Towards Integrating Low-Code in View-based Development

Anne-Kathrin Hermann
anne-kathrin.hermann@student.kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Lars König
lars.koenig@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Erik Burger
burger@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Ralf Reussner
reussner@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

ABSTRACT

In recent years, low-code development has been established as an innovative method for software development. It enables the development of a wide range of applications using graphical tools, with little or no knowledge of text-based programming languages. Closely related is model-driven development, where models play a primary role in specifying software systems and generating code partially automatically. While model-driven development supports development processes where developers from different domains work on different models that are kept consistent, in practice, classical model-driven tools are often difficult to use for domain experts with a less technical background. To bridge this gap, we propose a concept for integrating low-code platforms through projective views into model-driven development environments. We provide an initial evaluation of the feasibility of our concept using a development platform for smart home systems as a case study.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

Model-driven Development, Low-Code, View-based Development, Consistency Preservation

ACM Reference Format:
Anne-Kathrin Hermann, Lars König, Erik Burger, and Ralf Reussner. 2024. Towards Integrating Low-Code in View-based Development. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3688333

1 INTRODUCTION

Low-code development platforms (LCDPs) are primarily used to develop software [22, 25]. They enable the creation of a wide variety of applications using graphical tools and with little or no knowledge of a text-based programming language. Especially domain experts, who may have no knowledge of software development techniques, can use LCDPs to create complex applications. But LCDPs tend to be limited to the provided functionality and rigid in terms of the offered extensibility [24, 9]. They often offer standalone solutions without the possibility of using them in combination with other tools.

In contrast, model-driven techniques offer flexibility and options for extensions through, e.g., view-based development. In view-based development, developers use different views to work on the artifacts of a software system. These views consist of semantically related information and need to be kept consistent. For our approach, we focus on projective views on a central model, also called a single underlying model (SUM), which are generated on demand [2]. As a unifying meta-model for a SUM supporting various different views is difficult to construct, we rely on pragmatic SUMs, which are constructed by combining multiple meta-models with explicit consistency preservation rules [2].

Model-driven development, however, also faces various challenges, particularly regarding the usability, management of models, and maintainability of tools [4]. Low-code development places great emphasis on exactly these aspects: providing simple tools that are quick and easy to use. Comparing model-driven and low-code development, Bucaioni et al. [4] point out that even if the motivation, objectives, and technical solutions overlap, there are still differences in terms of usability: "Even if MDE targeted the reduction of complexity and proposed raising the level of abstraction for enabling domain experts to deal with software design, empirical research testifies that IT literacy is required and that there exist several issues related to tool usability, flexibility, and maintenance." [4]

The concept presented in this paper offers an approach to solving these challenges by bridging the gap between low-code and model-driven development. To this end, we have developed a process to integrate low-code development as a view into a view-based system. The view-based system is then able to keep the low-code view consistent with the other views of the system. Through that, developers can work on different levels of abstraction on the same system and choose the views most suited for their tasks. At the same time, as its defining properties are preserved, we retain the advantages that low-code offers. This enables a software development process where domain experts can contribute directly to the creation of a system using LCDPs that are easy to use and learn. As there is no restriction to the predefined functionality of an LCDP, the development platform can be extended through other views, which includes integrating additional low-code platforms.

MODELS Companion '24, September 22–27, 2024, Linz, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688333

In Section 2, we first introduce view-based and low-code development and provide the definition of low-code used in our paper, as there is none established yet. We also examine the differences and similarities between model-driven and low-code development. We then present our process for integrating low-code views into view-based systems in Section 3, which includes different integration strategies. Section 4 demonstrates our process by applying it to a simple, artificial low-code development platform for the development of smart home systems. We conclude our paper by highlighting related work in Section 5 and providing a summary and an outlook on the next research steps in Section 6.

2 FOUNDATIONS

2.1 View-based Software Development

During the lifetime of a software system, different developers work on its artifacts and thus have different views of the system, with their interests lying in different aspects [10]. In model-driven development, the idea of view-based development is to provide developers with *views* that contain relevant, semantically related information. Similar to meta-modeling, where models follow the definition of a meta-model, views follow the definition of a *view type* [10]. In this paper, we focus on *projective* view-based approaches, in which views are generated from an underlying model on demand [2]. The views themselves are transient, which requires the underlying model, also called the *single underlying mode (SUM)*, to contain all the information about the system. For editable views, changes to a view are therefore always committed back to the SUM.

In view-based development, a distinction is made between essential SUMs and pragmatic SUMs, also called *virtual single underlying models (V-SUMs)* [17]. In contrast to essential SUMs, where the SUM consists of a single, redundancy-free meta-model, pragmatic SUMs are assembled from multiple meta-models. Two examples of pragmatic SUMs, created as parts of our case study, are shown in Figure 3.

2.2 Consistency Preservation

Due to the fact that in view-based development, multiple views are used to develop a system, consistency between the views is important for the realization of the system [8]. Using projective view-based approaches, the challenge of maintaining consistency is not between the different views, but shifted to the SUM, from which the different views are projected [14]. With essential SUMs, as introduced in Section 2.1, consistency is maintained implicitly, as the SUM consists of a single, redundancy-free meta-model. With pragmatic SUMs or V-SUMs, however, explicit consistency specifications between the included meta-models are required.

One framework implementing a V-SUM is Vitruvius [15]. It enables the use of different models to describe a system, which are automatically kept consistent by (semi-)automatic consistency rules executed by the framework. Vitruvius supports different languages for the specification of consistency preservation rules. One of them is the Reactions language, an imperative, delta-based language for the specification of unidirectional consistency preservation rules. An example of a reaction between a UML model and a Java model is shown in Listing 1. The reaction itself contains a trigger (`after element [...] created and inserted as root`) on the UML model and a `call` block, in which routines are called that keep the Java model consistent. Routines consist of three blocks: `match`, `create`, and `update`. While the `match` block is used to retrieve elements from the source and target models, the `create` block is used to create new elements in the target model only. In the `update` block, arbitrary code can be executed to, e.g., set attributes, insert newly created model elements, or move existing ones.

```
reaction CreatedUmlClass {
    after element uml::Class
        created and inserted as root
    call {
        val umlClass = newValue
        createJavaClass(umlClass)
    }
}

routine createJavaClass(uml:Class umlClass) {
    match { /* retrieve_elements */ }
    create { /* create_elements */ }
    update { /* update_models */ }
}
```

Listing 1: Example of a reaction that creates a new class in a Java model whenever a class is created in a UML model.

In addition to automatic consistency preservation, there are also approaches that check and, if possible, try to repair inconsistencies between models. There are, however, consistency relations between models that cannot be preserved or repaired automatically, e.g., when additional information from a developer is required [14]. In practice, there are also cases where tolerating inconsistencies is reasonable [7], e.g., when they can only be resolved at a later stage of development.

2.3 Low-Code Development

In recent years, low-code development has become increasingly popular as a new way of developing software. Instead of using traditional programming languages, applications are usually created by dragging and dropping prebuilt modules [12]. With the help of graphical visualization, so-called citizen developers, who have little or no programming knowledge, can create complex programs [9]. Low-code development platforms, which permit the use of source code in certain areas, are differentiated from no-code platforms, which enable software development without requiring any programming knowledge. An essential feature of low-code development is therefore the elimination of text-based programming through visual languages [9]. This hides low-level concerns from developers, as low-code development platforms (LCDPs) allow them to focus on higher levels of abstraction.

The market research company Forrester Research coined the term low-code in 2014, defining low-code platforms as “Products and/or cloud services for application development that employ visual, declarative techniques instead of programming” [23]. Forrester Research predicted a promising future for these platforms [23].

Although the term low-code was already coined in 2014, the first peer-reviewed publications on low-code development have not been published until 2018. Since then, there has been a significant increase in related publications [4]. As low-code development is still quite young, there are aspects that are relatively unexplored, and an established definition in the community is also lacking. As mentioned above, there is a definition from Forrester Research, which has also significantly characterized the term low-code [23] but is missing some key aspects. Bucchiarone et al. [5] also describe these difficulties, as LCDPs were inspired by various modeling paradigms and were mostly adapted to the various domains. They define the primary objective of low-code development platforms as transferring programming tasks from software developers to domain experts. Bucaioni et al. [4] provide the following definition after an extensive literature review: They see LCDPs as "a set of methods and/or tools in the context of a broader methodology, being in this case MDE" [4].

Many also define low-code by looking at the fundamental properties. Among these, certain properties are repeatedly mentioned, from which a definition can be derived. For Di Ruscio et al. [9], these platforms are characterized by their reduction of technical complexity associated with the installation and operation of both development environments and the applications created. This is relevant because low-code is primarily aimed at citizen developers. Usually, the reduction in complexity is achieved by providing cloud-based development environments [9]. Furthermore, according to Di Ruscio et al. [9], every development with an LCDP consists of typical, tool-based steps: modeling of the domain, definition of a user interface, specification of the business logic, integration with external services, application generation and deployment, and maintenance [9]. Hinrichsen et al. [12] confirm this characterization: in their view, special features of low-code development environments are a simple setup and the ability to reduce complexity in operational processes. However, they also point out that, although these platforms offer many advantages, their complex inner workings usually remain hidden. The literature review by Pinho et al. [21] identifies eight characteristics of low-code development platforms. These include non-programmers as users, the use of visual tools and drag-and-drop functions, increased abstraction, a low level of code-based programming, model-based software development, rapid application development, software lifecycle management, and the use of cloud resources.

As there is no established definition of the term *low-code*, we deem it necessary to clarify with which meaning we use the term. Based on the frequently mentioned characteristics of low-code, we therefore make the following definition for our paper:

Definition 1. *Low-code development platforms enable domain experts to develop systems with little or no programming effort using visual languages and higher levels of abstraction by reducing technological, representation-induced, and domain-wise complexity.*

2.4 Low-Code and Model-Driven Development

Current research has not conclusively clarified the relationship between model-driven and low-code development. While Bucaioni et al. [4] and Cabot [6] regard them as synonymous, Di Ruscio et al. [9] and Hinrichsen et al. [12] highlight clear differences. For example, according to Di Ruscio et al. [9], not all model-driven techniques aim to reduce the amount of source code required, and not all low-code approaches are clearly model-driven [9]. In addition, some model-driven approaches do not include deployment and lifecycle management, whereas these aspects are nearly always integrated into low-code development platforms (LCDPs). Furthermore, the user profiles differ, with low-code development platforms often involving citizen developers who have less of a software engineering background, while users of model-driven development (MDD) techniques have a stronger technical focus. There are also low-code approaches that do not use models with explicitly defined languages or meta-models, but instead store data in relational databases or in schema-less XML/JSON documents [9].

According to Hinrichsen et al. [12], model-driven software development, generative programming, and low-code programming share many similarities. This viewpoint is supported by Forrester [23], Cabot [6], and Bucchiarone et al. [5]. However, differences between these approaches become apparent when considering the types of modeling languages used and the underlying software architecture [12]. The emphasis of LCDPs is on platform usability rather than the precise specification of the elements used [12]. Uyanık and Sayar [30] point out the importance of model-based approaches in automated code generation.

A different perspective is presented by Bucaioni et al. [4], who state that "it is widely accepted in the MDE [model-driven engineering] community to consider LCD [low-code development] as some sort of synonym for MDE, or to consider MDE techniques as foundations for LCD solutions" [4]. They see similarities primarily in the motivation, objectives, and technical solutions, while Di Ruscio et al. [9] highlight differences in precisely these areas. According to Bucaioni et al. [4], in some literature, low-code development is even viewed as a further development or maturity stage of MDE, particularly regarding usability and flexibility [4].

For Cabot [6], low-code development is a limited view of MDE in which only data-intensive web or mobile applications are considered. Furthermore, low-code can be viewed as a solution with a fixed language, as the underlying language is usually not visible in LCDPs. Nevertheless, LCDPs are more popular because they offer clear application scenarios and are less complex, whereas the modeling tools in MDE tend to be rather unwieldy and complex [6].

With regard to our Definition 1, low-code and model-driven development have similarities when it comes to reducing handwritten code and a higher level of abstraction regarding content. However, there are also distinct differences, as a current challenge in model-driven development is the lack of effective, easy-to-use tools [7], while low-code focuses mainly on this. Furthermore, model-driven development is not limited to visual languages but uses them in combination with textual ones [12].

3 CONCEPT

Our concept for integrating low-code development platforms into a view-based system includes retrieving the low-code meta-model, determining the related meta-models, as well as different strategies for integrating the low-code meta-model. Depending on the low-code development platform and the data available, different techniques can be used for retrieving or creating a meta-model, which we describe in Section 3.1. Although, in the end, consistency with a source code view is required, the meta-models directly related by consistency specifications or view definition can be different, as described in Section 3.2. In Section 3.3, we then discuss two different strategies for integrating a low-code view in the V-SUM with respect to the available meta-models. The entire process of instantiating our concept is shown in Figure 1.

3.1 Meta-model

For the integration of low-code views in a model-driven environment, it is essential to have a meta-model of the information represented in the low-code development platform (LCDP). As explained in Section 2.4, low-code and model-driven development are not mutually exclusive, i.e., there are LCDPs that employ model-driven techniques. We can therefore distinguish four cases:

(1) The LCDP is implemented using model-driven techniques, and the meta-model of the represented information is available.
(2) The LCDP is not implemented using model-driven techniques, but a schema of the represented information is available, e.g., an XML schema.
(3) The LCDP does not provide a meta-model or a schema, but created models can be exported and imported, e.g., in an XML format.
(4) The LCDP does not allow exporting or importing any data.

In the first case, the meta-model can directly be used to integrate the LCDP into a consistency-preserving development process. The second and third cases account for LCDPs that do not use model-driven techniques or do not make their meta-model available. If, in the second case, however, the LCDP provides another description of the format of the represented data, e.g., in the form of an XML schema, this description can be used to derive a meta-model. As an example, Neubauer et al. [19] show how to generate meta-models and Xtext grammars from XML schema definitions (XSDs), which could be used in this case. The third case describes a scenario where the LCDP permits the export and import of data but without a description of its structure, e.g., as XML files without a schema definition. In this case, techniques for deriving XSDs from XML data [3] can be combined with the work of Neubauer et al. [19] for deriving meta-models from XSDs. In the fourth case, the LCDP cannot be integrated at all, as data transfer from and to the LCDP is not possible. This can be the case if the LCDP provides only an executable or, more often, includes deployment and hosting of the created application. Except for the fourth case, where a direct integration is not possible, this shows that a meta-model for the information represented in the LCDP can be retrieved or derived for the integration in a model-driven, consistency-preserving development process.

3.2 Relations to Other Meta-Models

Integrating low-code views into a view-based consistency-preserving development process has the benefit of different views, including a low-code view that is customized for domain experts, being available for different development tasks, e.g., requirements engineering, database schema development, or test specification, which are all kept consistent. For integrating low-code views, consistency specifications to all related meta-models need to be established. A special role is, however, played by the source code view, which is not only the primary artifact of traditional software development, but also required for creating an executable, deliverable product. As with code generation on low-code development platforms, it is therefore

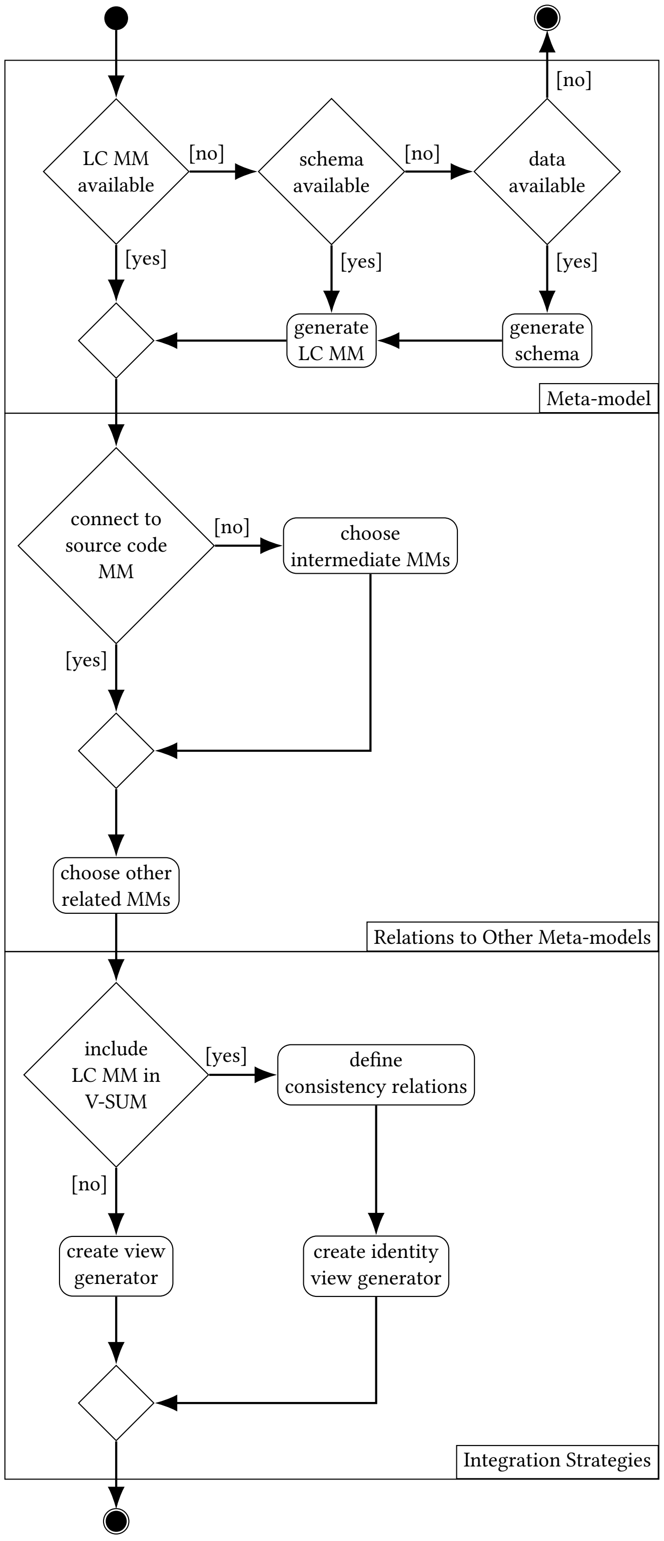


Figure 1: Process of integrating a low-code view from a low-code development platform into a view-based consistency preserving development process.

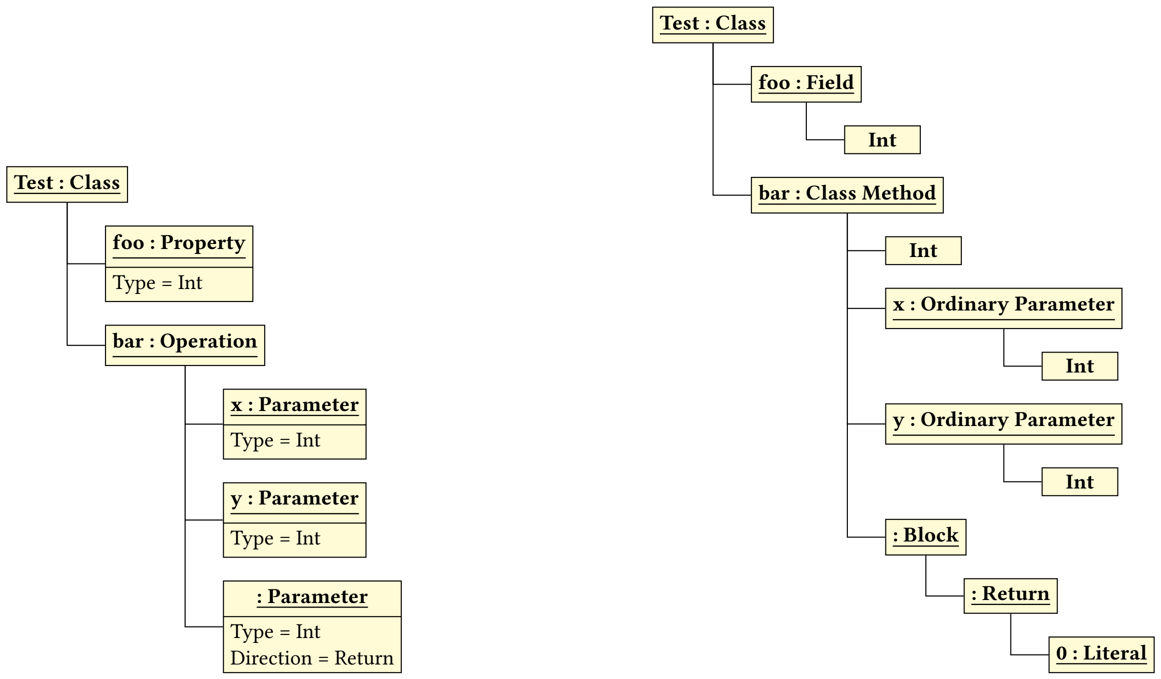


Figure 2: Comparison of a UML model (left) and a Java model (right) for the same class `Test` with an attribute `foo` and a method `bar`. The shown classes `Int` referenced from the objects in the Java model are placeholders for the same class `Int`.

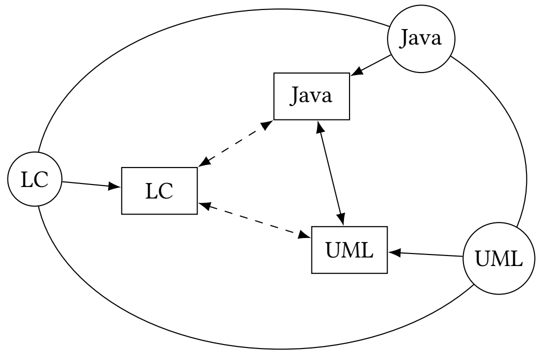
necessary to keep the low-code view consistent with the source code view. But this does not require a direct relationship, e.g., consistency preservation rules, between their meta-models, as other models could be used as intermediate models to achieve consistency with the source code view. Depending on the concrete use case, keeping consistency with a simpler intermediate model can reduce the complexity of the required consistency specifications. An example of this can be seen in the comparison of a UML model and a Java model, for a small example, in Figure 2.

Preserving consistency between a low-code view and a source code view, either directly or via an intermediate model, is a trade-off decision. A pragmatic argument is the availability of meta-models. If there are applicable meta-models already included in the V-SUM for other reasons, keeping the low-code view consistent with these can reduce the effort of consistency preservation. If no applicable meta-models are available, adding a new intermediate meta-model to the V-SUM would require adding consistency specifications both between the low-code view type and the intermediate meta-model and between the intermediate meta-model and the source code view type. The decision to use an intermediate meta-model also depends on the type of low-code view. For structural low-code views, e.g., commonly used meta-models abstracting from the source code, like UML, can be used, while meta-models representing the behavior of source code are less common. In these cases, it can be necessary to define consistency specifications between the low-code view type and the source code meta-model directly.

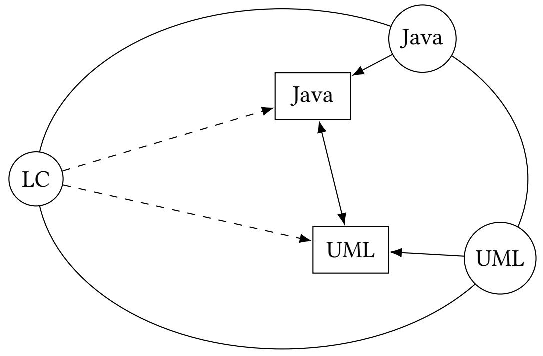
3.3 Integration Strategies

With the meta-model of the low-code view available and the related meta-models in the V-SUM decided, the final step to integrating a low-code view into a V-SUM, is to decide where the information shown in the view is projected from. By that, together with the selection of related meta-models in Section 3.2, we follow the process scenario *view type driven (existing view type)* from the Vitruvius approach [15]. There are two strategies: either the meta-model of the low-code view is included in the V-SUM as one of its internal meta-models and the view is projected directly from that, or the information can be derived from the chosen related meta-models in the V-SUM. Examples of V-SUMs created with the two integration strategies are shown in Figure 3. In both cases the meta-model of the low-code view is used as a view type for the V-SUM, the difference lies in the generation of the view and the consistency preservation in the V-SUM. Both approaches come with benefits and drawbacks, depending on the concrete scenario, which are discussed in the following.

If the meta-model of the low-code view is included in the V-SUM, as shown in Figure 3a, the view generation is the identity function, as the low-code model can directly be used as the low-code view. The complexity, however, lies in defining the consistency preservation rules to the related meta-models. The actual complexity is highly dependent on the connected meta-models, as discussed in Section 3.2, and also depends on the languages available for consistency preservation. For the Vitruvius framework [15] used in our case study, e.g., consistency preservation rules are specified in a low-level delta-based language, which requires considerable effort for defining the necessary rules. Having the low-code meta-model



(a) Example of a V-SUM where the low-code meta-model (*LC*) is integrated in the V-SUM. The dashed arrows represent the possible consistency relations described in Section 3.2.



(b) Example of a V-SUM where low-code views are projected from other meta-models. The dashed arrows represent the possible view type dependencies described in Section 3.2.

Figure 3: Examples of a V-SUM with three view types: A placeholder for a low-code (*LC*) view type, a *Java* and an *UML* view type. The two versions show the two different integration strategies described in Section 3.3. In the figures, rectangles (▭) represent meta-models, circles (◯) represent view types, unidirectional arrows (⟶, - - →) represent view type dependencies and bidirectional arrows (⟷, ← →) represent consistency relations.

available in the V-SUM could, however, be beneficial for connecting additional meta-models, as the consistency specifications to them can be developed independently, or for replacing the connected meta-models. The approach is the only possible solution if the meta-model of the low-code view describes information that is not available in the other meta-models of the V-SUM. It is also the preferred solution if the generation of the view would require complex reconstruction of information, such as the reverse engineering of software components from source code.

Generating low-code views directly from the models of the related meta-models, on the other hand, requires no additional effort for consistency preservation, as no meta-model is added to the V-SUM itself. However, with this integration strategy, the view generation transformation becomes more complex, as the information shown in the view must be derived from the information available in the V-SUM, which can be spread over several models. An example of a V-SUM created with this integration strategy is shown in Figure 3b. The complexity of the view generation transformation is dependent on the included meta-models and the language available for specifying the transformation. In general, however, we expect the transformation to be less complex if the included meta-models have a similar structure as the low-code view type. This could be the case, e.g., when introducing a graphical representation for already available information or when combining information from multiple meta-models. This integration strategy is beneficial if multiple different low-code view types are to be integrated, as it avoids issues present in larger networks of consistency specifications, as discussed in [16]. For both approaches, larger differences in the structure of the low-code meta-model and the related meta-models in the V-SUM require more complex view generation transformations or consistency specifications, respectively. When generating low-code views from related models in the V-SUM, however, these complex transformations need to be executed every time a view is checked out by a developer, which could be a performance issue.

4 CASE STUDY

This section outlines the case study employed to evaluate parts of the concept. For that, the integration of a low-code development platform into a view-based system will be used as an example, following the process shown in Figure 1. The low-code trend has also arrived in the IoT domain, and there are already several low-code development platforms [13], although most of them are not open source. For the sake of simplicity, we have decided to develop our own very slimmed-down version of an IoT low-code development platform, including a meta-model in the Ecore[1] format, such that we could use it directly as described in Section 3.1. We implemented our case study in the view-based, consistency-preserving framework Vitruvius [15], providing an implementation of a V-SUM, as explained in Section 2.1 and Section 2.2. In our scenario, we aim to ensure consistency between the low-code view and the source-code view, although our concept can be applied to keep consistency with other models as well. Therefore, both a Java meta-model [1, 11] and the UML [29] meta-model are within the V-SUM. As our low-code view is a structural abstraction, we can keep the low-code meta-model consistent with the UML meta-model, which simplifies the required consistency specifications, as discussed in Section 3.2. Consistency preservation rules between UML and Java already exist [28], which we use to achieve consistency between our smart home low-code view and the source code view. To keep the models consistent, we chose the first integration strategy described in Section 3.3 and shown in Figure 3a. Following the integration strategy, we integrated the low-code meta-model as an internal meta-model into the V-SUM and developed consistency preservation rules to the chosen UML meta-model. The source code for our case study can be found as a contribution to the Vitruvius projects[2,3].

In Section 4.1, we describe the low-code development platform and the implemented example. The implementation of our concept

[1]https://eclipse.dev/modeling/emf/, last visited on 2024-07-02
[2]https://github.com/kit-sdq/DemoMetamodels/pull/29
[3]https://github.com/vitruv-tools/Vitruv-CaseStudies/pull/295

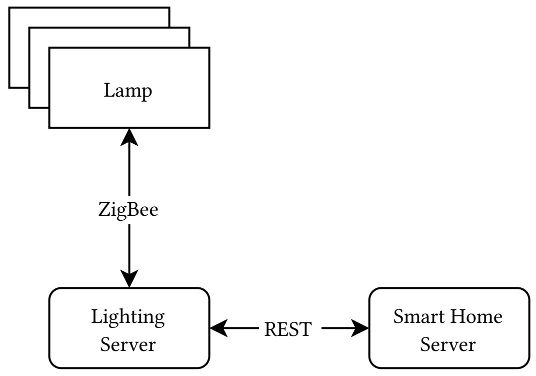


Figure 4: Part of the model of our example smart home system as it would be shown on the UI of our low-code development platform.

is presented in Section 4.2 with the realization of the view, in Section 4.3, in which the meta-model is explained, and in Section 4.4, which is concerned with the consistency preservation relations. Section 4.5 then focuses on the source code in the example.

4.1 A Low-Code Development Platform for Smart Home Systems

We have established a low-code development platform for designing smart home systems as the foundation of our evaluation. Using graphical visualization, a domain expert can model how the various components of such a smart home system should be organized.

Figure 4 shows a part of a designed smart home system featuring devices and servers. Other components of the system could be sensors or interfaces. The example illustrates how the various systems are connected and communicate with each other. The *smart home server* communicates through REST interfaces with the various subsystems, such as the *lighting server*, which controls the lamps. The smart home system shown in Figure 4 can be put together using drag-and-drop within a low-code development platform. However, the actual connection of devices and servers cannot be accomplished without writing source code. For instance, implementing a REST API is necessary for enabling communication between the *smart home server* and other subsystems. After a domain expert has modeled the smart home system on the low-code development platform, another domain expert establishes connections between sensors and servers. This involves implementing the modeled connections and protocols in source code to finalize the smart home system. By integrating this smart home low-code perspective into a view-based system, the planning of the smart home system can occur within the low-code environment, while the protocols are implemented using source code. The view-based system ensures consistency between both views, meaning changes made in the low-code environment are synchronized with the source code and vice versa. This collaborative approach allows experts from different domains to work effectively within their preferred environments.

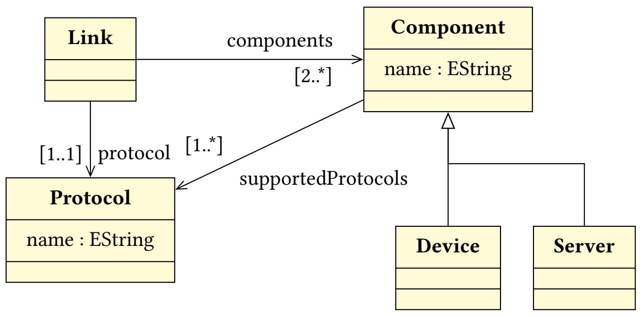


Figure 5: Meta-model for describing a smart home system consisting of devices and servers that are linked using mutually supported protocols.

4.2 Low-Code View

A practical application of our concept would integrate an existing low-code development platform into a V-SUM, without the need to re-implement its functionality or UI. For our evaluation, however, we chose to create a simple, artificial low-code development platform, including a meta-model and corresponding UI. This way, we can focus on and evaluate the integration of the low-code platform.

For the graphical implementation of this UI, we plan to use Eclipse Theia[4]. A Vitruvius, Eclipse Theia is based on the Eclipse Modeling Framework (EMF), which enables us to use it as a frontend for Vitruvius. In general, Eclipse Theia is a free and open-source framework for creating web-based development environments. It is highly modular and extensible, offering extensions including visual tools, drag-and-drop functionality, and other features suitable for a low-code UI. The implementation of the low-code UI, however, constitutes future work and will be implemented in the next stage of our evaluation.

4.3 Low-Code Meta-Model

A meta-model of the low-code view is necessary for implementing the concept described in Section 3. As part of our artificial low-code development platform, we therefore developed a meta-model using the Eclipse Modeling Framework (EMF) to enable its practical integration. Having a meta-model for the low-code development platform available constitutes the case with the least effort, as described in Section 3.1. For an existing low-code development platform without a meta-model available, future work on our evaluation would require deriving the meta-model using techniques mentioned in Section 3.1.

Figure 5 illustrates the meta-model for the low-code development platform designed for creating smart home systems. Components such as devices and servers can be connected using links, each of which is associated with a protocol selected from a list of supported protocols and involves at least two components.

4.4 Consistency Relations with the UML Meta-Model

Based on the meta-models, consistency is implemented with the help of consistency rules. We used the Reactions language [15] to

[4]https://theia-ide.org/, last visited on 2024-07-04

```
reaction ServerInserted {
    after element smarthome::Sever inserted in smarthome
        ::SmartHomeSystem[server]
    call { createServerUmlClass(newValue) }
}

routine createrServerUmlClass(smarthome::Server server) {
    match {
        val model = retrieve uml::Model
            corresponding to UMLPackage.eINSTANCE
        val superClass = retrieve uml::Class
            with it.name == "Server"
    }
    create {
        val class = new uml::Class
    }
    update {
        class.name = server.name
        model.packagedElements += class
        addCorrespondenceBetween(server, class)
        addUmlSuperClassGeneralization(class, superClass)
    }
}
```

Listing 2: Example of a reaction for adding a new server to a smart home model.

connect the smart home meta-model, i.e., the low-code meta-model, with the rest of the view-based system. Instead of maintaining consistency directly with the source code meta-model, Java in our case, we opted to use UML as an intermediate layer, as described in Section 3.2. Since the Reactions language supports only unidirectional consistency rules, we implemented both directions separately.

When a new *SmartHome* system is created, all abstract classes (*Component*, *Device*, *Server*) are created. If afterward a component is created, the UML class inheriting from the abstract class is created with a concrete name. For example, if a new server is created, this must also be done appropriately in UML. Listing 2 shows the implemented reaction for this process. As described in Section 2.2, a reaction consists of different parts. As the first part of a reaction, the event that triggers the reaction is described, in this case the insertion of a *smarthome::Server*. The called routine then retrieves the affected elements (`match`), creates new elements (`create`), and performs actions to preserve consistency in the target model (`update`). In the routine shown in Listing 2, the UML model as well as the abstract class *Server* are retrieved. The newly created UML class is then updated and inserted into the UML model. Finally, a correspondence link between the server in the smart home model and the UML class is created. Regarding the direction from UML to the smart home meta-model, modifications, additions, or deletions are permitted only within subclasses of our generated abstract superclasses. These abstract superclasses themselves must remain unchanged.

4.5 Consistency between Low-Code and Source Code

In our example, as already described, the low-code development platform alone is not sufficient to implement a complete smart home system. For the case study, we have therefore implemented part of the example system, as can be seen in Figure 6. The *LightingServer* implements the REST interface, which contains all the required information for a REST API. The *LightingServer* also manages several lamps. We maintain consistency between low-code and source code through consistency preservation rules between low-code and UML, and the already existing consistency preservation rules between UML and Java [28]. By that, when a change is made, it becomes visible in the other view. For example, if a new lamp is added in low-code, this is then automatically mapped in the source code.

5 RELATED WORK

Low-code development encounters several challenges. One of them is interoperability, i.e., the ability of a tool to exchange information internally (between components) and externally (between services). Low-code development platforms (LCDPs) often offer few or no practices such as versioning, collaboration tools, reuse of program modules, or automated tests, which makes maintenance more difficult compared to classical programming languages [12, 24, 9]. Bucaioni et al. [4] also highlight future concerns regarding the portability, maintainability, and scalability of LCDPs. They identify concerns about potential restrictions imposed by providers, a lack of adaptability, and potential lock-in effects [4, 12]. Pacheco et al. [20] also criticize the fact that many LCDPs have no option to export source code. They are particularly focused on UI/UX LCDPs, such as Figma or Sketch. There are already some plugins that solve this code generation problem, but none that export to a format that can be fed back into an LCDP. This is implemented for OutSystems as an example.

The work of Zaheri [31] addresses the challenge of consistency management on low-code development platforms. They identify

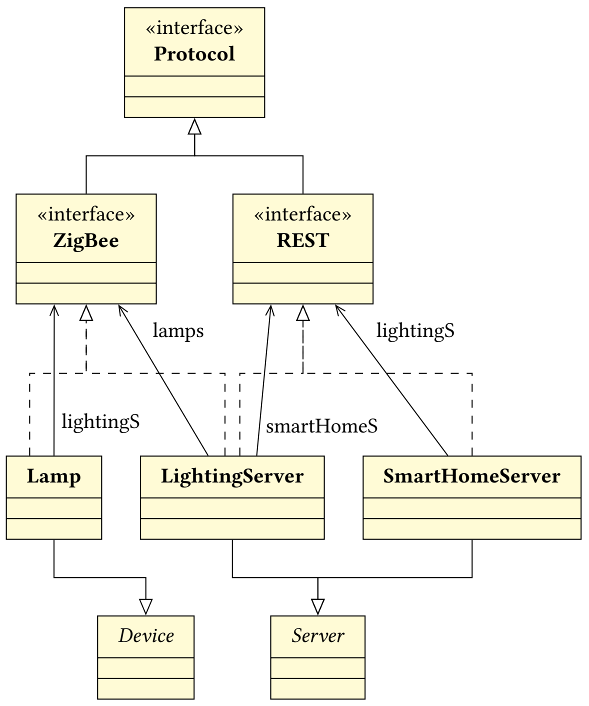


Figure 6: Class diagram for our smart home example.

several key aspects of this issue, including potentially conflicting viewpoints, a lack of separation of meta-levels, and inconsistencies in data and artifacts. Zaheri [31] proposes a solution that includes pre-processing, trace modeling, and consistency rule checking, followed by inconsistency discovery and recovery. They focus on vertical consistency between different levels of abstraction, while our work is independent of these abstraction levels. In particular, their concept deals with different instantiations and inconsistencies between them.

As already mentioned in Section 2.4, there is still no consensus on the relationship between low-code and model-driven development. However, there is already work in progress that attempts to bring them together. Michael and Wortmann [18] present a model-driven low-code approach for the configuration and reconfiguration of digital twins using language-specific plugins. Digital twins are used to monitor and control cyber-physical systems in various domains. The authors integrate their architecture with the code generation framework MontiGem[5] to create interactive digital twin cockpits. This platform allows users to create digital twins using a configuration wizard that conforms to the proposed reference architecture and uses the domain-specific languages available in MontiGem. A model-driven architecture has been developed that fulfills the main advantage of low-code, which is ease of use for domain experts. By doing so, they have built their own LDCP. However, this approach cannot be easily generalized because it is closely tied to this specific case.

On the other hand, the work by Hinrichsen et al. [12] presents a case study on the integration of low-code, generative, and model-driven programming to simplify software development processes in hardware-related areas. This required the development of a dedicated tool chain to ensure flexibility in the use of specific hardware components. An iterative approach was used to create a meta-model for the application domain, which served as the basis for the tool chain. This is thus a concrete application example, while we want to bring these domains together at a higher level of abstraction independent of a specific application.

For the further evaluation of our approach, we want to use Eclipse Theia to implement a UI for the low-code view used in our case study, described in Section 4.2. Something similar has already been done by Saini et al. [26] for collaborative modeling with the graphical User Requirements Notation (URN). Theia was used to build the textual models and generate the corresponding graphical models in the web browser [26]. They then extended this to other model types in general [27]. However, both are focused on real-time collaboration, while we focus on asynchronous collaboration, where changes are committed to the underlying model and then projected to views when required.

6 CONCLUSION

Both low-code development and model-driven development, which includes view-based development, encounter several challenges. Model-driven development, and consequently view-based development, is perceived as cumbersome. These challenges include a lack of good tools, insufficient agility in development and high barriers to entry, particularly for non-technical users. In contrast, low-code development platforms provide precisely that: user-friendly tools that facilitate rapid learning and usage, allowing users to focus entirely on the domain. However, while low-code development platforms tend to be rigid and restrictively feature-limited, a view-based system demonstrates flexibility and can be readily expanded, for instance, by introducing additional views.

The concept presented in this paper combines the advantages of both areas, such that the easy-to-learn and easy-to-use UI of a low-code development platform can be used, while at the same time, the developed artifacts can be extended through a view-based system. Through this, our concept enables development scenarios requiring collaboration between domain experts and, e.g., software developers. By integrating low-code meta-models into a V-SUM using consistency rules, the low-code view is kept consistent with the source code view, enabling both experts to work in their preferred environments on shared artifacts. Our case study shows the feasibility of the concept by linking a low-code view for the development of a smart home system, which we developed ourselves, with existing meta-models in a view-based system. To achieve this, we used the Vitruvius framework [15] and its Reactions language to define consistency preservation rules between the smart home meta-model and the UML meta-model as an intermediate layer towards the source code.

While the limitations of LCDPs are part of the motivation for this paper, they also limit the applicability of our approach, as we require the export and import of data from and to the LCDP to keep it consistent with the other development artifacts. In addition, the effort for specifying the consistency preservation rules or view generation transformations needs to be taken into account.

As future work, we plan to extend our case study by adding a UI for the low-code view in Eclipse Theia to enable an end-to-end evaluation of our concept. We also intend to link a different, behavioral low-code view with the source code model, i.e., without a UML model as an intermediate layer. The next step to validating our concept is to perform a larger evaluation with a low-code development platform that is used in practice. We are also going to implement the other integration strategy described in Section 3.3 in a further case study, i.e., to generate a low-code view from other, non-low-code models.

ACKNOWLEDGMENTS

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263 and KASTEL Security Research Labs.

REFERENCES

[1] Martin Armbruster. 2022. Parsing and Printing Java 7-15 by Extending an Existing Metamodel. en. Tech. rep. DOI: 10.5445/IR/1000149186.
[2] Colin Atkinson, Christian Tunjic, and Torben Möller. 2015. Fundamental Realization Strategies for Multi-view Specification Environments. In *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. ISSN: 1541-7719. (Sept. 2015), 40–49. DOI: 10.1109/EDOC.2015.17.
[3] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. 2007. Inferring XML schema definitions from XML data. In *Proceedings of the 33rd international conference on Very large data bases*. Citeseer, 998–1009.
[4] Alessio Bucaioni, Antonio Cicchetti, and Federico Ciccozzi. 2022. Modelling in low-code development: a multi-vocal systematic review. en. *Software and*

[5]https://se-rwth.github.io/research/MontiGem/, last visited on 2024-07-04

Systems Modeling, 21, 5, (Oct. 2022), 1959–1981. DOI: 10.1007/s10270-021-00964-0.

[5] Antonio Bucchiarone, Federico Ciccozzi, Leen Lambers, Alfonso Pierantonio, Matthias Tichy, Massimo Tisi, Andreas Wortmann, and Vadim Zaytsev. 2021. What Is the Future of Modeling? en. *IEEE Software*, 38, 2, (Mar. 2021), 119–127. DOI: 10.1109/MS.2020.3041522.

[6] Jordi Cabot. 2020. Positioning of the low-code movement within the field of model-driven engineering. en. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, Virtual Event Canada, (Oct. 2020), 1–3. ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3420210.

[7] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. 2019. Multi-view approaches for software and system modelling: a systematic literature review. en. *Software and Systems Modeling*, 18, 6, (Dec. 2019), 3207–3233. DOI: 10.1007/s10270-018-00713-w.

[8] Istvan David, Hans Vangheluwe, and Eugene Syriani. 2023. Model consistency as a heuristic for eventual correctness. en. *Journal of Computer Languages*, (July 2023). DOI: 10.1016/j.cola.2023.101223.

[9] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. 2022. Low-code development and model-driven engineering: Two sides of the same coin? en. *Software and Systems Modeling*, 21, 2, (Apr. 2022), 437–446. DOI: 10.1007/s10270-021-00970-2.

[10] Thomas Goldschmidt, Steffen Becker, and Erik Burger. 2012. Towards a tool-oriented taxonomy of view-based modelling. In *Modellierung 2012. Hrsg.: Sinz, Elmar J.. Fachtagung Modellierung, 2012, Bamberg*. Fachtagung Modellierung. 2012 (Bamberg, Deutschland, Mar. 14–16, 2012). Elmar J. Sinz and A. Schürr, (Eds.) Gesellschaft für Informatik (GI), 59–74. ISBN: 978-3-88579-295-6.

[11] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2010. Closing the Gap between Modelling and Java. en. In *Software Language Engineering*. Mark van den Brand, Dragan Gašević, and Jeff Gray, (Eds.) Springer, Berlin, Heidelberg, 374–383. ISBN: 978-3-642-12107-4. DOI: 10.1007/978-3-642-12107-4_25.

[12] Sven Hinrichsen, Stefan Sauer, and Klaus Schröder, (Eds.) 2023. *Prozesse in Industriebetrieben mittels Low-Code-Software digitalisieren: Ein Praxisleitfaden*. de. *Intelligente Technische Systeme – Lösungen aus dem Spitzencluster it's OWL*. Springer, Berlin, Heidelberg. ISBN: 978-3-662-67949-4. DOI: 10.1007/978-3-662-67950-0.

[13] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. 2020. Low-code engineering for internet of things: a state of research. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (MODELS '20). Association for Computing Machinery, New York, NY, USA, (Oct. 2020), 1–8. ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3420208.

[14] Heiko Klare. 2022. *Building Transformation Networks for Consistent Evolution of Interrelated Models*. PhD Thesis. Karlsruher Institut für Technologie (KIT). DOI: 10.5445/KSP/1000138566. ISBN: 978-3-7315-1132-8, ISSN: 1867-0067, Series: The Karlsruhe Series on Software Design and Quality / Ed. by Prof. Dr. Ralf Reussner, Volume: 34.

[15] Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling consistency in view-based system development — The Vitruvius approach. *Journal of Systems and Software*, 171, (Jan. 2021), 110815. DOI: 10.1016/j.jss.2020.110815.

[16] Heiko Klare, Torsten Syma, Erik Burger, and Ralf Reussner. 2019. A Categorization of Interoperability Issues in Networks of Transformations. en. *The Journal of Object Technology*, 18, 3, 4:1–20. DOI: 10.5381/jot.2019.18.3.a4.

[17] Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. 2020. Classifying Approaches for Constructing Single Underlying Models. en. In *Model-Driven Engineering and Software Development* (Communications in Computer and Information Science). Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić, (Eds.) Springer International Publishing, Cham, 350–375. ISBN: 978-3-030-37873-8. DOI: 10.1007/978-3-030-37873-8_15.

[18] Judith Michael and Andreas Wortmann. 2021. Towards Development Platforms for Digital Twins: A Model-Driven Low-Code Approach. en. In *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems* (IFIP Advances in Information and Communication Technology). Alexandre Dolgui, Alain Bernard, David Lemoine, Gregor von Cieminski, and David Romero, (Eds.) Springer International Publishing, Cham, 333–341. ISBN: 978-3-030-85874-2. DOI: 10.1007/978-3-030-85874-2_35.

[19] Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. 2015. XMLText: from XML schema to xtext. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2015). Association for Computing Machinery, New York, NY, USA, (Oct. 2015), 71–76. ISBN: 978-1-4503-3686-4. DOI: 10.1145/2814251.2814267.

[20] João Pacheco, Stoyan Garbatov, and Miguel Goulão. 2021. Improving Collaboration Efficiency Between UX/UI Designers and Developers in a Low-Code Platform. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. (Oct. 2021), 138–147. DOI: 10.1109/MODELS-C53483.2021.00025.

[21] Daniel Pinho, Ademar Aguiar, and Vasco Amaral. 2023. What about the usability in low-code platforms? A systematic literature review. *Journal of Computer Languages*, 74, (Jan. 2023), 101185. DOI: 10.1016/j.cola.2022.101185.

[22] Niculin Prinz, Christopher Rentrop, and Melanie Huber. 2021. Low-Code Development Platforms – A Literature Review. *AMCIS 2021 Proceedings*, 2, (Aug. 2021). https://aisel.aisnet.org/amcis2021/adv_info_systems_general_track/adv_info_systems_general_track/2.

[23] Clay Richardson, John R. Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. 2014. New Development Platforms Emerge For Customer-Facing Applications. en. Tech. rep. Forrester Research, Cambridge, MA, USA. Retrieved Dec. 15, 2023 from https://www.forrester.com/report/New-Development-Platforms-Emerge-For-CustomerFacing-Applications/RES113411.

[24] Karlis Rokis and Marite Kirikova. 2022. Challenges of Low-Code/No-Code Software Development: A Literature Review. en. In *Perspectives in Business Informatics Research* (Lecture Notes in Business Information Processing). Ērika Nazaruka, Kurt Sandkuhl, and Ulf Seigerroth, (Eds.) Springer International Publishing, Cham, 3–17. ISBN: 978-3-031-16947-2. DOI: 10.1007/978-3-031-16947-2_1.

[25] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. (Aug. 2020), 171–178. DOI: 10.1109/SEAA51224.2020.00036.

[26] Rijul Saini, Shivani Bali, and Gunter Mussbacher. 2019. Towards Web Collaborative Modelling for the User Requirements Notation Using Eclipse Che and Theia IDE. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. ISSN: 2575-4475. (May 2019), 15–18. DOI: 10.1109/MiSE.2019.00010.

[27] Rijul Saini and Gunter Mussbacher. 2021. Towards Conflict-Free Collaborative Modelling using VS Code Extensions. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. (Oct. 2021), 35–44. DOI: 10.1109/MODELS-C53483.2021.00013.

[28] Torsten Syma. 2018. *Multi-model Consistency through Transitive Combination of Binary Transformations*. Master's thesis. Karlsruher Institut für Technologie (KIT). DOI: 10.5445/IR/1000104128.

[29] 2017. Unified Modeling Language (UML) Version 2.5.1. Standard. Object Management Group (OMG), (Dec. 2017). https://www.omg.org/spec/UML/2.5.1.

[30] Burak Uyanık and Ahmet Sayar. 2024. Analysis and comparison of automatic code generation and transformation techniques on low-code platforms. In *Proceedings of the 2023 5th International Conference on Software Engineering and Development* (ICSED '23). Association for Computing Machinery, Singapore, Singapore, 17–27. ISBN: 9798400709463. DOI: 10.1145/3637792.3637795.

[31] MohammadAmin Zaheri. 2022. Towards consistency management in low-code platforms. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (MODELS '22). Association for Computing Machinery, New York, NY, USA, (Nov. 2022), 176–181. ISBN: 978-1-4503-9467-3. DOI: 10.1145/3550356.3558510.