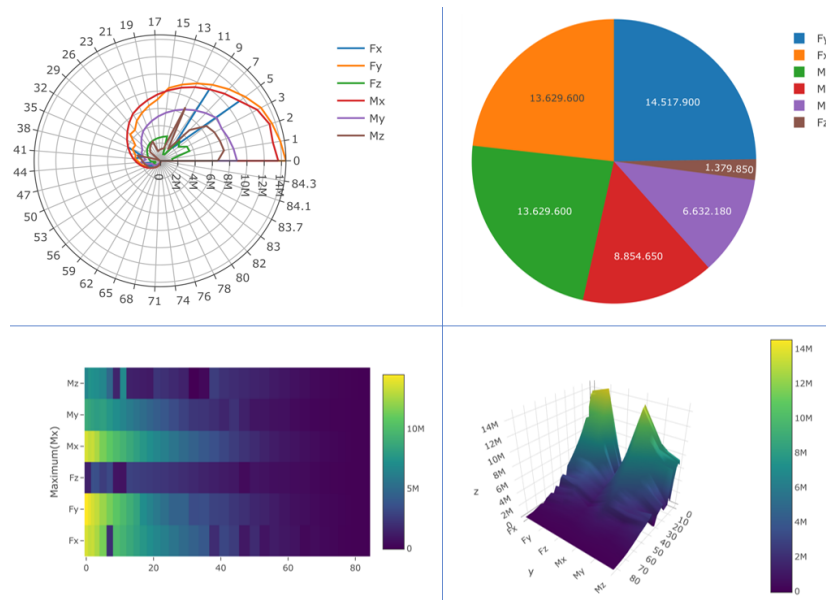


Figure 9: Different Application Specific Visualisations [39]

8. Discussion

The lessons we learned show the aspects of applying the model-driven approach with the highest impact on the projects. They can be grouped into three categories (Table 3): The lessons learned about communication with the customers and users of generated applications, code generator development, and generator integration. As shown in the table, the knowledge gained did not only help with the later projects but was also applied retrospectively. Some lessons, such as the lessons 6 and 16 about using models as communication tools and fast prototyping for identifying problems early, were recognized in later projects as they were significant for these projects in particular.

The rest of the section discusses the three lesson categories, describes their relevance for particular projects, shows the evolution of the MontiGem, and summarizes the global takeaways.

8.1. User Communication: Involvement of non-experts, prototyping

We used the models for communication with the users in almost every project (lesson 6). Synchronizing the project state between the stakeholders using the same artifacts allowed us to specify the project requirements precisely. For us, the developers, the benefit became apparent as soon as we had to communicate with the customers directly in the Ford project. The models as a communication tool were irrelevant to the Energy Management project, as we directly showed the results in the GUI. In every case where the models were used for communication, the customers preferred a graphical representation (lesson 18). It was essential in the ADD project, which had hundreds of classes in the class diagram and required a visualization for a comprehensible overview.

Skepticism against the model-driven approach was observed in the projects that involved customers with programming experience (lesson 9). The biggest concern about applying the approach was most visible in the Ford and the MaCoCo projects, our two biggest and earliest projects. Only minor questions were asked in the later projects, where we could show results from other projects.

Specific requirements in the Ford project led us to add a generator extension for multi-language support, including an in-app editor for adding translations (lesson 10). Given the resources, such tool extensions can be added for convenience. Our other projects focused on data management that did not require additional tooling.

Quick prototyping (lesson 13) becomes possible as the generator framework gets refined. In the early projects, hand-written software prototypes

#	Lesson	MaCoCo	Ford	EM	InviDas	ADD
User Communication: Involvement of non-experts, prototyping						
6	Use models as communication tools	A	L		A	A
9	MDE is easily underestimated by outsiders	A	L			A
10	Provide in-app editors to involve non-experts		L			
13	Accelerated prototyping due to MDE		A		L	A
16	Frequent prototyping helps identify problems early	A	A	A	L	A
18	Use visual model representation for end users	A	A		A	L
19	Generated graphical interfaces are great for prototyping					L
Generator development: Implementing a generator						
1	Amortisation of generative approaches	L	A	A	A	A
2	Develop new DSLs to generate more functionality	L				
5	Separating domain-specific and general code	L				
11	Generate modular and replaceable components			L	A	A
12	Minimize the amount of unused generated artefacts		A	L	A	A
14	Support hand-written extensions of generated code	A	A	L	A	A
Generator integration: Applying MDE to a project						
3	Benefits of early generator integration	L		A	A	A
4	Benefits of retrofitting a generator	L	A	A	A	A
7	Generating native apps instead of web apps		L			
8	Avoid mixing models and hand-written code	A	L			
15	Synchronize separate representations of models				L	
17	Easy integration of data storage and retrieval			A		L
20	Real-life projects push the development of tools in MDE	A	A	A	A	L

Table 3: Listing of the lessons learned presented in this work. 'L' annotates the project this lesson was learned in, and 'A' annotates other projects this lesson also applies to.

were needed to develop the generator. Eventually, the generator development did not need these prototypes anymore because it was shaped into a framework that generated them. In the ADD project, the CD2GUI extension further accelerated prototyping (lesson 19) by enabling the drafting of the user interface from the class diagram. Due to the acceleration, more prototypes could be created, and it was easier to recognize problems early (lesson 16). Prototyping with the generator helped us in every project and is continuously used in the MaCoCo project.

8.2. Generator development: Implementing a generator

We have learned that implementing a project with a generator in parallel to creating or further refining this generator does not introduce a significant overhead if MDE tooling is available and the developers are familiar with it (lesson 1). In all of our projects, MDE experts were involved, and the development went unhindered from the beginning. The exception was the start of the Ford project when the generator framework had to be separated from the MaCoCo project, as the need for a project-independent tool became clear (lesson 5). It only had to be done once, but had we done it earlier, the effort would have been lower.

In our biggest project, MaCoCo, the functionality became so complex that the existing models could not describe everything. For example, the ever-present validations of the data structure required introducing a new DSL (lesson 2). Such cases did not occur later; updating the tooling was sufficient for the smaller projects. An alternative that we often use is extending generated code with hand-written code (lesson 14). As with any other project, all of our projects have specific requirements that cannot be fulfilled by the MDE tooling alone.

In the brownfield projects (Energy Management, InviDas, ADD), our generated code had to fit the existing infrastructure. The code was not replaceable, so the framework required an update that made it more reusable in the future (lesson 11). Parts being replaced also implied that much of the generated code was no longer needed but was still generated. As this led to slower compilation times, the generator had to be adjusted again (lesson 12). This also becomes an issue when smaller projects use powerful generators. For example, the Ford project did not use all of the functionalities generated by our framework; it also suffered from slow building times.

8.3. Generator integration: Applying MDE to a project

Applying the MDE approach from the project start is overall beneficial in our experience (lesson 3). An exception was the transition point when the tooling was not yet refined and was still project-dependent, as was the case

at the beginning of the Ford project. Aside from using the generator from the beginning, retrofitting also pays off but comes at an initial cost (lesson 4). Every project had new patterns that we had to integrate into the tooling. The upgraded generator effectively replaced the hand-written code, and the upgrades were used in other projects.

We have learned several lessons concerning specific contexts and requirements while working on different projects. The generated solution may be unusable for specific project scenarios, such as building native mobile applications in the Ford project (lesson 7). The experience in the InviDas project showed that the models used for both development and communication must be synchronized (lesson 15). ADD and Energy Management projects had existing data storage to be used instead of the generated ones. The generator had to handle the existing architecture and produce the code that fits these parts (lesson 17). These lessons were not directly applied in every project but provided useful insights for future work.

Some of our DSLs allow the embedding of hand-written code in the models. In the MaCoCo and the Ford project, this was abused to build requirement-specific solutions, which made the models incomprehensible (lesson 8). Such hand-written code integration was minimized in our later projects.

Reusing the various components in different development scenarios, such as research, education, and industry, and implementing them for other use cases not only increased the set of supported features but also improved the stability, usability, and performance of both the generator and the generated application (lesson 20). The projects, developers, and user feedback were the main driving force behind the framework’s evolution.

8.4. Evolution of the MontiGem framework

The development of MontiGem started with the MaCoCo project (Section 3), where we focused on generator development and integration. Some changes always supplemented each transition to a new project; however, it had the most significant impact the first time when a framework was separated from the initial project. Figure 10 highlights the development of the MontiGem architecture over time. On the left, the rather monolithic architecture of MaCoCo is shown. The project includes one of the generators and both RTEs for the frontend and backend. Further, an extended RTE was needed in the Ford project (Section 4), removing corresponding sources from the project and introducing them as modular components. In current implementations, generators (Figure 10, right) DSL and RTEs are developed in their projects, making the framework highly modular. Separating modules into different projects introduced the possibility of providing features from

one use case to other projects. For example, within the Ford project, GUI DSL and GUI generator were extended to support user interfaces with multiple languages, allowing MaCoCo to generate multi-language user interfaces. A disadvantage of this approach is the implementation overhead since we had to avoid introducing breaking changes from one project to another. Adhering to best practices while implementing and maintaining a solid CI/CD environment reduced the required effort for the development team.

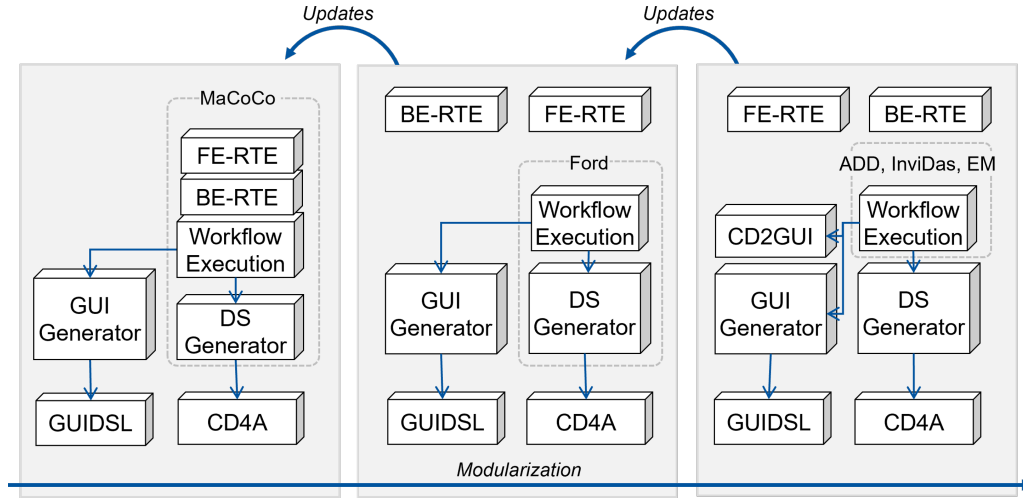


Figure 10: Modularization of the MontiGem framework over time

The tooling was further improved when we introduced the architecture to different industrial cases. We identified inefficient code elements and boosted performance for projects. Optimization for loading large data sets implemented for ADD was also helpful in the MaCoCo project.

Although we have made many improvements to make this approach more generic and flexible to meet the needs of new use cases, we are still tied to the architecture the generators were built to work with. While there are frameworks that address challenges such as cloud-based scalability or creating a native app for mobile applications, we cannot move to these approaches without significant effort and potentially losing the benefits described above.

The model-driven framework has shown to be a powerful and flexible tool to generate applications over a wide variety of use cases. Still, compared to other solutions, MontiGem provides little tooling as all input models are textual and are developed in the same IDE as the rest of the code is implemented in.

8.5. Global takeaways

Applying the MDE approach facilitates stakeholder interaction through rapid prototyping, model-driven development, and communication. Building software using textual models and providing a graphical implementation keeps the system in sync with user requirements. To further increase customer engagement, one can invest in developing tools that help build prototypes more rapidly and frequently, such as additional model-to-model transformers.

The code generator should be project-agnostic, support manual extensions, and enable the exclusion of large specialized modules. These properties are required if the tool is to develop multiple applications in different domains. This also means that a clear understanding of the domains a generator framework will be applied to is crucial for its development, especially at the start when the generator basis is formed. A generator needs to be focused on fulfilling specific tasks.

Applying the MDE approach in different projects drives further refinement and development of the tooling. The process is iterative and is influenced by various factors, such as the application domain, project requirements, and whether a project has an existing code base. When facing new challenges, one has to consider the generator’s current state and extend the tooling if necessary or use a different solution. For example, the project can use hand-written code to satisfy specific requirements. However, it is essential to apply the solution methodically. For example, we embedded hand-written code into the models, but because of that, we faced a different issue. Applying the MDE approach in real-world projects reveals the shortcomings of the tooling and helps to find a methodical solution.

9. Roadmap

Considering the lessons, we present an agenda for how to evolve the generator framework.

Combine lessons learned in a revised tool. We learned 20 lessons while developing the MontiGem framework over several projects. As a result, we are working on a new generator framework version. It will consider our lessons, tap into more recent frameworks, and use a new technology stack. The next version of the framework will feature higher modularity (lesson 11) and reduce unused generated code (lesson 12). The generator is developed with a clear separation of domain-dependent and generic components (lesson 5) while permitting hand-written and model-driven extensions for both (lesson 14).

Provide improved tooling. Visual representations of the models used can be helpful for the end user (lesson 18); therefore, a corresponding transformer is developed, extending the framework with additional views for the user. To support the developer in prototyping an app, an additional transformer is added that provides default user interfaces for any data structure given (lessons 13, 16, 19).

Speed up prototyping by involving Large Language Models. Application development benefits from early prototyping (lesson 13, 16). We currently evaluate the usage of Large Language Models (LLMs) to create specific models for a given DSL. LLMs such as GPT-4 have been established to be powerful tools for processing natural language. Several evaluations have been done on the capabilities of an LLM to produce models for a given DSL (c.f. [50, 51]). An LLM-based toolchain could be included in the development process, providing initial models based on a domain expert’s informal specifications. Using LLMs could reduce the modeling expertise needed by the developer and thus speed up the development process. As of now, only the creation of CD4A models is targeted.

Provide extensions for further DSLs. The benefits of MDE are well established; therefore, we strive to extend the number of DSLs and the variety of models supported by the framework. Currently, structural languages such as class diagrams and GUI models are supported. In the future, process-describing languages such as BPMN (cf. [37, 40]) or sequence diagrams, as well as simple logic defined in OCL, could be supported.

Use revised generator framework in industry and real-world projects. Practical applications drive the advancement of the generator framework (lesson 20). Therefore, further projects will likely use the new version of the framework, pushing development and yielding new lessons to be learned.

10. Related work

While developing several DSLs and generators for several years, we have gathered experiences about common issues, advantages of MDE, and aspects of the web application domain.

The lessons we learned about the benefits of the MDE for user communication are also found in the experiences of other researchers. Verbruggen and Snoeck [52], in their overview of model-driven engineering in practice, highlight its strengths, such as code generation, prototyping, and usage of models for communication with clients and the team. On the other hand, the integration of the model-driven approach often has issues and overheads. The

steep learning curve and debugging on the model level are qualities of MDE the researchers list as problematic. While we have also experienced the positive aspects, the negative points have not been a challenge in our projects, although similar works confirm such research statement [53, 54, 55]. Learning to write models does not take much time, even for untrained students, as long as examples and documentation are provided. Considering the number of examples, an inexperienced worker can almost always find a blueprint for their solution. Debugging on the model level is a nice-to-have feature, but in our experience is not necessary if mapping to the generated code is straightforward.

Regarding the integration of the generator tooling, several works report similar to our experience. In their multiple studies of applying the model-driven approach in practice, Hutchinson, Rouncefield, and Whittle [56, 57, 58] give an insight into common problems and benefits. One of their findings is that building the first project with the MDE is less cost-efficient. We confirm the statement in this work and add that having the right tools and modeling experts, although not wholly specialized in the project, mitigates the problem of building the very first MDE application. Another related point is if MDE is integrated into a project, it is best to do it at its start. We have seen how retrofitting a generator compares to initially building a generator in the MaCoCo project, which aligns with the studies. Our findings mismatch the recommendation of the studies to keep the domain narrow, such as a specific business application, rather than financial systems in general. Our research discovered that the same model-driven framework can be applied to different domains. Nadas et al. [59] bring up the difficulty of fitting sentences written in a natural language into the models. Their work dealt with privacy policies, the same as in our InviDas project, where we observed the issue without being able to address it other than manually.

The lessons we learned about developing our generator framework can be observed in related works. Brambilla and Fraternali [60] report on the WebML and WebRatio technologies very similar to our framework. We see the same lessons from the planning phases of developing DSLs and generators. They mention that one should foresee an eventual tooling extension, which can be reused later for a different domain. Implementing specific use cases should not be modeled but implemented by hand and integrated into a system. Similar lessons are recorded by Davies et al. [61], which they learned in 10 years of development using model-driven technology. They mention managing changes that come with the evolution of models, using a different DSL for different aspects of a generated system, data migration, etc. Our work concentrates on more specific cases, but the ideas align with the paper. In the research of bringing MDE into small companies, Cuadrado et al. [62]

include in their report the importance of having available tools when starting a project, which was the case for all of our projects. They also discovered that the companies moved away from using code generators since the requirements changed since nobody had the knowledge necessary for further tooling development. We have seen a similar shift in the Ford project, where a native Android application was developed from scratch. Extending the generator could have solved the problem, but neither we nor other MDE experts were further involved in the development.

Regardless of the domain, our experiences using MDE align with industrial cases and other practical applications [1, 63, 64, 65, 66]. Models are an excellent tool for communication between project stakeholders, especially if a graphical representation of the models is available. In the hands of experts, the tooling enables the quick development of prototypes. With the project's growth using a model-driven approach adding new features results in generator extension, integration of new DSLs, or implementing the functionality by hand.

11. Conclusion

This paper presents five real-world information systems we developed and continue developing using the model-driven approach. We summarize the most impactful lessons from the experience of our MDE experts team in developing and using a generator framework MontiGem. The tooling emerged from our first and biggest project MaCoCo, and continues to evolve as it is used in current and upcoming research projects. Initially, we greatly benefited from code generation, which had taken over the task of writing boilerplate code. We also experienced drawbacks, such as spending extra time retrofitting generators and modularizing the tool, partly due to our negligence in the planning phases. Later projects showed positive results by enabling fast prototyping and provided valuable insights into the shortcomings of our tool that enabled further improvements to the reusability of the MontiGem. The results not only impact the quality of the tool but also already running projects using the tool.

The lessons we learned and the context we give provides MDE practitioners with an idea of how to develop information systems and tools. In our projects, we confirm the most significant advantages, such as using models as a communication tool and fast, automatic code generation, and experience such hurdles as trying to retrofit a model-driven approach or to fit application-specific parts into a DSL. We also show our unique experiences, such as having no trouble introducing a code generator into a project under certain conditions and deriving a generator framework. Along with other

success stories, we contribute to the knowledge of bringing MDE to the industry. According to our observations applying the model-driven approach has great potential but requires a high level of expertise in the field.

We plan on improving and extending the generator framework further by increasing the number of supported DSLs while reducing the number of mandatory models needed to generate a web application. Additionally, we aim to increase the portion of generated code and add further functionalities to the generated system, such as automated testing and improving the generated artifacts, such as optimizing the generated client-server communication.

References

- [1] T. Stahl, M. Völter, K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*, Wiley, 2006.
- [2] G. L. Casalaro, G. Cattivera, F. Ciccozzi, I. Malavolta, A. Wortmann, P. Pelliccione, Model-driven engineering for mobile robotic systems: a systematic mapping study, *Software and Systems Modeling* 21 (1) (2022) 19–49. doi:10.1007/s10270-021-00908-8.
- [3] A. Bucchiarone, J. Cabot, R. F. Paige, A. Pierantonio, Grand challenges in model-driven engineering: an analysis of the state of the research, *Software and Systems Modeling* 19 (1) (2020) 5–13. doi:10.1007/s10270-019-00773-6.
- [4] F. Gemeinhardt, A. Garmendia, M. Wimmer, Towards Model-Driven Quantum Software Engineering, in: *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, 2021, pp. 13–15. doi:10.1109/Q-SE52541.2021.00010.
- [5] N. Moreno, J. R. Romero, A. Vallecillo, *An Overview Of Model-Driven Web Engineering and the MDA*, Springer London, 2008, pp. 353–382. doi:10.1007/978-1-84628-923-1_12.
- [6] A. Cicchetti, D. Di Ruscio, R. Eramo, F. Maccarrone, A. Pierantonio, becontent: A model-driven platform for designing and maintaining web applications, in: M. Gaedke, M. Grossniklaus, O. Díaz (Eds.), *Web Engineering*, Springer, 2009, pp. 518–522.
- [7] O. Pastor, J. Molina, *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*, Springer, 2010. doi:10.1007/978-3-540-71868-0.

- [8] L. Cretu, F. Dumitriu, *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*, Apple Academic Press, 2014. doi:10.1201/b17480.
- [9] M. González, L. Cernuzzi, O. Pastor, A navigational role-centric model oriented web approach - MoWebA, *International Journal of Web Engineering and Technology* 11 (1) (2016) 29–67. doi:10.1504/IJWET.2016.075963.
- [10] G. Rossi, M. Urbietta, D. Distanto, J. Rivero, S. Firmenich, 25 Years of Model-Driven Web Engineering: What we achieved, What is missing, *CLEI Electronic Journal* 19 (3) (2016) 5–57. doi:10.19153/cleiej.19.3.1.
- [11] K. Schewe, B. Thalheim, *Design and Development of Web Information Systems*, Springer, 2019. doi:10.1007/978-3-662-58824-6.
- [12] M. Urbietta, S. Firmenich, G. Bosetti, P. Maglione, G. Rossi, M. Olivero, MDWA: a model-driven Web augmentation approach—coping with client- and server-side support, *Software and Systems Modeling* 19 (2020) 1541—1566. doi:10.1007/s10270-020-00779-5.
- [13] M. Dalibor, J. Michael, B. Rumpe, S. Varga, A. Wortmann, Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits, in: G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, H. C. Mayr (Eds.), *Conceptual Modeling*, Springer International Publishing, 2020, pp. 377–387.
- [14] S. Alvarado, A. Cortiñas, M. Luaces, O. Pedreira, A. Places, Multilevel modeling of geographic information systems based on international standards, *Software and Systems Modeling* 21 (2021) 623–666. doi:10.1007/s10270-021-00901-1.
- [15] M. Snoeck, C. Verbruggen, J. De Smedt, J. De Weerd, Supporting data-aware processes with MERODE, *Software and Systems Modeling* (March 2023). doi:10.1007/s10270-023-01095-4.
- [16] B. Rumpe, *Agile Modeling with UML: Code Generation, Testing, Refactoring*, Springer International, 2017.
- [17] A. Boronat, Code-first model-driven engineering: On the agile adoption of mde tooling, in: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 874–886. doi:10.1109/ASE.2019.00086.

- [18] K. Lano, S. Kolahdouz-Rahimi, J. Troya, H. Alfraihi, Introduction to the theme section on agile model-driven engineering, *Software and Systems Modeling* 21 (4) (2022) 1465–1467. doi:10.1007/s10270-022-01016-x.
- [19] S. Mirachi, V. Da Costa Guerra, A. M. Da Cunha, L. A. V. Dias, E. Villani, Applying agile methods to aircraft embedded software: an experimental analysis, *Software: Practice and Experience* 47 (11) (2017) 1465–1484. doi:10.1002/spe.2477.
- [20] J. G. Süß, S. Swift, E. Escott, Using devops toolchains in agile model-driven engineering, *Software and Systems Modeling* 21 (4) (2022) 1495–1510. doi:10.1007/s10270-022-01003-2.
- [21] M. Snoeck, Y. Wautelet, Agile merode: a model-driven software engineering method for user-centric and value-based development, *Software and Systems Modeling* 21 (4) (2022) 1469–1494. doi:10.1007/s10270-022-01015-y.
- [22] K. Adam, J. Michael, L. Netz, B. Rumpe, S. Varga, Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project, in: 40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA’19), Vol. P-304 of LNI, Gesellschaft für Informatik e.V., 2020, pp. 59–66.
- [23] K. Adam, L. Netz, S. Varga, J. Michael, B. Rumpe, P. Heuser, P. Letmathe, Model-Based Generation of Enterprise Information Systems, in: M. Fellmann, K. Sandkuhl (Eds.), *Enterprise Modeling and Information Systems Architectures (EMISA’18)*, Vol. 2097 of CEUR Workshop Proceedings, CEUR-WS.org, 2018, pp. 75–79.
- [24] A. Gerasimov, P. Letmathe, J. Michael, L. Netz, B. Rumpe, Modeling Financial, Project and Staff Management: A Case Report from the MaCoCo Project, *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling* (2023).
- [25] A. Iung, J. Carbonell, L. Marchezan, E. Rodrigues, M. Bernardino, F. P. Basso, B. Medeiros, Systematic mapping study on domain-specific language development tools, *Empirical Software Engineering* 25 (5) (2020) 4205–4249.
- [26] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*, dslbook.org, 2013.

- [27] D. Karagiannis, H. C. Mayr, J. Mylopoulos (Eds.), Domain-specific conceptual modeling: Concepts, methods and tools, Springer, 2016.
- [28] Object Management Group, OMG Unified Modeling Language (OMG UML) (2017).
URL <https://www.omg.org/spec/UML/2.5.1/PDF>
- [29] B. Rumpe, Modeling with UML: Language, Concepts, Methods, Springer International, 2016.
URL <https://mbse.se-rwth.de/>
- [30] Chair of Software Engineering, Class Diagram For Analysis (2023).
URL <https://github.com/MontiCore/cd4analysis>
- [31] A. Gerasimov, J. Michael, L. Netz, B. Rumpe, S. Varga, Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems, in: B. Anderson, J. Thatcher, R. Meservy (Eds.), 25th Americas Conf. on Information Systems (AMCIS 2020), AIS, 2020, pp. 1–10.
- [32] K. Hölldobler, O. Kautz, B. Rumpe, MontiCore Language Workbench and Library Handbook: Edition 2021, Aachener Informatik-Berichte, Software Engineering, Band 48, Shaker Verlag, 2021.
- [33] H. Krahn, B. Rumpe, S. Völkel, MontiCore: a Framework for Compositional Development of Domain Specific Languages, International Journal on Software Tools for Technology Transfer (STTT) 12 (5) (2010) 353–372.
- [34] K. Hölldobler, J. Michael, J. O. Ringert, B. Rumpe, A. Wortmann, Innovations in Model-based Software and Systems Engineering, Journal of Object Technology (JOT) 18 (1) (2019) 1–60. doi:10.5381/jot.2019.18.1.r1.
- [35] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, C. Schulze, Systematic Synthesis of Delta Modeling Languages, Journal on Software Tools for Technology Transfer (STTT) 17 (5) (2015) 601–626.
- [36] T. Greifenberg, M. Look, S. Roidl, B. Rumpe, Engineering Tagging Languages for DSLs, in: Conference on Model Driven Engineering Languages and Systems (MODELS’15), ACM/IEEE, 2015, pp. 34–43.

- [37] I. Drave, J. Michael, E. Müller, B. Rumpe, S. Varga, Model-Driven Engineering of Process-Aware Information Systems, Springer Nature Computer Science Journal 3 (November 2022).
- [38] M. Dalibor, M. Heithoff, J. Michael, L. Netz, J. Pfeiffer, B. Rumpe, S. Varga, A. Wortmann, Generating Customized Low-Code Development Platforms for Digital Twins, Journal of Computer Languages (COLA) 70 (June 2022).
- [39] J. Michael, I. Nachmann, L. Netz, B. Rumpe, S. Stüber, Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines, in: Modellierung 2022, Gesellschaft für Informatik, 2022, pp. 33–48.
- [40] D. Bano, J. Michael, B. Rumpe, S. Varga, M. Weske, Process-Aware Digital Twin Cockpit Synthesis from Event Logs, Journal of Computer Languages (COLA) 70 (June 2022). doi:10.1016/j.cola.2022.101121.
- [41] A. Butting, J. C. Kirchhof, A. Kleiss, J. Michael, R. Orlov, B. Rumpe, Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices, in: 21th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22), ACM, 2022, pp. 108–121.
- [42] J. Michael, L. Netz, B. Rumpe, S. Varga, Towards Privacy-Preserving IoT Systems Using Model Driven Engineering, in: Proc. of MODELS 2019. Workshop MDE4IoT, CEUR Workshop Proceedings, 2019, pp. 595–614.
- [43] J. Michael, A Vision Towards Generated Assistive Systems for Supporting Human Interactions in Production, in: Modellierung 2022 Satellite Events, Gesellschaft für Informatik e.V., 2022, pp. 150–153.
- [44] J. Michael, B. Rumpe, L. T. Zimmermann, Goal Modeling and MDSE for Behavior Assistance, in: Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C), ACM/IEEE, 2021, pp. 370–379.
- [45] A. Gerasimov, P. Heuser, H. Ketteniß, P. Letmathe, J. Michael, L. Netz, B. Rumpe, S. Varga, Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend, in: Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers, CEUR Workshop Proceedings, 2020, pp. 22–30.

- [46] I. Drave, A. Gerasimov, J. Michael, L. Netz, B. Rumpe, S. Varga, A Methodology for Retrofitting Generative Aspects in Existing Applications, *Journal of Object Technology (JOT)* 20 (2021) 1–24. doi: <https://doi.org/10.5381/jot.2021.20.2.a7>.
- [47] A. Butting, N. Conradie, J. Croll, M. Fehler, C. Gruber, D. Herrmann, A. Mertens, J. Michael, V. Nitsch, S. Nagel, S. Pütz, B. Rumpe, E. Schauermann, J. Schöning, C. Stellmacher, S. Theis, Souveräne digitalrechtliche Entscheidungsfindung hinsichtlich der Datenpreisgabe bei der Nutzung von Wearables, in: *Selbstbestimmung, Privatheit und Datenschutz: Gestaltungsoptionen für einen europäischen Weg*, Springer Fachmedien Wiesbaden, 2022, pp. 489–508.
- [48] F. Drux, N. Jansen, B. Rumpe, A Catalog of Design Patterns for Compositional Language Engineering, *Journal of Object Technology (JOT)* 21 (4) (2022) 4:1–13.
- [49] M. Garcia, Bidirectional synchronization of multiple views of software models., *DSML* 8 (2008) 7–19.
- [50] J. Cámara, J. Troya, L. Burgueño, A. Vallecillo, On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml, *Software and Systems Modeling* (2023) 1–13.
- [51] M. B. Chaaben, L. Burgueño, H. Sahraoui, Towards using few-shot prompt learning for automating model completion, in: *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, IEEE, 2023, pp. 7–12.
- [52] C. Verbruggen, M. Snoeck, Practitioners’ experiences with model-driven engineering: a meta-review, *Software and Systems Modeling* 22 (1) (2023) 111–129. doi:10.1007/s10270-022-01020-1.
- [53] V. Kulkarni, Model Driven Software Development, in: P. Van Gorp, T. Ritter, L. M. Rose (Eds.), *Modelling Foundations and Applications*, Springer Berlin Heidelberg, 2013, pp. 220–235.
- [54] H. Alfraihi, K. Lano, Practical Aspects of the Integration of Agile Development and Model-driven Development: An Exploratory, Vol. 2019, 2017, pp. 399–404.
- [55] S. Karg, A. Raschke, M. Tichy, G. Liebel, Model-Driven Software Engineering in the OpenETCS Project: Project Experiences and Lessons Learned, in: *ACM/IEEE 19th International Conference on Model*

- Driven Engineering Languages and Systems, MODELS '16, ACM, 2016, p. 238–248. doi:10.1145/2976767.2976811.
- [56] J. Hutchinson, M. Rouncefield, J. Whittle, Model-Driven Engineering Practices in Industry, in: 33rd International Conference on Software Engineering, ICSE '11, ACM, 2011, p. 633–642. doi:10.1145/1985793.1985882.
- [57] J. Whittle, J. Hutchinson, M. Rouncefield, The State of Practice in Model-Driven Engineering, IEEE Software 31 (3) (2014) 79–85. doi:10.1109/MS.2013.65.
- [58] J. Hutchinson, J. Whittle, M. Rouncefield, Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure, Science of Computer Programming 89 (2014) 144–161, special issue on Success Stories in Model Driven Engineering. doi:https://doi.org/10.1016/j.scico.2013.03.017.
- [59] A. Nadas, T. Levendovszky, E. K. Jackson, I. Madari, J. Sztipanovits, A model-integrated authoring environment for privacy policies, Science of Computer Programming 89 (2014) 105–125, special issue on Success Stories in Model Driven Engineering. doi:https://doi.org/10.1016/j.scico.2013.05.004.
- [60] M. Brambilla, P. Fraternali, Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience, Science of Computer Programming 89 (2014) 71–87, special issue on Success Stories in Model Driven Engineering. doi:https://doi.org/10.1016/j.scico.2013.03.010.
- [61] J. Davies, J. Gibbons, J. Welch, E. Crichton, Model-driven engineering of information systems: 10 years and 1000 versions, Science of Computer Programming 89 (2014) 88–104, special issue on Success Stories in Model Driven Engineering. doi:https://doi.org/10.1016/j.scico.2013.02.002.
- [62] J. S. Cuadrado, J. L. Cánovas Izquierdo, J. G. Molina, Applying model-driven engineering in small software enterprises, Science of Computer Programming 89 (2014) 176–198, special issue on Success Stories in Model Driven Engineering. doi:https://doi.org/10.1016/j.scico.2013.04.007.
- [63] T. Nepomuceno, T. Carneiro, P. H. Maia, M. Adnan, T. Nepomuceno, A. Martin, AutoIoT: A Framework Based on User-Driven MDE

- for Generating IoT Applications, in: 35th Annual ACM Symposium on Applied Computing, SAC '20, ACM, 2020, p. 719–728. doi: 10.1145/3341105.3373873.
- [64] B. Lelandais, M.-P. Oudot, B. Combemale, Applying Model-Driven Engineering to High-Performance Computing: Experience Report, Lessons Learned, and Remaining Challenges, *Journal of Computer Languages* 55 (10 2019). doi:10.1016/j.col.2019.100919.
- [65] A. Ferrari, A. Fantechi, S. Gnesi, Lessons Learnt from the Adoption of Formal Model-Based Development, in: A. E. Goodloe, S. Person (Eds.), *NASA Formal Methods*, Springer Berlin Heidelberg, 2012, pp. 24–38.
- [66] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, W. Gilani, Mde adoption in industry: Challenges and success criteria, in: *Models in Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 54–59.