A Novel Approach to Integration of Manual Changes in Generated Code: SeamlessMDD

Bojana Dragaš[1], Nenad Todorović[1], Tijana Rajačić[2], Gordana Milosavljević[1] and Željko Vuković[1,*]

[1]*University of Novi Sad, Faculty of Technical Sciences, Serbia, Trg Dositeja Obradovića 6, Novi Sad, Serbia*
[2]*Schneider Electric, Industrijska 3G, Novi Sad, Serbia*
E-mail: bojana.zoranovic@uns.ac.rs; nenadtod@uns.ac.rs; tijana.rajacic@se.com; grist@uns.ac.rs; zeljkov@uns.ac.rs
[*]*Corresponding Author*

Received 02 November 2024; Accepted 26 March 2025

Abstract

Model-driven development (MDD) significantly improves web development by generating source code from models at higher abstraction levels, which enhances productivity and enables shorter, less expensive development cycles. However, not all aspects of a web application can be captured in a model, necessitating manual changes to the generated code. To prevent the loss of manual changes due to subsequent code generation, various strategies are recommended in the literature: keeping handwritten and generated code in separate files, utilizing protected regions, or employing a version control system (VCS) to merge handwritten and generated code. Unfortunately, these approaches can introduce complexity into web application architecture and impose an additional burden on developers.

This paper presents SeamlessMDD, our novel open-source framework for seamless integration that allows us to maintain handwritten and generated code intertwined without the need to adjust the web application architecture

Journal of Web Engineering, Vol. 24_4, 499–528.
doi: 10.13052/jwe1540-9589.2442
© 2025 River Publishers

or established workflows. The framework provides the following features: (1) incremental and iterative transformations derived from model versions comparison ensuring that only code for affected model elements is generated or modified; (2) integration of generated and handwritten code using API-based code generators (ABG) that operate on the syntax trees of target programming languages; (3) case-specific support for change propagation and conflict resolution, as opposed to VCS-based systems that operate on a single line. The proposed features were tested in practice within a complex industrial MDD tool.

This paper is an extended version of a paper presented at the 24th International Conference on Web Engineering, held in Tampere, Finland.

Keywords: Model-driven developement, code generation, handwritten code integration.

1 Introduction

Model-driven development (MDD) is a software development methodology focused on models as primary artifacts [6, 25]. It is claimed to achieve higher developer productivity, shorter and less expensive development cycles, and smooth portability to other hardware and infrastructure. MDD significantly improves web development by generating source code from models at higher abstraction levels, which notably enhances productivity and enables shorter and less expensive development cycles [4, 8].

However, due to the differences in abstraction levels between source code and models, not all aspects of a web application can usually be captured in a model. As a result, manual changes to the generated code are often necessary. Model-driven tools employ various strategies to preserve manually written code. Manual changes are often kept in separate files, and language-dependent mechanisms like inheritance, partial classes, or aspect-oriented programming can be used to integrate generated and manually written parts of the code [12]. Unfortunately, this introduces additional complexity in web application architecture, making it unacceptable or impractical for certain types of code artifacts, such as HTML or XAML. The use of protected regions where manual changes can be inserted complicates the development of code generation tools, reduces the readability of the code, and can lead to the loss of manual changes if not used properly. Utilizing a version control system (VCS) to integrate manual and generated code, as discussed in [5], can place a considerable burden on developers who need to resolve conflicts

and approve merging in cases where numerous code artifacts are affected by a single model element change, which is common in web development.

Several causes hinder the adoption of MDD in the industry. Using model-driven tools often interferes with established ways of thinking, activities, and roles in the development process [11]. Large-scale enterprise solutions face additional challenges due to increased complexity, which significantly affects development [29]. Maintenance becomes difficult when the solution reaches the maturity phase, and major refactoring is often avoided. Although the introduction of the MDD methodology could leverage some, if not all, of these issues, stakeholders are hesitant to implement fundamental changes in the development workflow and established architecture, fearing that such changes could disrupt components that are already tested and in use. As de Lange et al. stated in [8, 9], many MDD approaches enforce top-down rather than iterative implementation, making them difficult to align with contemporary agile development. Selic in [24] emphasizes that a prudent and practical way to introduce a new MDD project into the existing large-scale environment is to integrate its development workflow with the established system's processes and environment. Drastic changes could disrupt existing functionalities, alter the usual workflow, and induce various social issues regarding acceptance.

Our work aims to provide a gradual and effortless transition from the traditional development of web applications to MDD. This paper presents SeamlessMDD[1] – our open-source framework for seamless integration that allows us to maintain handwritten and generated code intertwined without the need to adjust the web application architecture or the established way of working. We propose a novel workflow that is based on the following:

(1) Incremental and iterative transformations derived from model versions comparison so that only code for affected model elements is generated or modified.
(2) Integration of generated and handwritten code using API-based code generators that operate on syntax trees of target programming languages.
(3) Case-specific support for change propagation and conflict resolution (as opposed to VCS-based systems that operate on a single line).

Apart from the Introduction, this paper is organized as follows. In Section 2, we summarize the theoretical foundations of the proposed research along with a survey of the available literature. Section 3 introduces our SeamlessMDD framework, detailing a defined set of requirements, an established

[1] https://github.com/BojanaZ/SeamlessMDD

solution architecture, and a seamless generation workflow. An illustration of the SeamlessMDD application for automating the further development of an existing web application is presented in Section 4. Verification of the approach is described in Section 5. Finally, conclusions and future work are presented in Section 6.

2 Background and Related Work

One of the easiest ways to implement model-to-code transformation is to transform the entire source model to the target code using batch transformations based on rendering code templates. The modeler (a person who creates or maintains a model) focuses on describing the system with the appropriate modeling language. The transformation specialist develops the infrastructure for the transformation, including templates and code generators. The templates contain lower-level details and contextual information that are not inferable from the model but are required for the functioning solution [7]. The transformation utilizing templates often provides a substantial portion of the source code for the target application.

Batch transformations are straightforward because they do not require complex algorithms or computations. However, their main disadvantage is the lack of efficiency in large-scale systems; even a minor change in the model requires re-executing the entire transformation [15]. Furthermore, for features not supported by the modeling language, which must be developed manually, a carefully planned strategy is essential for preserving handwritten code during subsequent code generation.

A common approach to prevent loss of manually introduced features in the next code generation cycle is to separate the generated and handwritten code into different files and folders. Greifenberg et al., in [12], examined and contrasted the available techniques for maintaining this separation. Various language-specific techniques, such as inheritance, dependency injection, and partial classes, can be used to achieve this. As a result, the application is often divided into two parts – generated and manually written – leading to changes in the established architecture. Furthermore, the traditional MDD workflow expects the generated code to be altered only by a code generator. Code generated under this assumption often fails to adhere to coding conventions and guidelines, can be counterintuitive, and, in some cases, may even be considered incomprehensible for manual analysis [23].

Tools like Acceleo [1] maintain a clear distinction between handwritten and generated code by utilizing language-agnostic protected areas. They

employ special tags to indicate where handwritten code should be inserted after generation, or they also separate code into different files or folders.

Commercial tools such as WebRatio [3] and Mendix [2] allow for the introduction of tool-specific, non-modeled features through model annotation. However, this approach can be more complex than directly modifying the target code, as changes must be made through the modeling tool. Consequently, each developer who needs to make code changes must be trained to use the modeling tool and have access to it. This requirement may impact efficiency and could lead to licensing and practical issues when multiple entities own or maintain parts of the solution.

Falzone and Bernaschina [10] and Bernaschina et al. [5] separate handwritten and generated code using branches of a version control system (VCS). The code generator behaves as a *virtual developer* – the generated code is committed to the VCS branch just as a human developer would produce it, and later merged into the codebase. The benefits of this approach include maintaining the same workflow as traditional code-centric development (since the developer perceives the code generator as *just another developer*) and automating code merging and conflict resolution using a modern VCS. However, VCS performs line-based merging, a variant of textual merging, which regards a line of code as a unit of change. Although efficient and scalable, line-based merging cannot detect two modifications on the same line. Automatic conflict resolution could also introduce errors in the source code (when the merging result is not syntactically correct) and register non-critical conflicts, such as newline insertion or indentation changes. Weaknesses of this approach may become more evident as the number of artifacts increases. This issue is especially prominent in web development since a change to one model element can affect many artifacts in both the front-end and back-end.

Our approach differs in that it does not require handwritten code to be separated. The developer modifies the code, whether it was manually written or generated, thus maintaining their usual workflow. Additionally, we aim to achieve syntactical merging. However, a downside is the need to develop off-the-shelf support for merging and conflict resolution.

To achieve efficient transformation, we explored different techniques and approaches to achieve incremental, efficient, and scalable transformation.

Incremental transformation allows only the relevant parts of the source code to be changed by the code generator while keeping the rest intact [14, 17]. It detects applied model operations, transforms only the affected elements of the source model, and precisely alters only the relevant sections of the target code. Previously existing target code is updated, and in most cases,

all manually introduced changes are preserved. While incremental transformation requires more initial effort to compare models and identify performed modeling operations, it often proves to be more computationally efficient and time-saving in the long run compared to batch transformation [13].

In the work of Ogunyomi in [21] and Ogunyomi et al. [20], the authors explored techniques to increase the efficacy of model-to-text transformation, focusing on achieving scalable transformation. They noted that, ideally, the transformation should be selective; that is, only changed model elements and the affected parts of the transformation should be executed. This approach is known as source-preserving incrementality [7]. Additionally, Ogunyomi states that the re-execution of a transformation should be limited to the affected parts of the input model, and the scale of the propagated change should be proportional to the impact of the change rather than the size of the model. Although our focus is not solely on scalability, we based our work on these ideas to reduce the scale of the transformation and, as a result, diminish the chance of emerging conflicts.

The field of Delta-oriented programming (DOP), explored in [19, 22], and [18], offers an interesting perspective on the challenge of capturing model changes and propagating them to software product lines (SPL). DOP focuses on creating software products by transforming the initial code base (base artifact) through the incorporation of delta modules as units of transformation. These delta modules modify existing solutions by inserting, changing, or removing parts of the code. Special attention is paid to validity and minimizing conflicts by defining rules that determine whether a delta module should be included in the product. Our work is inspired by this field, although it aims for a more general application.

3 SeamlessMDD Framework

In this section, we present our approach that supports the integration of the handwritten and generated code to provide a smoother adoption of the MDD to the industry settings by conforming to the already established workflows and system architecture. The SeamlessMDD framework, a prototype implementation of this approach, is developed in Python3. To allow domain modeling in a standardized manner, we integrated PyECore[2] into our solution. PyECore is implementation of ECore [26] in Python. It provides an API to handle meta-models and models, which serve as input for code

[2]https://pyecore.readthedocs.io/en/latest/

generators. To detect a minimal changeset between models, we used the DeepDiff[3] module. Code generation was handled with the Jinja[4] templating engine. API-based generators require a parser for each language/file type.

The rest of the section is organized as follows. In Section 3.1, we define a set of requirements for a tool that should support seamless integration. Section 3.2 presents the architecture of the SeamlessMDD framework. Section 3.3 presents the workflow to support integration of handwritten and generated code.

3.1 Requirements

In our view, the following prerequisites are needed to transition smoothly to the MDD in industrial settings.

- **R1. Generated code must be seamlessly integrated with the handwritten code.** It must follow the same coding conventions as developers do, as if the code generator was another team member. This enhances code readability and reduces anxiety levels of team members unfamiliar with the MDD.
- **R2. Simultaneously introducing model changes and manually modifying the existing code base must be possible.** Falzone et al. in [10] referred to it as model and text co-evolution. Manual changes to the generated code must be preserved after all subsequent code generation cycles.
- **R3. Manual changes have precedence over automatic ones.** Any manual modification the developer introduces should be preserved, whether submitted to previously handwritten or generated code.
- **R4. The developer should be able to introduce multiple model changes with the possibility of reverting them effortlessly.** Usually, multiple model changes precede transformation. The framework should support a workflow where the developer makes a few model changes, tests their impact on the target code, introduces a few more changes, reverts others, tests potential transformation results, and starts the generation process. This requirement helps developers become familiar with the MDD methodology without fear they will break the existing functionalities.

[3] https://zepworks.com/deepdiff/current/
[4] https://jinja.palletsprojects.com/en/3.1.x/

- **R5. Generated code must not violate existing functionalities.** At a minimum, the generated code should be syntactically correct.
- **R6. It should be possible to model only a part of the system, rather than having to model the entire system, while the rest of the solution can continue to be developed or maintained manually.** This way, efforts can be broken down to a more achievable scale, affecting only certain subsystems and not the entire solution, enabling gradual adoption of MDD.
- **R7. Case-specific conflict resolution support.** Version control systems (VCSs) offer line-based conflict resolution support, which can be insufficient for large-scale enterprise solutions. The conflict display should be case-specific, based on the code segment corresponding to the model element change. Our framework should offer developers guided support for conflict resolution by displaying affected code fragments, explaining the problem, and offering semantically appropriate automated resolutions.
- **R8. Reducing the scale of the transformation.** It is essential to reduce the scale of the transformation to minimize the need for merging and the possibility of conflicts occurring. The framework should transform only the parts of the model which have changed, and affect only the relevant file artifacts. In this way source incrementality and minimality are preserved. Ogunyomi in [21] states that efficient transformation requires that the scale of the propagated change be proportional to the scale of the model change, not the bare size of input models.
- **R9. Well-kept code.** Unnecessary or irrelevant artifacts (corresponding to the deleted model elements) should be removed. Before executing such removal, the system must validate that the artifact is unused in the rest of the solution.

3.2 The SeamlessMDD Architecture

The SeamlessMDD framework comprises Model specification and Code Generation modules (Figure 1). The model specification module is dedicated to the production and evolution of the software system model. It requires a meta-model to conform to, upon which a model is created and validated. The aim is to provide the modeler with a graphical or form-based modeling environment. In its current state, the creation of a model is available by programmatic instancing of meta-classes. All model changes are validated against the meta-model, making the representation always consistent.

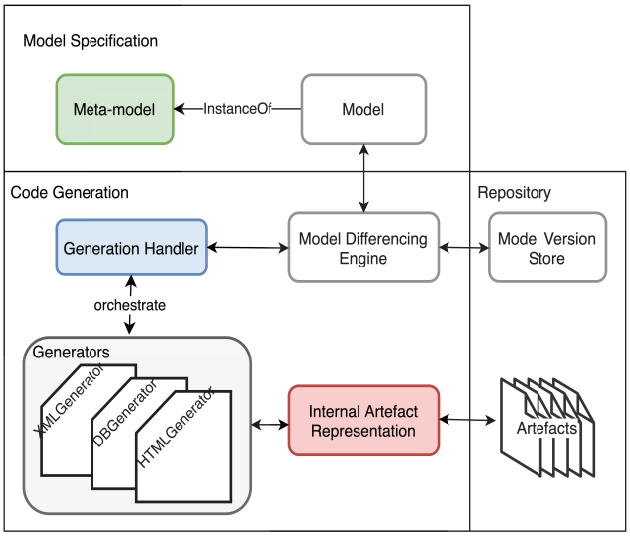


Figure 1 An overview of the SeamlessMDD architecture.

Model Version Store holds the model change history. With each generation cycle, the current model is stored as a new version, allowing easier access and providing a basis for later minimal change-set inference.

The Code Generation Module (Figure 2) supports the model-to-code transformation. The Generator Handler obtains the data about updated model elements necessary for the code generation, orchestrates the generation process, and acquires the output. Firstly, the Generation Handler obtains the previous model version from the last generation cycle and requests the Model Differencing Engine to compare it with the current one to produce a minimal set of changes (deltas). Deltas are used as input to the various code generators. Each Code Generator adds, updates, and removes code for a single type of artifact. Artifacts represent various formats of textual files, such as XML, C# source code, HTML, database scripts, configuration files, etc. In certain cases, there may be multiple generators for a single target language.

In most cases, code generators utilize parsers to obtain concrete syntax tree from the artifact. This approach helps in introducing syntactically correct alterations.

3.2.1 Generation approaches

To achieve efficient model-to-code transformation, code generators select one of various approaches, which we will illustrate in the following section.

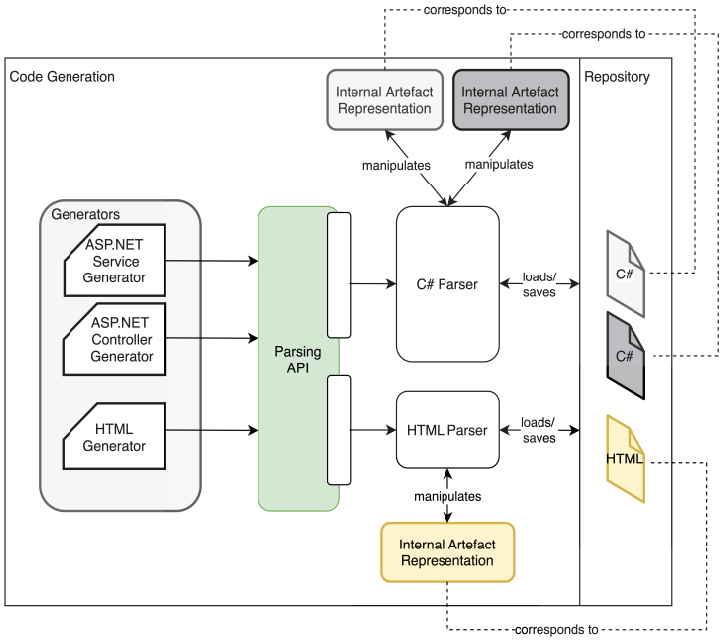


Figure 2 An overview of the approach to code generation.

- **Batch generation based on templates.** The simplest and most desirable option is to generate the entire content solely by evaluating a given template with existing model elements. This scenario is typical for the first generation cycle when the artifact is being newly created. Additionally, software projects often require files, such as configuration files, that must be present in the codebase but are not derived from the model. Consequently, subsequent generation cycles would not affect these files, making them ideal candidates for batch generation.
- **Separation of handwritten code.** If the target language supports a mechanism to separate handwritten and generated code without affecting the established architecture, we would choose that approach. This scenario is favorable since it does not require additional merging and reduces the need for conflict resolution. One example is C# generators, as C# does support partial class creation. In successive generation cycles, the partial classes containing generated code can simply be rewritten. However, it should be noted that even with this approach, human error cannot be completely eliminated. Therefore, it is advisable for a tool implementing this approach to include at least a simple check

to determine if files containing partial classes have been modified since the last generation cycle.

- **API-based generation with segment generation from a template.** In this scenario, the artifact is available, and it is necessary to insert a syntax tree segment. The segment is obtained by template evaluation and parsed to become a syntax tree segment that can be merged with the existing, already parsed artifact's syntax tree. Navigation and syntax tree merging are achieved using API-based generators.
- **API-based generation.** In the most complex cases, it is necessary to manipulate syntax tree elements directly through ParsingAPI. This is the least desirable scenario due to the need to apply fine-grained syntax tree transformations, which can become complicated. In addition, it increases the possibility of later conflicts and merging issues.

To summarize, code generators use templates to generate code fragments corresponding to model elements. If the artifact does not exist, the code generator employs one or more templates to produce it in a batch transformation. Otherwise, the artifact's content is parsed and transformed into a syntax tree with additional context, which we refer to as Internal Artifact Representation (IAR). Generators, provided with a set of model deltas and generated code fragments, propagate appropriate changes to the IAR by creating, removing, or altering the nodes of the syntax tree.

3.3 The Seamless Workflow

Figure 3 shows the steps of the seamless workflow. This subsection will provide more details regarding the SeamlessMDD framework functioning in

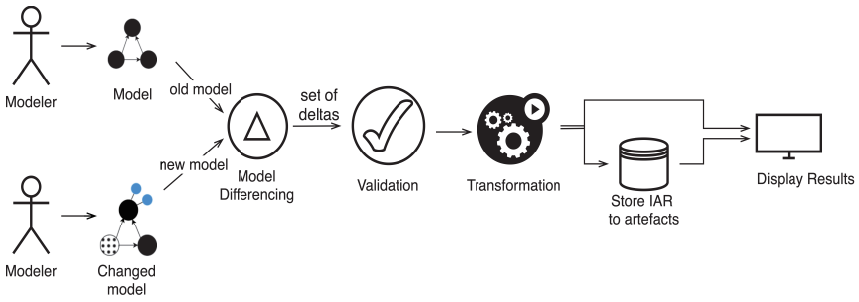


Figure 3 Seamless workflow from the developer's perspective.

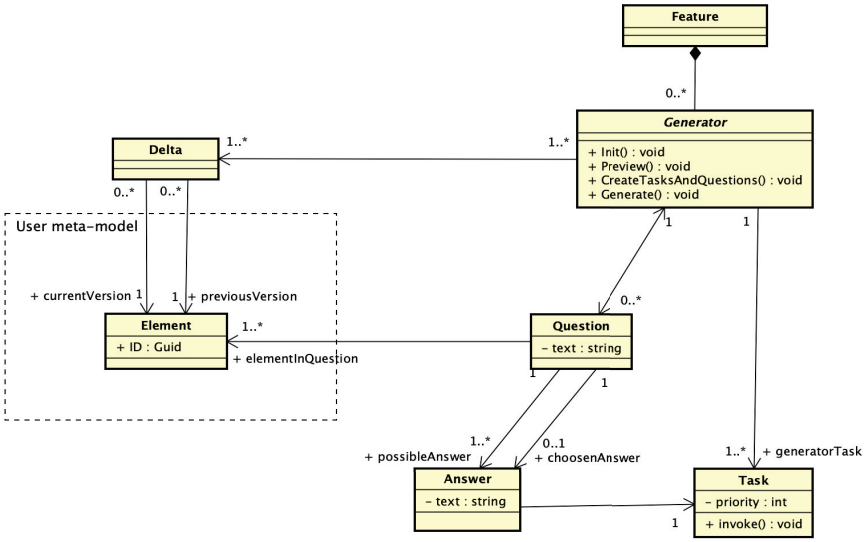


Figure 4 Classes participating in the code generation process – a simplified UML class diagram.

each step. A simplified class diagram upon which the workflow is based is given in Figure 4.

Model change. The modeler starts the development process by creating a model. The created model must be correct, i.e., according to the underlying meta-model. The modeler is encouraged to experiment, knowing that every model change is easily reversible. When they become satisfied with the created model, they can choose to see what impact their changes would have on the target code (”Preview”) or to initiate code generation (”Generate”). ”Generate” and ”Preview” modes share a significant number of workflow steps, as will be explained in the following.

Model differencing. Model differencing aims to compare two model versions to detect introduced changes and model elements affected by those changes. Most generators do not require entire model comparison since they operate on a specific part. For example, the database schema generator uses only database-related model elements such as columns and tables.

The Generator Handler sends comparison requests for selected model elements to the Model Differencing Engine. The result is instantly delivered if the Model Differencing Engine has already calculated the needed difference information for the sake of some previously invoked code generator. Otherwise, the model differencing engine performs model element comparison

resulting in a set of *Delta*s, each comprising *old model element*, *new model element*, and *list of changed attributes*. This is used to detect which operation from the predefined set occurred by the following simple algorithm:

```
if [old_model_element is empty ∧
new_model_element is not empty] create()
else if [old_model_element is not empty ∧
new_model_element is empty] delete()
else modify()
```

Validation. The validation phase aims to detect inconsistencies between the current source model and target artifacts. It plays a crucial role in the framework's usability by significantly reducing the probability of conflicts occurring by considering the syntax and semantics of possible issues. The syntax validation relies on the API-based generators that interact with the IAR established on the artifact's syntax tree. Direct manipulation of the target code syntax tree guarantees the output is syntactically correct. Each generator performs independent validation, meaning the generation process could finish valid for some while with errors for others.

Regarding semantics, the validation process checks if the target code is compatible with the model, i.e., previously generated content was not significantly manually changed (more details can be found in [27]). Developers can manually change the code base, such as deleting certain parts of the code, modifying some code corresponding to the model element attributes, and inserting new attributes.

Upon conflict detection, each code generator explains the emerging issue (*Question*) and proposes a semantically appropriate set of possible resolutions (*Answers*). Each *Answer* provides a textual description of the offered solution and a sample action (*Task*) that should be performed if the *Answer* is to be chosen. The question-and-answer mechanism gives the developers better insight into the nature of the conflict and mitigates case-specific resolution. Numerous questions are the same for all code generators and are reused from the parent generator class. New questions can be created to cover some cases specific to the particular artifact.

Examples of conflicts requiring resolution using questions and answers are the following. The code must be updated due to a change in the modeling element, but the corresponding file artifact (or a code fragment) does not exist. The options are to recreate a file (or code fragment) and then perform the change or to skip the code generation. Or a code fragment corresponding to a newly added model element already exists in the source code but in

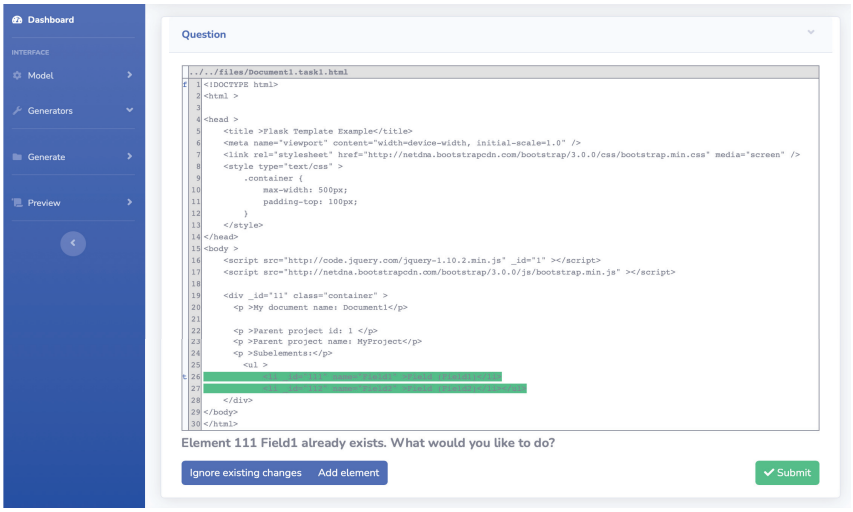


Figure 5 Question with answers.

a slightly different form. The options are to add a new code, replace the existing one with a new one, leave the code as it is, or perform their semantic merge by collecting appropriate nodes from both. Figure 5 displays a preview window with a sample Question and two available Answers. By choosing the answer, the effect of the action is shown in the preview, enabling developers to experiment with different possibilities.

Along with these responsibilities, the validation phase examines the available context, obtaining information for the next phase, such as the path to the correct template, XML path to the parent node, etc.

Transformation. After completing validation, the transformation phase is rather straightforward. All Tasks obtained from the validation phase are added to *TaskCollection* and executed one at a time upon internal artefact representation. The standard operations (insertion, modification, and deletion of modeling elements) are performed as follows.

Insertion: The generator responsible for the Task accesses the template (determined in the validation phase) and evaluates it, obtaining code segment, then converts it to a set of target syntax nodes. Usually, nodes form a simple tree structure. The top-level segment node is inserted into the already loaded syntax tree of Internal Artefact Representation on a pre-calculated position.

Modification: The generator accesses the predefined part of the loaded syntax tree of Internal Artefact Representation and performs modification.

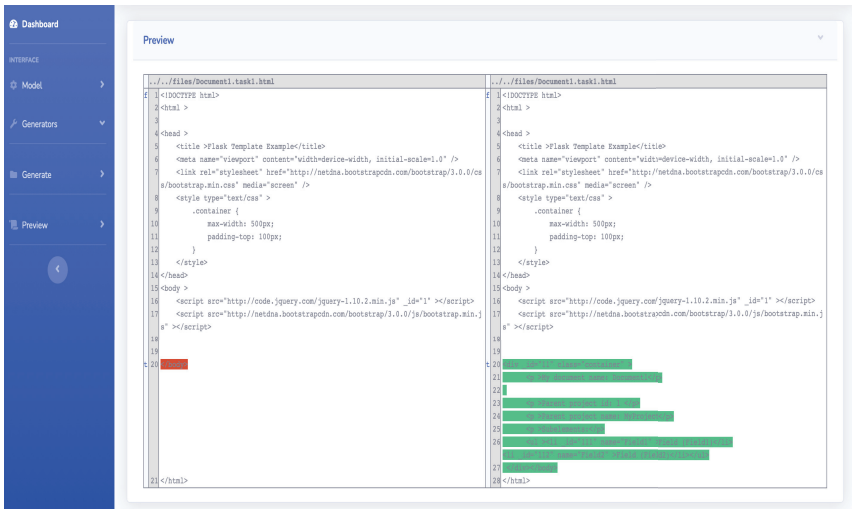


Figure 6 Code preview example for model element addition.

Deletion: The generator accesses the node in the predefined part of the loaded syntax tree of Internal Artefact Representation and deletes it.

All changes are propagated to internal artefact representation at the end of this phase. Currently, the task orchestration mechanism is rather simple since it bases the execution order on the priority programmatically assigned to each created task.

Preview or generate? SeamlessMDD framework operates in two modes: *Preview* and *Generate*. *Preview* mode allows displaying the transformation results without affecting the codebase. The central focus is to display changes that would be introduced if the generation process has been initiated. This allows the development team to detect possible modeling or coding mistakes and alter them.

Generate mode directly propagates the transformation results to the artifacts, making the transformation irreversible and permanent. The only way to revert the changes in the model and code in this phase is by using the VCS.

Display results. The resulting display contains a list of affected target files. When one file from the list is chosen, old and new file versions are shown with differences clearly marked (diff). Figures 6 and 7 show two examples of result displays one when the new model element is inserted and the other when a model element gets deleted. If inconsistencies are encountered, the developer can resolve them through the question-and-answer mechanism

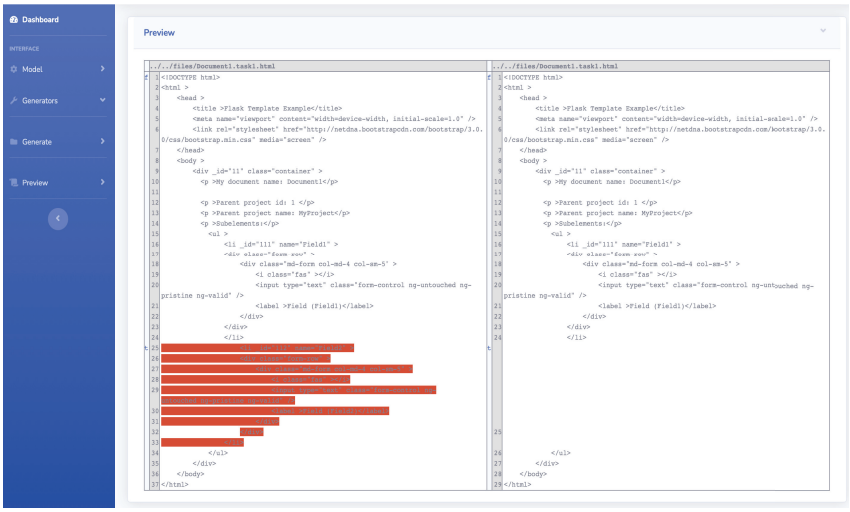


Figure 7 Code preview example for model element deletion.

(Figure 5). The given answers are saved, so the developer does not have to spend more time answering when the actual generation process starts. However, if needed, the given answers can be annulled.

4 An Example of SeamlessMDD Framework Usage

To demonstrate the applicability and usability of the SeamlessMDD framework, this section provides an example of its use by automating the extension of a part of an existing web application developed by a different team. The sample we used is a server-side application that supports a Rent-a-Car service. The company has branches, each offering a different set of vehicles. Users can search for vehicles based on their location, model, make, and availability. After finding a suitable car, a user can rent it and return it to the same or a different location.

The application was developed manually, with the model presented in Figure 9 serving only as a specification document. The development team utilized a simple UML profile (Figure 8) to enrich a UML class diagram and precisely specify the behavior of the web application. The *Resource* stereotype extends the UML metaclass *Class* with a set of tagged values that describe implemented operations (*Create*, *GetOne*, *GetAll*, *Remove*, *Edit*), providing semantics for typical Web API operations. The stereotype *Identifier*

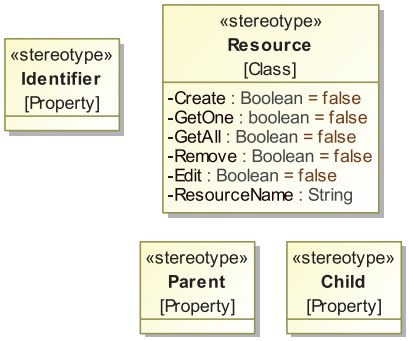


Figure 8 A simple UML profile for enhancing class diagrams with the semantics of web applications.

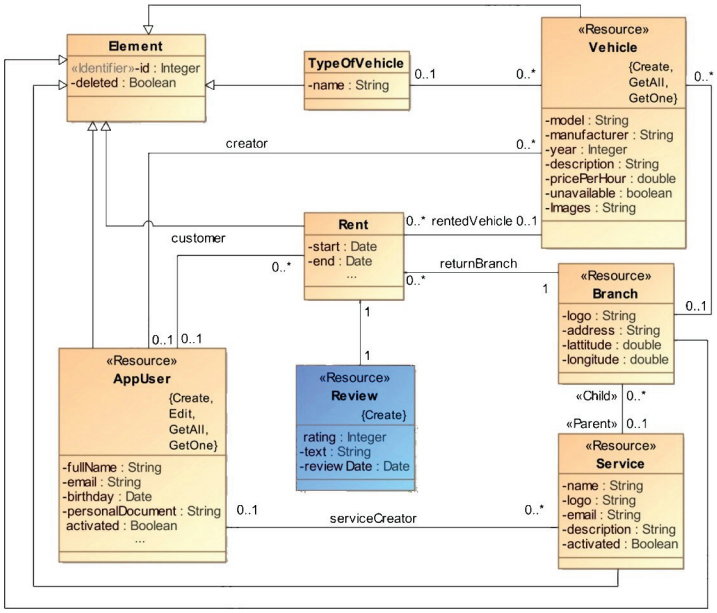


Figure 9 A part of the data model for the Rent-a-Car web application. *Review* is a new resource that needs to be added to the codebase by the SeamlessMDD tool.

marks the identifying property, while the *Parent* and *Child* stereotypes define the parent-child relationship between classes.

The sample application implements a typical range of functionalities, such as data management and persistence, handling HTTP requests, and adhering to common validation and security practices. It was developed using

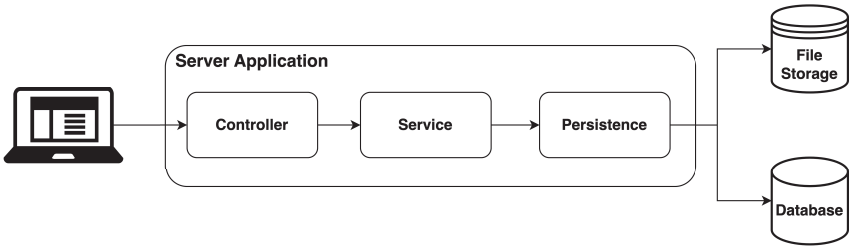


Figure 10 Architecture of the Rent-a-Car web application.

C# and the ASP.NET framework, with the Entity Framework serving as the object-relational mapper and Unity for dependency injection.

Given that the sample architecture consists of several layers (as displayed in Figure 10), including Persistence, Service, and Controller, any model changes must be propagated across these layers, impacting multiple files in each layer. Furthermore, model changes may also affect migrations and database scripts, necessitating distinct approaches to code generation for different layers.

To use SeamlessMDD for automating the further development of the Rent-a-Car application, the first step was to create a meta-model based on the PyECore meta-metamodel. This meta-model needed to provide concepts presented in a UML class diagram, incorporating applied stereotypes from the UML profile shown in Figure 8.

The second step was to create a portion of the Rent-a-Car model based on this meta-model. In our example, we aimed to add a new resource named *Review*, which would allow customers to express their satisfaction or dissatisfaction with a vehicle rental by leaving comments and a numeric rating from 1 to 5.

We needed to specify only the affected model elements: the existing resource *Rent*, the new resource *Review*, and an association between them. The model was created using the SeamlessMDD form-based modeling interface.

When we initiated the first code generation cycle, the changes made in the codebase are shown in Table 1.

Since the model element *Review* did not exist previously, the first generation cycle is mostly straightforward for creation of its artifacts. The majority of updates in the codebase are introduced simply by evaluating developed templates in batch mode. To generate C# classes for the first time, we exploit the partial class concept to keep handwritten and generated code separate.

Table 1 Updates to the codebase after the first generation cycle

Layer	Change	Type of Code Generation	File
Model	Create a *Review* model class	Batch	Review.cs
Model	Create a partial class for *Review* model class	Batch	Review.generated.cs
Model	Insert a new *Review* property in the class *Rent*	ABG	Rent.cs
Persistence	Add DBSet to DBContext	ABG	DBContext.cs
Persistence	Create C# migration for *Review* DB table creation	Batch	AddReviewTable.cs
Persistence	Create a .designer migration descriptor	Batch	AddReviewTable.designer.cs
Persistence	Create a *Review* repository interface	Batch	IReviewRepository.cs
Persistence	Create a *Review* repository class	Batch	ReviewRepository.cs
Configuration	Add a *Review* repository to dependency injection configuration	ABG	IUnitOfWork.cs
Service	Create a *Review* service class	Batch	ReviewService.cs
Controller	Create a *Review* controller class	Batch	ReviewController.cs
Configuration	Add new classes to .csproj	ABG	RentApplication.csproj

However, updates to existing code in the first generation cycle are more complex. When adding a new resource *Review* to the model, it is necessary to introduce the references to the other model elements; hence, a new property named *Review* needs to be added to the existing class *Rent*, referencing the newly added *Review* resource. First, the tool checks if the class *Rent* exists. If it does not, the tool raises an issue. In the next step, the tool examines whether the Rent.generated.cs partial class has already been created. If not, the tool creates it using the template. Next, the tool uses ABG to navigate the class's syntax tree and position itself on the relevant node. After that, new nodes are created to represent the *Review* property using ABG.

Some time after the code has been generated, a developer manually introduces the *Create* operation into the source code of the *Review* resource to support an HTTP POST request. Without being informed, the modeler also extends the model with the *Create* operation by setting the *Create* attribute of the *Review* resource to true, which leads to a conflict. Table 2 displays the updates that are introduced to the codebase following this modification.

The *PostReview* method is manually added in the *ReviewController* by the developer, and adding it again by the code generator would create ambiguity.

Table 2 Updates to the codebase after adding the *Create* operation

Layer	Change	Type of Code Generation	File
Service	Update the *Review* service class with a new operation	ABG	ReviewService.cs
Controller	Update the *Review* controller class with a new operation	ABG	ReviewController.cs

The validation phase of the second generation cycle detects these inconsistencies and raises an issue through a question-and-answer mechanism. The question informs the modeler that a method named *PostReview* already exists, with possible solutions being to skip this particular generation step, generate another method with a different name, or replace the existing method with the generated implementation.

The ASP.NET framework supports several methods for creating Web API routes, one of which is convention-based. The core idea is to use predefined rules to map URLs to controller actions. This approach requires that controller methods adhere to a predefined naming convention. Since the development team followed this convention, it has been built into code generators so that potential conflicts could be detected.

5 Experiences and Validation

SeamlessMDD was developed in the context of providing customization for a large-scale software solution. In this section, we aim to reflect on the benefits and constraints observed during the development of this project.

5.1 Internal Evaluation

The SeamlessMDD approach is particularly effective and convenient for developers in terms of increased productivity, improved code quality, ease of use, and scalability.

Productivity evaluation. Adopting the seamless approach significantly decreased the time required to develop and deploy the software system. Once the code generators are developed and tested, there is minimal need to implement boilerplate code, which notably reduces manual, time-consuming tasks. Although there is an initial cost associated with creating meta-models, models, implementing generators, and performing training, the benefits are quickly realized when applied across several projects.

Increased code quality. Generated code provides consistency throughout the entire solution. This consistency can enhance the readability of the source code, but the main advantage is that the behavior of the implementation remains consistent as well. A downside is that if an error exists in the generated code, it will appear in multiple locations. However, once the error is fixed in the code generator, that fix will also be applied consistently throughout.

Importance of style. To maintain a consistent style across the entire codebase, it was necessary for the generated code to adhere to the same coding standards as handwritten code. This improves readability and facilitates code analysis. The significance of this practice is even greater when the generated code is intertwined with manually written code, as is the case in our approach. From our experience, we have learned that this recommendation can be extended even further by stating that it is not only necessary to follow general coding standards in the generated code, but that the generated code should sometimes adapt to the style of a particular existing artifact when integrated into it.

We encountered an example of this while developing the commercial version of our tool, described in 5.2. One of the artifacts that needed to be generated required inserting lines into an existing class. While the code generated by our tool compiled and executed correctly, developers quickly reported that the result was unreadable and unmaintainable. The reason for this was that the class had very specific formatting for both code and comments, which the generated code misaligned. After some analysis, we determined that the original formatting style of the class was reasonable and not subject to change. The solution was to modify the generated code. However, implementation was not as simple as changing a template. Actual formatting varied across different parts of the class, as well as across various versions of the class in several existing projects that needed to be maintained. To match it, the surroundings of the line where the generated code needed to be inserted had to be analyzed first. An additional challenge was that most existing parsers for languages in which white spaces are not part of the syntax simply ignore them. To address this concern, we developed a library named RoseLib to simplify the use of Microsoft's Roslyn parser for C#. Further details on this implementation can be found in [28].

From this example, we may conclude that adapting the style of generated code to the style of each artifact where it is to be inserted is a problem that may not have a general solution. Still, we maintain that it is worth considering, at least in special cases.

Ease of use. Great attention was paid to intuitive and comprehensible use.

- Visual, form-based model creation and customization renders familiar the developers, saving them from the need to master complex modeling tools.
- Like other MDD tools, we support thorough validation to reduce possible errors from model creation to code generation. In addition, our tool offers mechanisms for conflict resolution.
- The ability to easily revert an entire generation cycle and annul introduced changes gives the developers the necessary confidence to experiment and become familiar with our tool. Furthermore, the preview screen allows developers to easily notice mistakes introduced in the modeling phase.

Scalability. Large-scale software solutions can benefit significantly from MDD but the introduction of this approach must be feasible and straightforward. To achieve this:

- We allow defining the model for only the relevant parts of the potentially large system. The model can then be gradually expanded to cover additional components as needed.
- In the model differencing phase, we extract a minimal set of identified changes. This impacts the subsequent phases, the scale of the transformation, and ultimately the number of modified artifacts.

5.2 External Evaluation

The principles implemented by the SeamlessMDD framework were tested in practice within the complex MDD tool custom-made for and in cooperation with the Schneider Electric (SE) development center in Novi Sad. SeamlessMDD is an open-source re-implementation of its core.

The development of the SE MDD tool aimed to increase the speed of customization of one of SE's large-scale software products to different SE clients' needs, reduce costs, and minimize the number of introduced errors caused by manual implementation. Automating the development process was very suitable, due to the complex architecture of the solution in question, where any change of even minute details of a requirement could result in changes spread across many layers and parts of the implementation. As an example, adding a single field to a form could in some cases yield changes or addition of dozens of artifacts in different languages. The product at stake was a mission critical system, where the state of such fields could affect many

aspects of the operation, including control of physical devices. Errors in such a system can result in severe consequences, including the loss of life.

However, changing the architecture of the system or the development process just to enable classic MDD techniques such as code separation was impossible, given the large number of software projects in various stages of the life cycle that the tool had to support and the large number of developers and software teams working on those projects. Most of the requirements presented in 3.1 were specified during the analysis of what SE MDD tool should support to enable automation in such an environment, as well as its subsequent use in development. The rest of the requirements emerged while developing and verifying the tool core.

During the two years of gradual, incremental development, the SE MDD tool was used on 10 customization projects and accepted by five teams specialized in different technologies. The basic tool core and the first set of 12 API-based generators for developing a part of the desktop application chosen for a pilot project were implemented in three months. The rest of the time was spent enriching the core and developing API-based code generators that worked on several hundreds of file artifacts within two SE complex software products. The tool has been in use all along; only the amount of handwritten code has gradually decreased over time as the scope of supported parts has grown. A requirement that previously took an average of three weeks per team for manual development could now be completed within two to three days when using the tool, assuming optimal organization. The most productive team consisted of specialists who covered all the necessary technologies, ranging from database experts to web front-end developers. On the other hand, separate teams organized based on technology expertise (database, web, desktop) and working on the same feature required more time due to increased cooperation needs.

In addition to optimizing the development process, the tool was used during the requirement specification phase of project customization for clients. By virtue of the tool, within hours, it was possible to provide clients with a prototype of the needed customization, running inside the actual software in a test environment.

The SE MDD tool was developed in C# using the .NET compiler platform[5] extended by our RoseLib[6] library [28] and four parsers for HTML, XML, XAML, and T-SQL.

[5]https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk

[6]https://github.com/nenadTod/RoseLib

6 Conclusion and Future Work

This paper presented a novel approach to developing MDD tools aimed at providing a gradual and effortless transition from traditional development to MDD. It is based on the premise that the developers could better accept the MDD approach by allowing them to maintain the current workflow, system architecture, and the existing level of code readability, along with the freedom to experiment and revert introduced changes. All of the above is supported by the SeamlessMDD framework by implementing: (1) incremental and iterative transformations derived from model versions comparison; (2) integration of generated and handwritten code using API-based code generators; (3) case-specific support for change propagation and conflict resolution.

The SeamlessMDD framework is an open-source re-implementation of the core of an MDD tool developed inside Schneider Electric, which was successfully used in software projects in different life cycle stages. SeamlessMDD was developed in Python3, using PyECore7 for meta-models specification. The original tool was developed in C#.

However, this approach has some drawbacks. While it may be convenient for the developers (MDD tool users), implementing the MDD solution in this way is more demanding for the team creating it. On average, it took 20 person-hours to develop an API-based code generator for a specific file artifact. In contrast, developing a code generator for the same artifact using templates and batch transformation only took an average of two hours. The implementation of API-based code generators requires significant effort due to the need to specify every detail of code manipulation at the syntax tree level.

Another challenge arises when previously developed API-based generators need to be modified to accommodate changes in the architecture of the solutions for which the tool generates code. Some developers who use the SE MDD tool took on the responsibility of maintaining the API-based code generators for their respective parts of the solution, while others required assistance from the tool developers. Finding an efficient way to simplify the development of API-based generators became necessary.

The first step in facilitating the development of API-based generators for C# was to create our RoseLib library [16, 28, 30] on top of the .NET Compiler platform, which allows for the manipulation of coarse-grained code fragments. The second step was to automate the creation of API-based generators using machine learning (in progress).

Besides optimizing the development of API-based generators, we plan to enhance the SeamlessMDD framework by incorporating a graphical

modeling environment for languages specified by PyECore7. The SE MDD tool features a complex form-based user interface that is customized for specifying the concrete SE solutions. The form-based model specification was used to enable different kinds of experts to use or experiment with the tool, including SMEs (subject matter experts). In addition, SeamlessMDD should support modeling in a generic way for all PyECore7-based meta-models provided.

Acknowledgement

This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-137/2025-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project "Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad 2025" (No. 01-50/295).

References

[1] Acceleo. https://eclipse.dev/acceleo. Accessed: 2024-04-16.
[2] Mendix. https://www.mendix.com. Accessed: 2024-04-16.
[3] Webratio. https://www.webratio.com. Accessed: 2024-04-16.
[4] F. P. Basso, R. Mainardi Pillat, T. Cavalcante Oliveira, F. Roos-Frantz, and R. Z. Frantz. Automated design of multi-layered web information systems. *Journal of Systems and Software*, 117:612–637, 2016.
[5] C. Bernaschina, E. Falzone, P. Fraternali, and S. L. Herrera Gonzalez. The virtual developer: Integrating code generation and manual development with conflict resolution. *ACM Transactions on Software Engineering and Methodology*, 28:1–38, 2019.
[6] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition, 2017.
[7] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
[8] P. de Lange, P. Nicolaescu, T. Winkler, and R. Klamma. Enhancing mdwe with collaborative live coding. *Modellierung 2018*, 02 2018.
[9] Peter de Lange, Petru Nicolaescu, Alexander Tobias Neumann, and Ralf Klamma. Integrating web-based collaborative live editing and wireframing into a model-driven web engineering process. *Data Science and Engineering*, 5:240–260, 2020.

[10] E. Falzone and C. Bernaschina. Intelligent code generation for model driven web development. In Cesare Pautasso, Fernando Sánchez-Figueroa, Kari Systä, and Juan Manuel Murillo Rodríguez, editors, *Current Trends in Web Engineering*, pages 5–13, Cham, 2018. Springer International Publishing.

[11] F. Fieber, N. Regnat, and B. Rumpe. Assessing usability of model driven development in industrial projects. *ArXiv*, abs/1409.6588, 2014.

[12] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. M. S. Nazari, K. Müller, A. N. Perez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann. A comparfison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 74–85, 2015.

[13] S. Johann and A. Egyed. Instant and incremental transformation of models. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 362–365, 2004.

[14] F. Jouault and M Tisi. Towards incremental execution of atl transformations. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations*, pages 123–137, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[15] A. Kusel, J. Etzlstorfer, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schönböck, W. Schwinger, and M. Wimmer. A survey on incremental model transformation approaches. *CEUR Workshop Proceedings*, 1090:4–13, 2013.

[16] T. Lalošević, Ž. Vuković, and G. Milosavljević. A meta-model and code generator for evolving software product lines. In *Konjović, Z., Zdravković, M., Trajanović, M. (Eds.) ICIST 2020 Proceedings*, pages 123–128, 2020.

[17] T. Le Calvar, F. Jouault, F. Chhel, and M. Clavreul. Efficient atl incremental transformations. *The Journal of Object Technology*, 18:2:1, 01 2019.

[18] M. Lienhardt. Pydop: A generic python library for delta-oriented programming. *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B*, 2023.

[19] M. Nieke, A. Hoff, I. Schaefer, and C. Seidl. Experiences with constructing and evolving asoftware product line with delta-oriented programming. In *Proceedings of the 16th International Working Conference on*

Variability Modelling of Software-Intensive Systems, VaMoS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[20] B. J. Ogunyomi, L. Rose, and D. Kolovos. Incremental execution of model-to-text transformations using property access traces. *Software & Systems Modeling*, 18:1–17, 02 2019.

[21] B.J. Ogunyomi. *Incremental Model-to-Text Transformation*. PhD thesis, University of York, 2016.

[22] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, pages 77–91, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[23] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.

[24] B. Selic. The pragmatics of model-driven development. ieee softw. 20(5), 19-25. *Software, IEEE*, 20:19–25, 10 2003.

[25] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September 2003.

[26] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

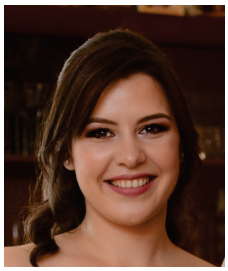
[27] N. Todorović, B. Dragaš, and G. Milosavljević. Supporting integrative code generation with traceability links and code fragment integrity checks. In M. Trajanovic, N. Filipovic, and M. Zdravkovic, editors, *Disruptive Information Technologies for a Smart Society*, pages 490–501, Cham, 2024. Springer Nature Switzerland.

[28] N. Todorović, A. Lukić, B. Zoranović, R. Vaderna, Ž. Vuković, and S. Stoja. Roselib: A library for simplifying .net compiler platform usage. In *ICIST 2018 Proceedings*, pages 216–221. Information Society of Serbia - ISOS, 2018.

[29] A. Vogelsang, T. Amorim, F. Pudlitz, P. Gersing, and J. Philipps. Should i stay or should i go? on forces that drive and prevent mbse adoption in the embedded systems industry. In M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, and D. Winkler, editors, *Product-Focused Software Process Improvement*, pages 182–198, Cham, 09 2017. Springer International Publishing.

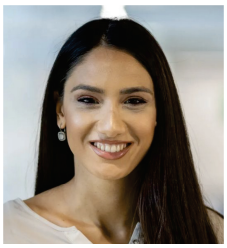
[30] B. Zoranović, N. Todorović, Ž. Vuković, A. Lukić, and G. Milosavljević. Testing of large scale model-driven solutions. In *Konjović, Z., Zdravković, M., Trajanović, M. (Eds.) ICIST 2019 Proceedings*, pages 174–177, 2019.

Biographies



Bojana Dragaš is a teaching assistant and a PhD student of the Faculty of Technical Sciences, University of Novi Sad, Serbia. Her research interest fall under the field of Model-Driven Engineering (MDE).

Nenad Todorović is a PhD student at the Faculty of Technical Sciences, University of Novi Sad, Serbia. His research interests are related to Model-Driven Engineering (MDE) and its application in different industrial settings.



Tijana Rajačić is a technical team lead in Schneider Electric development center in Novi Sad, Serbia. She is interested in design patterns, clean code and Model-Driven Engineering.

Gordana Milosavljević is a full professor at the Faculty of Technical Sciences, University of Novi Sad, Serbia. Her research interest fall under the field of Model-Driven Engineering (MDE), agile methodologies and business informatics.

Željko Vuković is an assistant professor at the Faculty of Technical Sciences, University of Novi Sad, Serbia. He does MDE, networking and network security for work and firefighting for fun.