Model Slicing on Low-code Platforms

Ilirian Ibrahimi[1,2,*], Dimitris Moudilos[2]

[1]Johannes Kepler University, Institute of Software Engineering, Altenberger Straße 69, Linz, Austria
[2]CLMS UK. Battle House,1 East Barnet Road,New Barnet, Herts EN4 8RR, UK, and Andrea Papandreo 19, Athens, Greece

Abstract

Low-code platforms (LCP) use models as the main artifact during the software development process. Typically, the modeling activity concerns both structural and behavioral aspects of the generated application, like the underlying data model (DM), User Interface (UI), and business logic (BL), resulting in a collection of interconnected models. Thus, reusing model fragments across different projects would be a highly beneficial feature for LCPs and their users.

This paper presents a model slicing approach for LCP models that combines DM, UI, and BL modeling concerns. A model slice consists of a DM class which serves as an input for the approach, its DM constraint-related classes e.g., base classes, and its related UI entities as well as BL functions.

The model slicer operates on separated model repositories which will be queried to find related entities to the DM input class and integrate them automatically into the LCP. We conducted an experimental evaluation with zAppDev models and concluded that 77.78% of the DM classes are cross-connected to any entity among the zAppDev models. Hence all these connected entities can be extracted as model slices and reused automatically.

Keywords

MDE, Low-code platfroms, Model slicing, Model reuse, Knowledge graphs

1. Introduction

Low-code platforms (LCP) are cloud-based applications that serve for building full-stack software applications without necessarily requesting coding knowledge. One of the main capabilities of an LCP is modeling the software application by designing its data models (DM), the user interface - Form models, and business logic model(s) (BL), writing as less as possible domain-specific code for implementing and deploying a complete software application.

Typical software engineering activities like coding in a general-purpose language, code formatting, modularization, database configuration, deployment, etc, are automated [1, 2]. By giving priority to modeling rather than coding, LCP enables so-called *citizen developers*, i.e., stakeholders with very limited or even no coding experience, the opportunity to create full-stack software applications which makes the LCPs more popular and useful in the software development industry [3].

LCPs leverage model-driven engineering techniques (MDE) so that models are the cornerstone artifacts that drive the overall engineering process [4, 5]. Some of these models as a whole or

Staf 2022 Workshop - 2nd International Workshop on Foundations and Practice of Visual Modeling (FVPM)
*Corresponding author.
ibrahimi.ilirian@gmail.com (I. Ibrahimi); d.moudilos@clmsuk.com (D. Moudilos)
https://github.com/iliriani (I. Ibrahimi); https://clmsuk.com/ (D. Moudilos)

part of it (i.e., their elements) might be shared among different LCP systems. Thus, finding any solution on how to reuse these models, which may be of different languages like XML, JSON, etc., and different levels like DM, UI models, etc., and providing all this information automatically to the user in the domain modeling stage would be a novel and highly on-demand task.

Hence, this paper presents an approach for model reuse through model slicing on LCP. To get the information on heterogeneous models, our approach converts all the heterogeneous models to a homogeneous graph which will serve as a knowledge graph (KG). And to cope with the different levels of models, we created two different repositories, one for the DM, and another for the Form models. The two repositories persist the KG for the DM and the Forms respectively. The model slicing approach gets as input a DM class and queries both repositories in order to get related entities to it. The required entities within the DM repository like base class, composition, etc. will constitute the horizontal slice since they belong to the same model level (i.e., DM) as the input class. And the related entities from the Form repository will constitute the vertical slice since they belong to a different level than the input class. Both slices will be merged as a single model slice and provided to the developer[1] in a JSON format.

As a proof of concept, we have tested our approach on the zAppDev[2] LCP by using 5 different zAppDev DM (in XML) with a total of 27 different domain classes, and 47 different Form models related to these domain models. The evaluation revealed that 77.78% of the given DM classes had any kind of cross-model relation i.e. we could extract successfully 21 distinct cross-language and cross-level model slices from these zAppDev models. The approach has been developed on Spring boot and is provided as a REST API.

In the rest of this paper, we present in Section 2 a running example in order to better understand the aim of the model slicing approach. In Section 3 we outline and explain how the model slicing approach works. In Section 4 we present an experimental evaluation of our approach. Afterward, in Section 5 we present some related work to model slicing, and finally, in Section 6 we provide the conclusion and the tentative future work.

2. Running Example

To clarify the concepts used throughout this paper, we will initially provide some background information.

2.1. Background Information about the zAppDev LCP

zAppDev is a web-based, model-driven development environment, allowing developers of any technology and proficiency level to easily create, edit and reuse models of software artifacts (e.g. database models, business logic models, user interface models, and more), covering the complete application development lifecycle while having total control of the process. As explained in [1] an LCP typically consists of 4 different layers which are included as well on zAppDev and are comprised of 1. The application layer is represented by the Form models, 2. The service

[1]For the sake of brevity, in this paper we will use the terms developer interchangeably for citizen developer
[2]https://zappdev.io/

integration layer by API adapters, 3. The data integration layer is represented by data models, the service models, API Adapters, and finally 4. The deployment layer is represented by the Cloud.

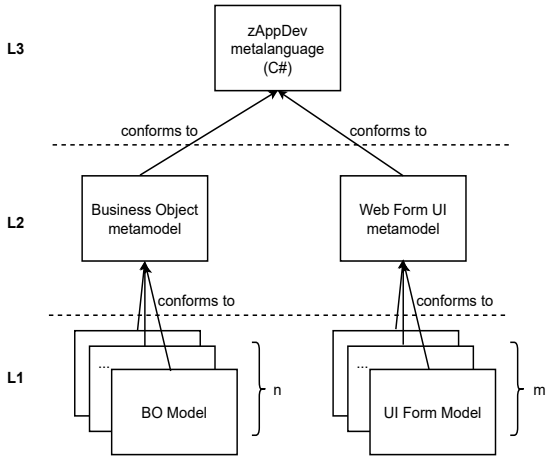

Figure 1: Metamodeling layers on the zAppDev LCP

The layers of interest for us are the application and data integration layers and their corresponding models since these are the only models the developers have direct contact with. Concretely, in this work, we will work with zAppDev data models a.k.a **business object models (BOs)**, and Form models a.k.a **UI-Form models**. The BO and UI-Form models belong to the zAppDev L1 level of the 3-level meta-modeling architecture as depicted in Fig. 1. Both, the BOs and the UI-Form models are instances of the Business Object metamodel and Web Form UI metamodel respectively (L2 level). Whereas both metamodels conform to the zAppDev metalanguage written in the c# programming language (L3 level). The metamodels and the metalanguage are embedded on zAppDev and the developers have no access to them, they start the work directly by creating BO models as instances of the BO metamodel and auto-generate the UI-Form models from the BO models or they can initially design the UI-Form model and connect it afterwards to the relevant BO model. Lastly, the developers define the business logic of any UI component within the UI-Form model by using the Mamba language.

Now, by explaining a running example we will be trying to clarify how this approach will extract model slices from LCP models.

2.2. Running example

Assume that in an LCP there is an `Invoice` software containing a BO, a UI-Form model named "Invoice Form" auto-generated from the BO classes or manually constructed by the developer, and the business logic functions related to the BO classes. The architecture of the `Invoice` software is depicted in Fig. 2. In the zAppDev LCP, the BOs are presented in XML format, the UI-Forms as JSON files, and the DSL functions are written in the Mamba[3] language. As we can see in Fig. 2 on the left part, the BO is constructed from the classes `Invoice`, `Client`, and `Company`. We aim to get only the related cross-level and cross-language artifacts to the `Client` BO class. As depicted in Fig. 2, the `Client` BO class is related to the Invoice Form with a label with the text `Client` on it and a combo box. Further, we can see that the `Client` class has an Edit function related to it. To emphasize the connection of all the `Client` related cross-level and cross-language entities we rounded and connected them with a blue cycle and blue lines.

[3]https://docs.zappdev.com/MambaLanguage/About/

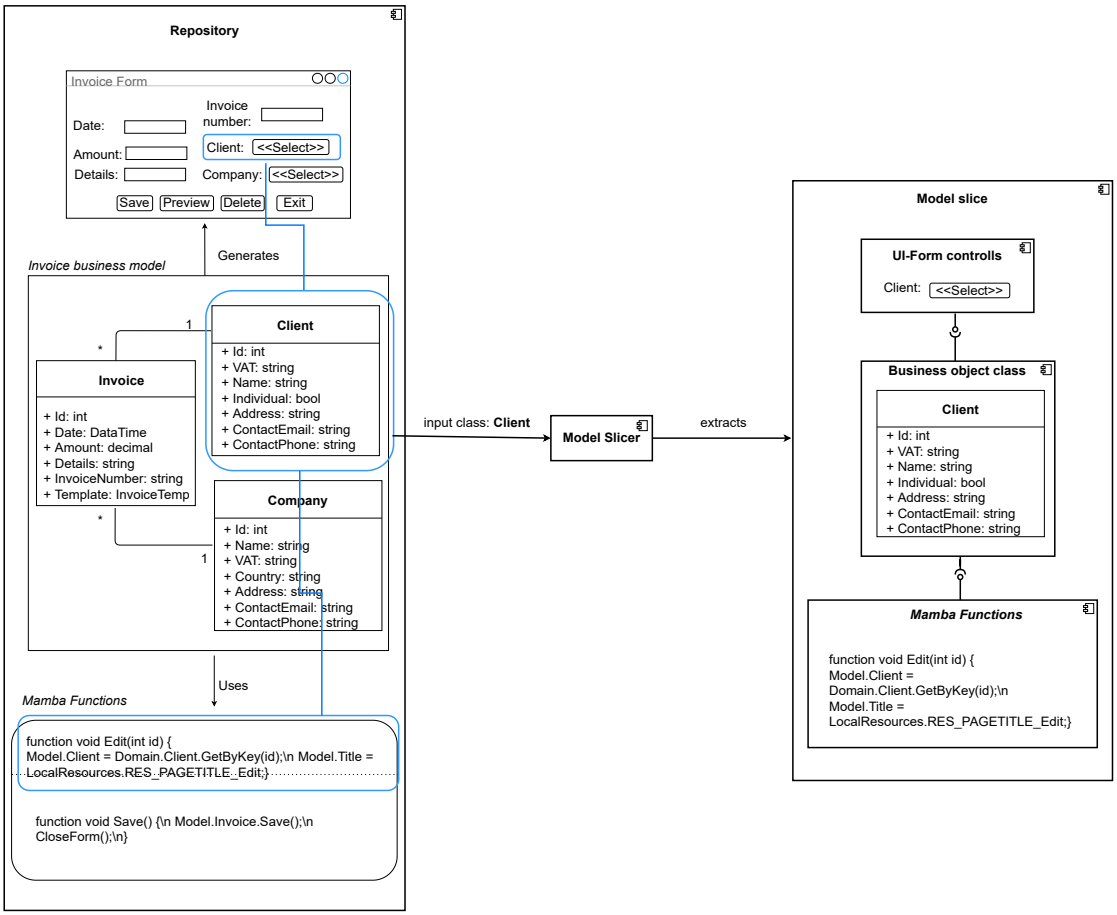

Figure 2: Model reuse through model slicing - running example

Now the idea of a model slicer approach on an LCP as shown in Fig. 2 would be to provide to the approach only the `Client` BO class as input and it would be capable to compute and extract all the cross-related entities to the `Client` BO class and integrate those on an LCP. As shown on the right side of Fig. 2 the extracted model slice from the approach is itself a model which contains only the `Client` relevant entities across the LCP models.

Concluding, the model slicer tends to find connected entities to the BO input classes on cross-level and cross-language models and presents these as a model slice to the developers so everything that is connected within the entire production line on any LCP can be reused and integrated automatically on an LCP. Inspired by this running example, we have created a model slicing approach that will be explained in more detail in Section 3.

3. Approach

This section will be presenting in more detail how the model slicer extracts model slices from cross-level models. The overview of the model slicer is presented in Fig. 3.

3.1. Repositories

The first step toward model slicing is persisting the models in a repository so they can be reused for different business needs afterwards. Since all models in zAppDev are graph-based, the repository of our approach is also graph-based. We selected the Resource Description Framework (RDF) [6] as our model format since it is a graph-based model and the standard format of W3C[4], this is relevant to LCPs which are cloud-based. Thus, various models will be converted and merged into a single RDF graph. We will use and refer to this RDF graph as the knowledge graph (KG) of our approach.

Since developing any software on an LCP ones needs to create its' DM, its' UI in a Form model, and the business logic which is persisted in any of these two (especially in zAppDev), we have created two different repositories for the model slicer, one which persists the knowledge graph for the DM, and another for the Form models. Thus, as explained in Fig. 3, *step 1* of our approach is creating the repositories which persist the knowledge graph for the DM and Form models by converting them to RDF and merging them to their respective knowledge graphs.

3.2. Input Class

Step 2 of our approach is getting the input class. The input class is the core entity of any model slice because any other entity of the model slice has to be related in a specific form to it. Hence, the input class defines the slicing criterion for our approach. After having the input class, the slice service - which is responsible for the business logic part of the model slicer - will query the repositories to find relevant connections between the existing entities within the repositories and the input class.

3.3. Horizontal Slice

The first check the input class will go through is if it has any **constraints class** that has to be integrated with the input class. For instance, if the input class inherits a class within the BO classes, or has a composition class, then the base/composition class has to be integrated as well in the LCP in order to avoid model validation errors. Thus in *step 3*, our approach checks in the DT (BO) repository if there is any such constraints class related to the input class, and if any such class can be found it will be provided to the developers together with the input class. Since the input class and the constraint classes are on the same level (within the business object) we call this kind of relation as horizontal slicing. Since we are slicing only zAppDev models so far, the horizontal slice is defined only by checking if the input class has any base class within the BO repository.

3.4. Vertical Slice

Next, the model slicer service will check if there is any related entity within the Form KG to the input class. In step 4 the model slicer will return all the relevant information about the related entities to the input class. In our case, we query the information about UI components, i.e., the UI component name, the UI component data source - which shows to which specific

[4]https://www.w3.org/TR/2004/REC-rdf-concepts-20040210

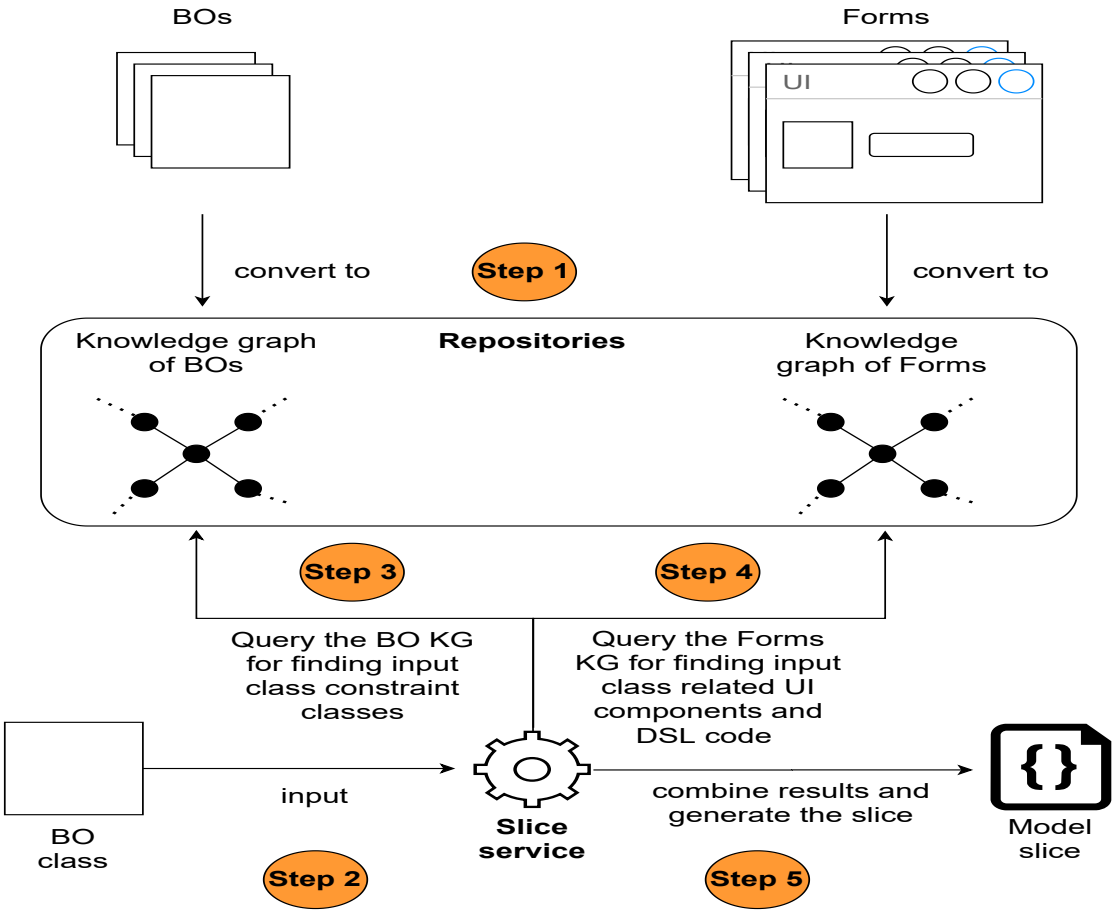

Figure 3: Model slicing approach overview

attribute of the input class the UI component is related, and the type of the UI component, e.g. TextBoxControll, CheckBox, etc. Although a lot of other relevant information can be retrieved e.g. the cascade style sheet (CSS) information about the UI components, UI components layout information, etc. Since in zAppDev the DSL functions a.k.a. Mamba functions are persisted within the Form models, in step 4 the model slicer also queries for related Mamba functions to the input class. Hence the UI components and DSL functions are not in the same model level as the input class, we call this relation of connected cross-level models a vertical slice.

Finally, the extracted horizontal and vertical slices will be merged as a single model slice and integrated into the LCP.

An example of model slicing is presented in Fig. 4. We see that the meta-class `Class 1` is connected to the UI components `Comp. 1` and `Comp. 2` and also to the DSL functions *Func. 1* and *Func. 2*, thus the connection of all these entities would give a single slice (Slice 1). Further in Fig. 4, we can see that the meta-class `Class 3` has a constraint class `Class 2` within the BO, it is also related to the component `Comp 3` on the UI - Form model, and it has also a related DSL function *Func. 3*. The connection of these related entities to `Class 2` would produce another

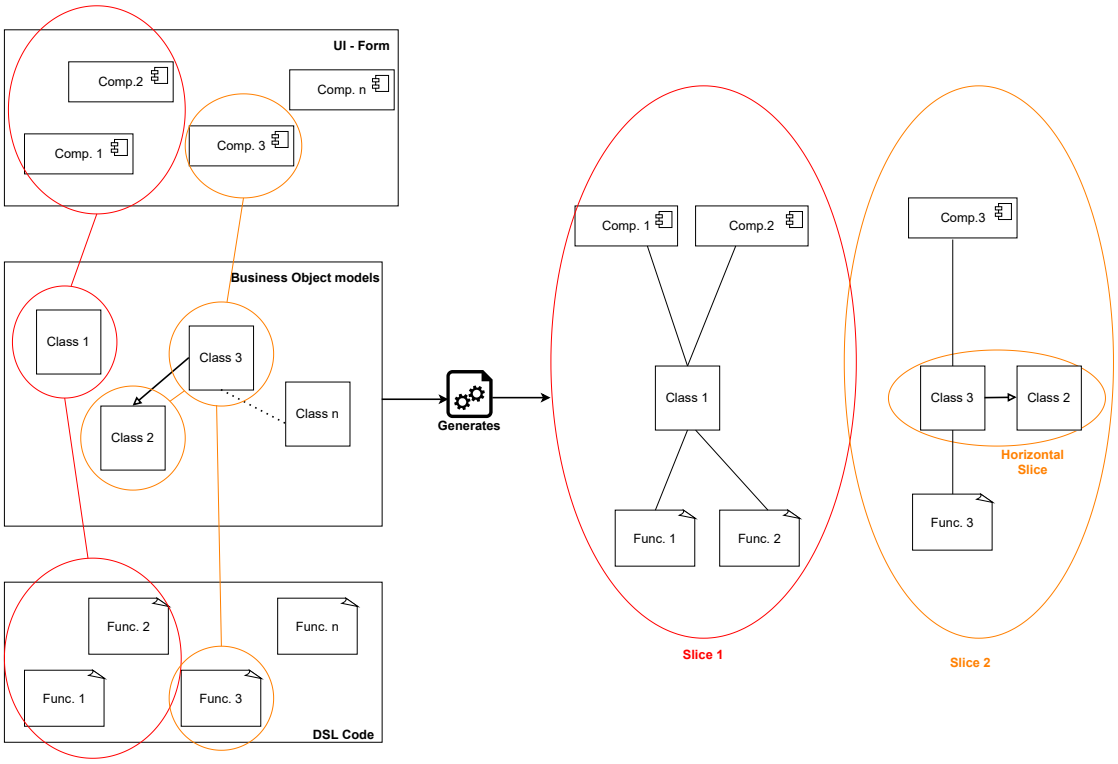

Figure 4: Model reuse through model slicing

Model slices on zAppDev models				
BO models	BO classes	UI-Form models	Horizontal Slices	Vertical Slices
1 CoreBO	10		0	9
2 DTOs	2		0	2
3 Expenses	5	47	0	4
4 ProjectBO	3		0	3
5 TaskBO	17		1	8
Total	48 (27 Distinct)	47	1	26 (21 Distinct)

Table 1
Model slices on zAppDev models

modeling slice (Slice 2).

4. Experimental evaluation

This section demonstrates how the model slicer extracts model slices on real LCP models.

As a proof of concept, we got 5 different BOs and 47 different UI-Form models generated by these business objects. We have created two different knowledge graphs which contain all

the information about the BOs and the Form models respectively. Then we selected each class iteratively from the BOs and gave this as an input class to the model slicer and checked the extracted model slices. In Fig. 5 we have presented the information that will be extracted by the model slicer. In this demo, the model slicer will try to extract a model slice related to the `Client` class.

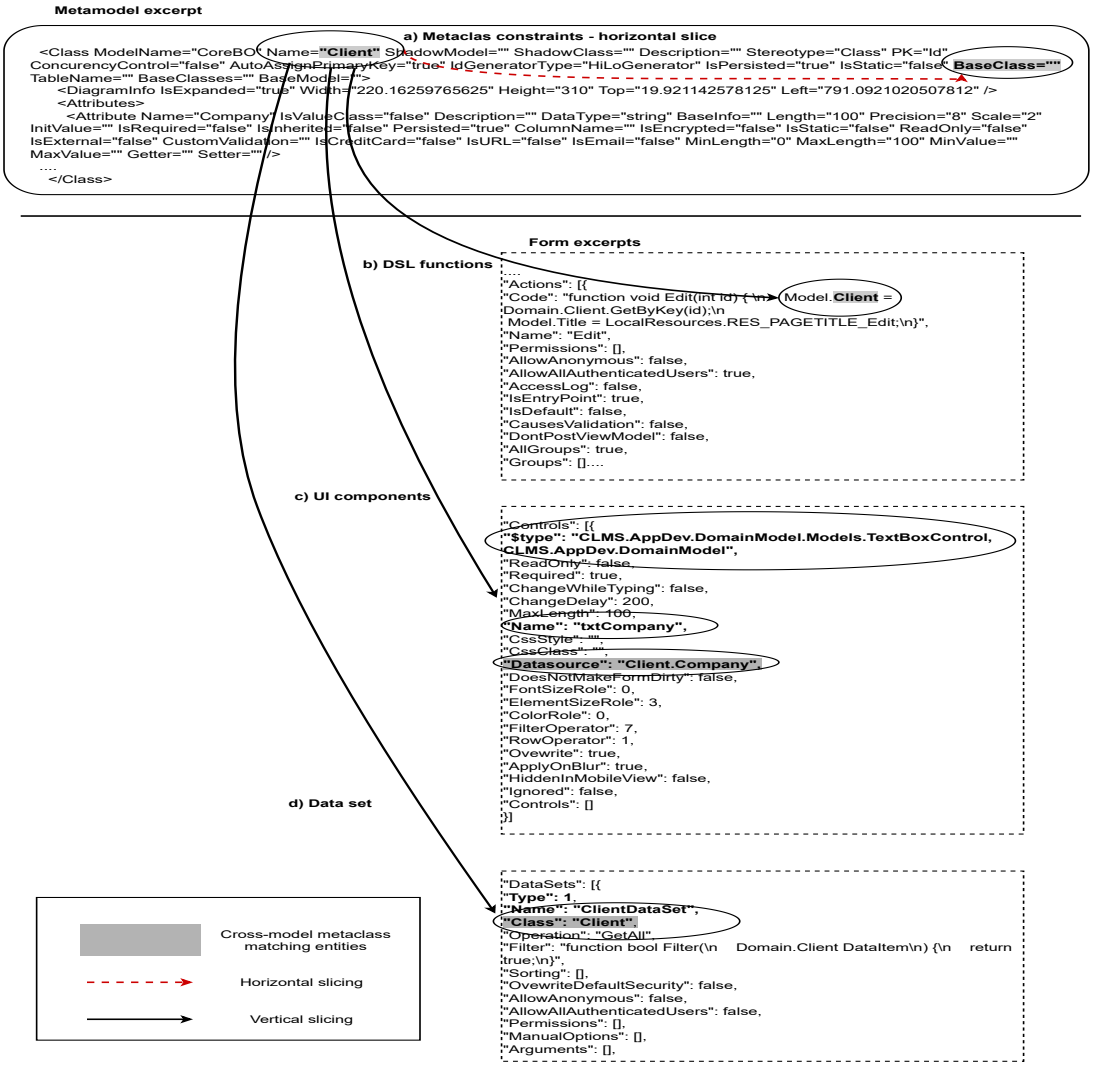

Figure 5: Model slicing in zAppDev

Initially, the model slicer will check within the BOs at the *baseClass* element to find any base class so it can create the horizontal slice (a)). This check is presented with the red dashed arrow. In this demo, the `Client` class hasn't any base class, and since at the time of writing this paper this is the only checked constraint for BO classes, will the model slicer not define any horizontal slice. Next, the model slicer will check for extracting the relevant Mamba functions from the zAppDev Form models (b)). The model slicer will check within the *Code* notation to find any related

function to the `Client` class. The found functions will be returned as part of the vertical slice.

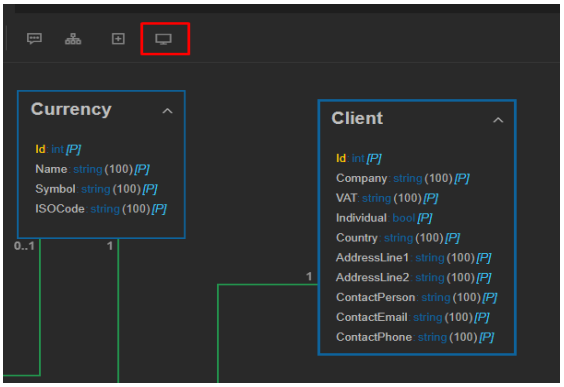

Figure 6: Triggering the model slicer service for the Client class

Next in c), the model slicer will extract the required information about the relevant UI components to the `Client` class. First, the model slicer will check if there is any *Datasource* notation that is related to the `Client` class, if there is any, then it gets the information about that Datasource related *Name* and *$type* notation. These three notations: *Name, Datasource,* and *$type* will be returned also as part of the vertical slice.

Finally, the model slicer checks for any related *dataset* to the `Client` class (d)). It checks the notation *Class* within the *Dataset* notation if it is Client. If there is a match, then will the model slicer get the respective *Name, Operation,* and *Filter* information. All this information will also be returned as part of the vertical slice.

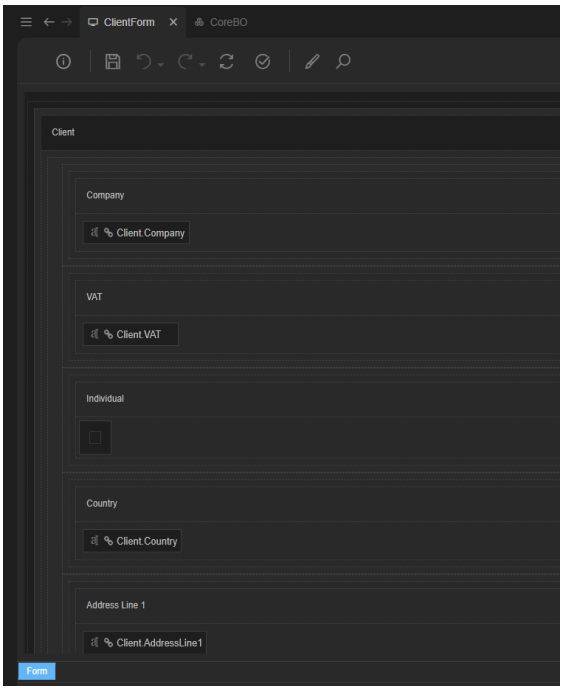

Figure 7: Integration of the Client model slice in zAppDev

In the end, all the information about the horizontal and vertical slices will be merged in a single JSON file and integrated into the zAppDev LCP. This model slice is extracted after selecting the `Client` BO class and clicking the ”Create forms from Slice” button located in the menu bar as shown in Fig. 6. The model slice is integrated on the zAppDev LCP as a UI-Form model including the extracted information for the `Client` BO class. A snapshot of the integrated model slice on zAppDev is depicted in Fig. 7.

In Table 1 we have outlined the results of how many model slices could be extracted from the zAppDev models. For the study, we have used 5 different business object models and 47 different UI-Form models. We listed all the BO classes from all the 5 BO models and counted 27 distinct classes. We iterated through each of these BO classes and set each of them successively as an input class to the model slicer. Of all these BO classes only one class had a base class (PMOUser had as base class ApplicationUser) i.e., a horizontal slice. Also from the set of 27 distinct BO classes, the model slicer could extract 21 vertical slices which

means that 21 BO classes have at least one related entity on the UI-Form models. From this amount of data we got from zAppDev, we could conclude that **77.78%** of the BO classes are related at least to a base class or at least to one UI-Form model entity. This fact reveals the emerging need for a cross-language and cross-level model reuse approach on LCP that can be facilitated through the model slicer provided in this work.

The model slicer has been developed using Spring boot and is provided as a REST API for use in the zAppDev LCP. The source and the repositories containing the KG used for the evaluation are available on GitHub[5]

5. Related Work

Although to the best of our knowledge this is the first approach towards model ruse through slicing on cross-level and cross-language LCP models, inspired by program slicing approaches [7, 8], we will show some related works to model slicing.

Salay et al. [9] present an algorithm for megamodels slicing. The algorithm gets as input the megamodel and by using the traceability relation among the entities of the models that construct the megamodel it extracts the model slice. Our approach is search-based and not static based, i.e., it doesn't iterate through the cross-level model entities of a megamodel, it queries different repositories to find relevant matches to the input class.

Taenzer et al. [10] present a formal framework for creating model slicers that are capable to change incrementally a model slice after performing any change on it. Our approach is search-based and generates model slices from a single input class to support the LCP users during the modeling process. It is not required that we implement the update of model slicing since it can be updated directly by the LCP users based on their needs after being integrated.

Compare to the approaches that enable model slicing for a specific model type [11, 12, 13, 14, 15, 16] our approach checks for related entities among different models of different types and extract them as a model slice.

6. Conclusion and Future work

In this work, we have presented an approach that enables the reuse of cross-related models on an LCP through model slicing. The model slicing approach gets as input a data model class and queries the data model for any constraint-related class, and also the Form model repositories to get UI components and DSL functions. The current approach enables model slicing of zAppDev models but conceptually it can be used for any LCP.

In future work, we plan to fine-grain the model slicer by proving UI component layout information, slicing the class attributes, etc. We also aim to integrate the model slicer on a model recommendation approach so that after selecting a suggested data model class, all its cross-related model entities will be integrated automatically.

[5]https://github.com/iliriani/Model-slicer

Acknowledgments

This project has received funding from the EU Horizon 2020 research and innovation programme under the Marie Skłodowska Curie grant agreement No 813884.

References

[1] A. Sahay, A. Indamutsa, D. Di Ruscio, A. Pierantonio, Supporting the understanding and comparison of low-code development platforms, Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020 (2020) 171–178. doi:10.1109/SEAA51224.2020.00036.

[2] R. Waszkowski, Low-code platform for automating business processes in manufacturing, IFAC-PapersOnLine 52 (2019) 376–381. URL: https://doi.org/10.1016/j.ifacol.2019.10.060. doi:10.1016/j.ifacol.2019.10.060.

[3] P. Vincent, K. Iijima, M. Driver, J. Wong, Y. Natis, Licensed for Distribution Magic Quadrant for Enterprise Low-Code Application Platforms (2019) 1–34. URL: https://www.gartner.com/doc/reprints?id=1-1ODOM46A{&}ct=190812{&}st=sb.

[4] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, M. Wimmer, Low-code development and model-driven engineering: Two sides of the same coin?, Software and Systems Modeling (2022). URL: https://doi.org/10.1007/s10270-021-00970-2. doi:10.1007/s10270-021-00970-2.

[5] A. Bucaioni, A. Cicchetti, F. Ciccozzi, Modelling in low-code development: a multi-vocal systematic review, Software and Systems Modeling (2022). URL: https://doi.org/10.1007/s10270-021-00964-0. doi:10.1007/s10270-021-00964-0.

[6] O. Lassila, R. R. Swick, Resource description framework (RDF) model and syntax specification. World Wide Web Consortium Recommendation (1999). URL: http://www.w3.org/TR/REC-rdf-syntax.

[7] H. V. Nguyen, C. Kästner, T. N. Nguyen, Cross-language program slicing for dynamic web applications, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, 2015, p. 369–380. URL: https://doi.org/10.1145/2786805.2786872. doi:10.1145/2786805.2786872.

[8] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, IEEE Press, 1981, p. 439–449.

[9] R. Salay, S. Kokaly, M. Chechik, T. S. E. Maibaum, Heterogeneous megamodel slicing for model evolution, in: ME@MoDELS, 2016.

[10] G. Taentzer, T. Kehrer, C. Pietsch, U. Kelter, A formal framework for incremental model slicing, in: FASE, 2018.

[11] R. Ahmadi, J. Dingel, E. Posse, Slicing uml-based models of real-time embedded systems, 2018. doi:10.1145/3239372.3239407.

[12] S. Sen, N. Moha, B. Baudry, J.-M. Jézéquel, Meta-model Pruning, in: ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Denver, Colorado, USA, United States, 2009. URL: https://hal.inria.fr/inria-00468514.

[13] A. Bergmayr, M. Wimmer, W. Retschitzegger, U. Zdun, Taking the pick out of the bunch - type-safe shrinking of metamodels, in: S. Kowalewski, B. Rumpe (Eds.), Software Engineering 2013, Gesellschaft für Informatik e.V., Bonn, 2013, pp. 85–98.

[14] H. Kagdi, J. I. Maletic, A. Sutton, Context-free slicing of uml class models, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM '05, IEEE Computer Society, USA, 2005, p. 635–638. URL: https://doi.org/10.1109/ICSM.2005.34. doi:10.1109/ICSM.2005.34.

[15] P. Kelsen, Q. Ma, C. Glodt, Models within models: Taming model complexity using the sub-model lattice, in: D. Giannakopoulou, F. Orejas (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 171–185.

[16] A. Blouin, B. Combemale, B. Baudry, O. Beaudoux, Modeling model slicers, in: J. Whittle, T. Clark, T. Kühne (Eds.), Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 62–76.