A Low-Code Approach for Data View Extraction from Engineering Models with GraphQL

István Koren
Process and Data Science,
RWTH Aachen University
koren@pads.rwth-aachen.de

Nico Jansen
Software Engineering,
RWTH Aachen University
jansen@se-rwth.de

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

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

Enno Böse
RWTH Aachen University
enno.boese@rwth-aachen.de

***Abstract*—Access to data for analysis and control tasks is at the heart of digitization efforts in the manufacturing industry. While sophisticated modeling languages like SysML describe systems and their components, data often ends up in purpose-built relational and time series databases. To generate value, information must be retrieved and integrated from multiple sources. In this paper, we propose an innovative method for leveraging SysML engineering models and database queries by combining them in a collaborative low-code web environment. First, we make heterogeneous databases available via GraphQL, a state-of-the-art approach for building Web APIs. Then, our web application enables domain experts to exploit containment relations in SysML models to connect diverse data sources. The outcome is an integrated GraphQL API that matches the engineering model structures by resembling views across multiple database sources. The discussed approach incorporates the benefits of data-oriented development and low-code platforms beyond the business automation domain.**

***Index Terms*—Low-Code, Data Integration, Data-Driven Applications, Manufacturing, Industry 4.0, Model-Driven Software Engineering, SysML, Engineering Models**

I. Introduction

The manufacturing industry is investing considerable resources in the digitization of its processes as a means to boost productivity [1]. The prerequisite to new kinds of analysis tools driven by machine learning is the availability of data from all levels of production, starting with machines and ranging all the way to order and logistics data. In reality, however, a great diversity of data sources and sinks can be observed [2]. For example, even data from the same machine is spread across different databases that were specifically built for particular purposes. Across domains, such as engineering, production, and usage, diversity grows. For a comprehensive analysis, cross-domain data needs to be integrated.

The Systems Modeling Language (SysML) [3] is a general-purpose modeling language for designing the structure, behavior, requirements, and parametrics of cyber-physical systems. It is adopted in various domains [4].

While SysML supports the systems engineering process by leveraging models instead of a traditional document-based approach, integration with heterogeneous runtime data is often missing. Thus, there is a gap between the models describing a system and the (often unstructured) data collected during runtime. However, sophisticated, highly data-driven analyses and applications require structured data preprocessing. Therefore, combining these models, explicitly representing the systems' (sub-) hierarchies, with data queries can set the foundation for an integrative view. Such a technique would enable stakeholders as domain experts without advanced software development knowledge to model schema integration tasks tailored to their specific needs, avoiding the expense of hiring software specialists. Therefore, we are interested in the research questions, how data mapping tasks can be facilitated for domain experts (RQ1) and whether (GraphQL) APIs can be derived from SysML models (RQ2).

In this paper, we present a low-code approach and tool support that enables linking engineering models with the data the modeled systems produce. The aim is to realize database views as object-oriented GraphQL queries. Our approach combines Model-Driven Software Engineering (MDSE) [5] methods with user-friendly functionalities of collaborative low-code web applications, to account for domain experts with varying data literacy. We extend the low-code platform Direwolf Modeler [6] with SysML model support and develop a service that converts SysML models to working GraphQL services using the MontiCore language workbench [7]. The proposed solution demonstrates the feasibility and usability of the model-driven approach for data integration tasks but also highlights challenges of possible real-world adoptions, particularly regarding security.

The outline of this paper is as follows. In Section II, we present related work in the area of data integration and low-code model-driven design. We discuss the built-upon technologies in Section III. In Section IV, we outline our approach and technical realization. Section V concludes this paper.

II. Related Work

Easy availability and integration of data has multiple benefits in the context of production processes [8]. However, to analyze and use the data, uniform storage and access to data must be ensured. *Ontology-based data access* (OBDA) [9] and *Database Federation* [10] are research domains that deal with integration tasks. Database federation refers to an architecture in which a middleware provides uniform access to heterogeneous data sources. Applications and developers might access this data by a single query that gets processed by the management system and relayed to the underlying sources. The same architecture is implemented in newer database management systems in conjunction with GraphQL. For example, the open-source tool *Hasura*[1] is able to connect various relational and non-relational databases in a unified API.

OBDA handles the mapping of objects, representing the high-level abstraction of a domain of interest, and the data sources. Clients express queries in terms of an ontology, rather than a database schema, and the OBDA system translates them and enables access [9]. Typically, a mapping language specified by domain experts and database engineers is used.

In this paper, we are working on the synergy between low-code development approaches [11] and MDSE [12]. This overlap shares the use of models and the aim to reduce the amount of source code required to create a software system [11]. The automated transformation of models into software implementations opens the possibility to integrate information from other formal descriptions, e.g., engineering models, into the software. Several MDSE and low-code approaches for production systems exist, however, they do not cover the integration of data from different data sources in combination with engineering models. For instance, one MDSE approach for digital twins acts upon data generated from cyber-physical production systems [13]. Other investigations describe the generation of digital twin cockpits from event logs [14], or from data structure models [15] supporting parameter management in the engineering process of wind turbines. A further prominent example comes in the context of the EU TYPHON project[2] dealing with heterogeneous database infrastructures featuring high scalability. It proposes a model-based strategy incorporating multiple languages to design, deploy, as well as evolve distributed data stores and derive target store-specific data access queries from a high-level representation [16]. However, these approaches rely on explicitly crafted data structure models rather than automatically combining their data and data sources with existing engineering models.

Most commercial LCDPs are proprietary and closed-source (e.g., *Microsoft PowerApps*[3] and *Make*[4]), which hinders interoperability, extensibility, and reusability [17], [18]. We develop our tool with open standards like SysML and GraphQL to consider these aspects explicitly. Since the cooperation between domain experts from engineering and data experts from computer science is at the heart of our approach, we aim for a web-based collaborative modeling tool [19] that is easily accessible by all stakeholders.

[1]https://hasura.io/
[2]https://www.typhon-project.org/
[3]https://powerapps.microsoft.com/
[4]https://www.make.com

III. Background

The main idea behind our approach is to reuse engineering models to generate views on data stores. In the following, we first introduce the underlying languages and tools, namely SysML block diagrams that describes the structure of systems. We then show GraphQL as an object-oriented and hierarchical query language, the Direwolf Modeler, and the MontiCore language workbench.

Block Definition Diagram (BDDs) as specified as part of the SysML 1.6 specification [3] describe system structures. A block is a modular unit describing certain system parts and their relations. A block can comprise a real-world system, such as a car or a wheel, but also more abstract concepts, such as a production process or the automotive domain (driver, passenger, car, baggage, etc.). An Internal Block Diagram (IBD) captures the internal structure of a block. BDDs define multiple relations that can be used for different purposes: Generalization, part association, shared association, reference association, as well as multi-branch versions of these elements.

GraphQL [20] is a query language for APIs and a server-side runtime for executing queries using a defined type system. A GraphQL data schema document contains multiple definitions that are either executable or represent a GraphQL type system. A service is created by defining types with their fields, and operations for each field on each type. The operation is either a *query*, a *mutation*, or a *subscription*. An exemplary GraphQL API of a service providing general information of a machine could have a root `Query` type with a field named `machines` that returns a list of machine objects with fields for the machine's id, name, and location. To execute a GraphQL query, the client sends the query string and the operation (query or mutation) to the server. The server then executes the query and returns the requested data. GraphQL is a hierarchical query language, which means that the data is queried from the root to the leaves of the graph.

There is a large number of tools, frameworks and libraries available for GraphQL. The commercial open source tool Hasura offers instant real-time GraphQL APIs on existing PostgreSQL and other databases. It connects to existing databases, and automatically generates a GraphQL API. Hasura provides a built-in GraphQL engine, which allows to write custom GraphQL queries and mutations. It provides functionality for database federation with *Remote Schemas* but lacks a built-in way to link different database sources together in such a way as underlying engineering models prescribe.

Direwolf Modeler is a modular low-code framework for creating universal graph-based graphical modeling applications [6], with an emphasis on visual development, simplicity, and accessibility. The framework supports various node- and edge-based metamodels, making it extensible for new graph-like modeling languages. It enables the simultaneous integration of multiple modeler instances and, as a collaborative tool, allows stakeholders to work together in the browser. The user interface is kept deliberately simple and offers drag-and-drop functionalities of modeling elements from an element palette.

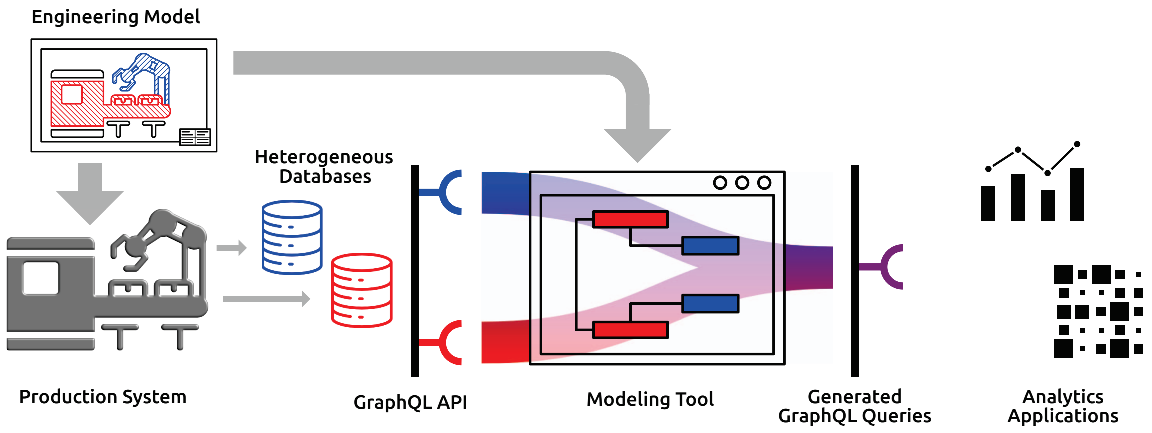

Figure 1: From engineering model to generated GraphQL API.

MontiCore is a language workbench for the development of Domain-Specific Languages (DSLs) [7]. DSLs are defined via context-free grammars, from which MontiCore generates software components for model processing such as a parser, transforming models into an Abstract Syntax Tree (AST), well-formednes and visitor infrastructures [21], and a symbol table for models [22]. Additionally, it provides for template-based code generation that builds upon the Apache FreeMarker[5] template engine. *Visitors* allow traversing and operating on the AST data structure of the model. In the generation process, parameterized templates are converted into programming language code.

IV. From SysML to Generated GraphQL Schema

Our general concept is based on the observation that some information about structures, which is necessary for further processing heterogeneous data from different databases, is available in the engineering models. Therefore, the goal of our work is the automated elicitation of this information and the derivation of GraphQL queries for database federation and further processing of this now structured data. Figure 1 explains our approach. Production systems (bottom left), which are described by engineering models (top left), produce data that are written to various databases. The model features an industrial conveyor belt (red) with a gripper arm (blue) that sorts out faulty parts. Belt and gripper store data at different frequencies to different databases. The general goal of this work is to reuse these engineering models in a low-code environment and integrate access with the heterogeneous databases via the GraphQL API (mid). The tooling automatically derives queries to provide the corresponding data in a structured way for integrated analyses (right).

Transformation Approach: In this work, we mainly refer to engineering models of SysML. More precisely, we concentrate on SysML BDDs, describing system structures, and forming the modeling baseline of our approach. We build on the modeled system components and their relationships for extracting linking information of the underlying runtime data of a corresponding machine. For instance, an association between two blocks represents a data relation between the value properties, i.e., the data sources. In our modeling tool, domain experts establish the connection between SysML BDD blocks and GraphQL types by providing tagging information. Thus, each block in the model is augmented with additional data source details, allowing for different data sources for distinct elements. For each property and for the block name, a tag specifies a source (i.e., a table in one of the incorporated databases). The tables are then joined via the information provided by the association. We define two properties for each association, `joinFrom` and `joinTo`. A domain expert defines how data is joined together, similar to how foreign key relations in a traditional SQL database are designed. Then, the final (integrated) GraphQL schema is generated. For instance, a table *resource* could contain all machine data, i.e., the name and cost per hour of all machines. The root query name for this table is then "resource", while the machine data are the fields of that query. An example GraphQL query would thus be `{resource {name cost_per_hour}}`.

Deriving GraphQL schemas from the augmented SysML BDDs requires an unambiguous mapping of the individual diagram elements. Table I provides an overview of this *translation* between concepts of *SysML BDDs into GraphQL*. Every SysML block b generates a new GraphQL schema type with the name of the block (row 0 in Table I). Analogously, every interface and enumeration has a corresponding interface or enumeration (rows 1 & 5). If a block is associated with an interface, then the GraphQL type also implements the interface (row 2). For every attribute, or association with cardinality 1, we create a new GraphQL field with the name and type of the attribute, or the object type in case of an association (row 3). For associations with cardinality higher than 1, we generate a field with the association name and the type wrapped in square brackets (row 4), indicating a collection-like access schema. An enumeration literal is converted into a corresponding GraphQL enumeration literal pendant (row 6).

[5]https://freemarker.apache.org/

We create a new GraphQL scalar type if we encounter a field type that is not a scalar type but should be one (row 7). For associations that expect at least one result, we add an exclamation mark representing required parameters (row 8). Furthermore, BDDs support multiple other association types that we do not specify explicitly. For example, generalization can be included by adding the properties of the generalized block to the specialized block. Similarly, the part association can be included by specifying that all properties of a part are inserted into the block it is part of. No separate type should be created for a part either. As SysML originated as a UML profile, this method directly resembles related work that maps UML and IFML models to GraphQL [23].

Table I: Mapping from SysML BDD elements to GraphQL schema components.

#	BDD Metaclass	GraphQL schema
0	b : Block	type b.name {...}
1	i : Interface	interface i.name {...}
2	b : Block.implements → i : «Interface »	type c.name implements i.name {...}
3	a : {attrs., assocs. max. mult. 1}	a.name : a.type (field)
4	as : assocs. max. mult.>1	as.name : [as.type] (field)
5	e : «Enumeration »	enum e.name {...}
6	el : Enum Literal	el.name (enum value)
7	t : Type not in GraphQL scalar types	scalar t.name
8	f : structural feature min mult. 1	f.name : f.type! (type marker)

Technical Realization: We use the open-source tool Hasura to make data sources available as GraphQL APIs. It offers data integration functionalities for heterogeneous data sources via remote schemas that need to be configured via textual config files. Based on the integrated APIs, our low-code tool allows for a demand-driven composition of queries, automatically derived by extracting the interrelations within the SysML models. Using the customization possibilities of the Direwolf Modeler, we realized an extension incorporating SysML BDDs with augmentation possibilities via attachable tags. Thus, each block includes a `title`, an `attributes` property containing all fields of a block, and a `tags` property containing all the additional tags that were added to an element. Users can add attributes by specifying a string that consists of the name of the attribute, followed by a colon and lastly the type of the attribute (e.g., `name: String`). Furthermore, associations are available in the BDD modeling palette that can be applied by dragging from one block to another. Each association has attributes, tags, and the properties `cardinality_origin`, `cardinality_target`, `JoinFrom`, and `JoinTo`. The cardinality properties express the relationship's multiplicities (at both ends) between blocks based on the direction of the edge. In our model, associations are always directed, to get unambiguous schema mappings with corresponding *JoinFrom* and *JoinTo* properties. Therefore, to create reciprocal references, we require two separate associations. To further process the data access enriched models, we use Direwolf's capability to convert different kinds of models into a JSON representation that we employ in the next step.

Extracting and Processing Model Information Generating GraphQL schemas and servers from the augmented BDDs modeled in Direwolf requires further processing of the models, exported as JSON artifacts. We use the MontiCore JSON parser to handle these JSON representations and create an AST. The parser dissects the JSON document into a data structure adhering to the JSON grammar representation so that each entry has a corresponding type, such as `JSONArray`, `JSONProperty`, or `JSONObject`.

After deriving the AST, we need to collect the information relevant to the generation process. For this purpose, we employ a visitor, which MontiCore automatically provides for each language [7], an extended realization of the general visitor pattern [24]. By design of Direwolfs export, we have two arrays within the input JSON document, one for all nodes and one for all edges of the graph induced by the original SysML model. Here, we make use of MontiCore's possibility to customize the overall traversal strategy to only shallowly traverse the AST as we are only interested in the array's nodes and edges. Thus, the visitor operates on the nodes of the JSON AST and collects a list of Java `node` objects and the edges into a list of `edge` objects. The visitor returns the sub-AST that results from its execution. On the node array, we collect the properties of that node into the `node` object. On the edge array, we proceed analogously.

GraphQL Schema Generation. To generate the GraphQL schema, based on the extracted features from the JSON AST, we add fields as attributes to each node if it has an outgoing edge to another node, as shown in Table I. The name of this field is the association name and the type is the name of the target node. Furthermore, we annotate the new field with square brackets if the cardinality for the target is greater than one. Additionally, a root query is created for every node as the access point for the respective data source. A *query* defines the entry point for all queries of the generated GraphQL schema. Instead, the *root queries* are generated for every node to query all fields (including associations) from one data source.

GraphQL Server Generation. We generate a GraphQL server in JavaScript that runs on the open-source *Apollo Server*[6]. We generate an `index.js`, and, for every GraphQL type, `resolver.js` and `datasource.js` files. A *datasource* class encapsulates fetching data from particular data sources via HTTP. Functionalities of datasource files are available to the *resolver* as APIs that have the node's name. The topmost resolvers send API calls to all datasources. These calls are cached for scalability reasons. The `JoinFrom` and `JoinTo` properties of the nodes are used to match the result tables with the respecting fields of our GraphQL type system. Finally, the index file contains functions to import the corresponding datasource files, create an Apollo instance, instantiating the resolvers, and starting the server.

V. Conclusion and Future Work

The digitization of industry relies on making large amounts of data available for advanced analysis and control tasks. However, these data treasures often sit in heterogeneous

[6]https://www.apollographql.com/docs/apollo-server/

databases that have been set up for specific purposes, making integration difficult. For instance, *Digital Shadows*, which form a purposeful view of an observed object or process, require a specific focus and by selection and aggregation of data that may originate from heterogeneous sources [25]. Our proposed method of combining SysML engineering models and database queries in a collaborative web environment relies on domain experts working together with data experts. We benefit from parts-of hierarchies inherent to SysML models, resulting in an integrated GraphQL schema matching the engineering model structures. By extending Direwolf Modeler with SysML BDDs, we facilitate data mapping tasks for domain experts, thus answering RQ1. Furthermore, we developed a prototype for generating a GraphQL server with the MontiCore language workbench and its text generating capabilities for creating GraphQL schemas, answering RQ2. In comparison to proprietary LCDPs, every part of our software can be swapped for another technology stack. A preliminary technical evaluation exhibits a low overhead with a linear increase of the response time with each added node. The generation time is almost unaffected by the number of nodes.

Our prototype shows a number of limitations. E.g, the generated resolver exhibits overfetching issues, i.e., data rows are fetched multiple times, which becomes critical with an increasing number of source tables. Real-world aspects that we considered out of scope include the integration of more complex and real-time time series data sources. Database federation, in general, has the risk of compromising huge amounts of data if a security vulnerability is exploited.

In future work, we want to research how additional GraphQL operation types, such as mutations and subscriptions, could be combined with our solution. Moreover, our approach can be used to connect digital twins with the different data sources from their cyber-physical counterpart in low code development platforms [26]. Our tool connects modeling and domain experts in a visual, collaborative interface requiring a usability study. To test our prototype in real-world applications, we are working on scalability measures by employing caching and batching of queries. Finally, usability evaluations with domain experts need to be performed. Overall, we see great potential for collaborative LCDPs involving both modeling and domain experts to tackle data integration and analytics tasks common to digitization endeavours.

ACKNOWLEDGEMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de

REFERENCES

[1] P. Brauner *et al.*, "A Computer Science Perspective on Digital Transformation in Production," *ACM Trans. Internet Things*, vol. 3, 2022.
[2] F. Tao, Q. Qi, A. Liu, and A. Kusiak, "Data-driven smart manufacturing," *Journal of Manufacturing Systems*, vol. 48, pp. 157–169, 2018.
[3] Object Management Group, "OMG Systems Modeling Language (OMG SysML) Version 1.6," Standard, 2019. [Online]. Available: https://www.omg.org/spec/SysML/1.6/
[4] I. Drave *et al.*, "Modeling Mechanical Functional Architectures in SysML," in *Int. Conf. on Model Driven Eng. Languages and Syst.* ACM, 2020.
[5] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering (FOSE '07)*, pp. 37–54, May 2007.
[6] I. Koren, R. Klamma, and M. Jarke, "Direwolf Model Academy: An Extensible Collaborative Modeling Framework on the Web," in *Companion Proc. of Modellierung 2020*, vol. 2542. CEUR-WS.org, 2020, pp. 213–216.
[7] K. Hölldobler, O. Kautz, and B. Rumpe, *MontiCore Language Workbench and Library Handbook: Edition 2021*, ser. Aachener Informatik-Berichte 48. Shaker, 2021.
[8] L. Gleim *et al.*, "FactDAG: Formalizing Data Interoperability in an Internet of Production," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3243–3253, 2020.
[9] A. Poggi *et al.*, "Linking Data to Ontologies," in *Journal on Data Semantics X*, ser. LNCS. Springer, 2008, vol. 4900.
[10] L. M. Haas, E. T. Lin, and M. A. Roth, "Data integration through database federation," *IBM Systems Journal*, vol. 41, no. 4, pp. 578–596, 2002.
[11] D. Di Ruscio *et al.*, "Low-code development and model-driven engineering: Two sides of the same coin?" *Software and Systems Modeling*, 2022.
[12] S. Sendall and W. Kozaczynski, "Model Transformation: the Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.
[13] P. Bibow et. al., "Model-Driven Development of a Digital Twin for Injection Molding," in *Int. Conf. on Advanced Information Systems Engineering (CAiSE'20)*, ser. LNCS. Springer, 2020.
[14] D. Bano, J. Michael, B. Rumpe, S. Varga, and M. Weske, "Process-Aware Digital Twin Cockpit Synthesis from Event Logs," *Journal of Computer Languages (COLA)*, vol. 70, June 2022.
[15] J. Michael, I. Nachmann, L. Netz, B. Rumpe, and S. Stüber, "Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines," in *Modellierung 2022*. GI, 2022, pp. 33–48.
[16] D. Kolovos, F. Medhat, R. Paige, D. Di Ruscio, T. Van Der Storm, S. Scholze, and A. Zolotas, "Domain-specific Languages for the Design, Deployment and Manipulation of Heterogeneous Databases," in *11th Int. WS on Modelling in Software Engineering (MiSE)*. IEEE, 2019.
[17] A. C. Bock and U. Frank, "In search of the essence of low-code: An exploratory study of seven development platforms," in *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 57–66.
[18] A. Sahay, A. others Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *46th Euromicro Conf. on Software Eng. and Advanced Applications (SEAA)*. IEEE, 2020.
[19] M. Franzago, D. Di Ruscio, I. Malavolta, and H. Muccini, "Collaborative model-driven software engineering: A classification framework and a research map," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1146–1175, 2018.
[20] F. Inc, "GraphQL June 2018 Edition (GraphQL)," Facebook Inc, Standard, 2018.
[21] F. Drux, N. Jansen, and B. Rumpe, "A Catalog of Design Patterns for Compositional Language Engineering," *Journal of Object Technology (JOT)*, vol. 21, no. 4, pp. 4:1–13, October 2022.
[22] A. Butting, J. Michael, and B. Rumpe, "Language Composition via Kind-Typed Symbol Tables," *Journal of Object Technology*, 2022.
[23] R. Rodriguez-Echeverria, J. L. Cánovas Izquierdo, and J. Cabot, "Towards a UML and IFML Mapping to GraphQL," in *Current Trends in Web Engineering*, ser. LNCS, vol. 10544. Springer, 2017, pp. 149–155.
[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
[25] F. Becker, P. Bibow, M. Dalibor, A. Gannouni, V. Hahn, C. Hopmann, M. Jarke, I. Koren, M. Kröger, J. Lipp, J. Maibaum, J. Michael, B. Rumpe, P. Sapel, N. Schäfer, G. J. Schmitz, G. Schuh, and A. Wortmann, "A Conceptual Model for Digital Shadows in Industry and Its Application," in *Conceptual Modeling*, ser. LNCS. Springer, 2021.
[26] M. Dalibor *et al.*, "Generating Customized Low-Code Development Platforms for Digital Twins," *Journal of Computer Languages (COLA)*, vol. 70, 2022.