Generating Customized Low-Code Development Platforms for Digital Twins

Manuela Dalibor[1], Malte Heithoff[1], Judith Michael[1], Lukas Netz[1],
Jérôme Pfeiffer[2], Bernhard Rumpe[1], Simon Varga[1], Andreas Wortmann[2]
[1] Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de
[2] Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW),
University of Stuttgart, Stuttgart, Germany, www.isw.uni-stuttgart.de

***Abstract*—A digital twin improves our use of a cyber-physical system and understanding of its emerging behavior. To this effect, a digital twin is to be developed and configured and potentially also operated by domain experts, who rarely have a professional software engineering background and for whom easy access and support, e.g., in form of low-code platforms are missing. In this paper, we report on an integrated method for the model-driven engineering of low-code development platforms for digital twins that enables domain experts to create and operate digital twins for cyber-physical systems using the most appropriate modeling languages. The foundation of this method is (1) a code generation infrastructure for information systems combined with (2) an extensible base architecture for self-adaptive digital twins and (3) reusable language components for their configuration. Using this method, software engineers first configure the information system with the required modeling languages to generate the low-code development platform for digital twins before domain experts leverage the generated platform to create digital twins. This two-step method facilitates creating tailored low-code development platforms as well as creating and operating customized digital twins for a variety of applications.**

***Index Terms*—Digital Twin, Low-Code Development Platform, Domain-Specific Languages, Model-Driven Engineering, Code Generation**

I. Introduction

Digital Twins (DTs) [1] promise improving the understanding and use of Cyber-Physical Systems (CPSs) in automated driving [2], manufacturing [3], medicine [4], and further domains. For instance, they aim to prevent crashes [5], reduce resource consumption [6], and predict health issues [7]. DTs are software systems that are representing ('twinning' [8]) a CPS to act as its surrogate for analyses [9], control [10], or predicting its behavior [11]. On one hand, they gather data from and about the twinned CPS that may include data not accessible by the twinned system itself, such as historic data about its usage or strategic data about its planned use received from another system (ERP, MES). Consequently, a quintessential prerequisite for leveraging this data is making it available to other systems and human operators. On the other hand, DTs need to be configured and reconfigured over time to extract the desired data, aggregate it properly, and act on it. In many domains, the CPSs and, hence, their DTs are configured by domain experts yielding highly specialized, sparse, expertise about the CPS, its environment, and its use, but no formal software engineering education. These domain experts need to take care of changing the programming of the CPS and its DT over time to reflect deterioration or hardware changes (*e.g.,* replaced sensors or actuators) over time. Thus, it its crucial to enable these domain experts to properly configure, connect, and control DTs. Yet, no general approach for engineering and configuration of digital twins through domain experts exists [12], [13].

Low-code development [14] is a recent trend in model-driven end-user programming in which highly integrated modeling tools for a specific kind of applications enable end users in creating and configuring corresponding applications. Corresponding Low-Code Development Platforms (LCDPs) could support domain experts in creating, configuring, and controlling digital twins. Yet, to the best of our knowledge no LCDPs for digital twins exists.

To facilitate engineering and operations of DTs with LCDPs, we present a pervasive model-driven method to generate tailored LCDPs for creating, configuring, and controlling DTs. Our method relies on (1) the MontiGem toolchain [15] for the generation of information systems, (2) an extensible digital twin architecture model [16], [17], [18], (3) reusable language modules to define the modeling languages selectable in the LCDP and available in the digital twins, and (4) their integration with each other as well as their transformation into an integrated LCDP.

The *contribution* of this paper, thus, is a two-stage method to create tailored LCDPs for the creation and configuration of DTs. This method combines the model-driven generation of information systems with the generation of DTs from models of their architecture and latest results from software language engineering. First, models and languages for the DTs to be developed with the platform are collected. From this, the first code generator produces a web-based LCDP capable of modeling with these languages. Second, the DT to be developed is configured with various models, possibly including models of the languages collected beforehand. Through a second code generator, an executable instance of that DTs is produced that features self-adaptive behavior and visualization capabilities.

This method, hence, facilitates creating highly-specialized, local LCDPs for DTs, as well as creating, configuring, and controlling DTs by domain experts. Within a former publication [19], we have discussed our vision towards low-code approaches for digital twin engineering together with some

challenges. This paper is a significant extension and shows the method for engineering LCDPs for digital twins in detail together with a comprehensive example, more related work, and an intensive discussion.

In the following, Sec. II presents a motivation for the low-code development of digital twins in injection molding before Sec. III discusses low-code development, introduces digital twins and the DT architecture used in the following, presents the generator framework MontiGem and introduces language plugins. Sec. IV provides an overview of our approach for generating the low-code development platform and using the platform to generate digital twins and explains how we generate the low-code development platform. Sec. V describes how the LCDP can be used for the low-code development of digital twins. Sec. VI shows the functionalities of the generated digital twin during runtime and how it can be configured using models. Afterwards, Sec. II investigates its benefits through a case study, before Sec. VIII relates our approach to others. Sec. IX discusses observations and Sec. X concludes.

II. Use Case Injection Molding

In the "Internet of Production"[1] excellence cluster, we devised a self-adaptive, model-driven digital twin architecture and applied it to injection molding [16], [18]. Injection molding [20] is a batch processing production technique in which a plastic granule is heated and injected under pressure into an injection mold (the negative form of the product to be produced). It is one of the most common mass production processes for the plastic parts that we interact with on a daily basis. Figure 1 highlights the quintessential components of an injection molding machine.

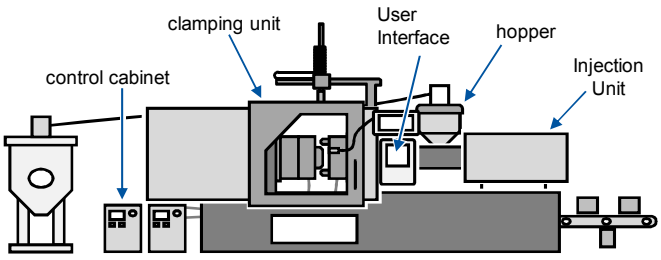


Figure 1: Main components of an injection molding machine

The typical injection molding process is as follows: (1) An operator configures the machine through its UI and inserts plastic granule through a hopper. (2) In the injection unit, the plastic granule is heated and molten into the desired consistency. (3) The screw transfers the plastic to the nozzle. (4) Under pressure, the injection unit injects the molten plastic into the mold while the clamping unit keeps the mold closed during injection so that the applied pressure is countered and the mold halves do not open up. (5) After a cooling time, the machine ejects the work piece from the mold.

Generally, various device components with multiple influencing variables (such as temperature or pressure) and process parameters are directly related to successful production of injection molded parts. But injection molding machines are sensitive to unintended and hardly predictable changes in their behavior due to deterioration and environmental effects. Consequently, two injection molding machines of the same type will require different configuration depending on their history and the environment. Hence, the expertise on how to configure a specific machine is highly specific and rarely made explicit but demands practical knowledge by experienced machine operators. This slowly increasing deviation towards highly specific configuration of the machine as-operated is common to all kinds of CPS.

For the large majority of these systems, domain experts without formal software engineering training need to be able to configure them accordingly. Current configuration mechanisms are unobstructed and untraceable, *i.e.,* the domain expert provides an initial configuration, starts a production, investigates the result, and adjusts the configuration accordingly. This takes time, consumes energy, and produces waste.

Leveraging DTs [21], [1] that monitor and control the system of interest can reduce configuration time, save energy, and prevent waste [18]. To this end, the DTs need to be programmed with adequate mitigation measures, *i.e.,* undesired conditions and countermeasures. Given that domain experts are rarely software engineering experts, low-code development can support them in successfully employing DTs accordingly. This, however, assumes that the operator of the system in question already has a DT to work with, as well as system to configure it properly, in place. As this also rarely is the case, we devised a method to generate LCDPs for the creation and configuration of digital twins such that domain experts can choose the most suitable modeling languages to (low-)code the configuration of their highly-specific DTs.

III. Background

We aim to enable domain experts without formal software engineering training to configure a DT leveraging low-code development. To this end, we combine various modeling techniques and model-driven methods to create and configure the platform. The models we use for configuration are more abstract than programming languages and allow domain experts to create powerful full digital twin applications nonetheless. This section introduces essential foundations.

A. Low-Code Development

Low-code development is a success story of model-driven engineering [14]. The term *low-code* first appeared in literature by Forrester in 2014 [22] as "platforms that enable rapid delivery of business applications with a minimum of hand-coding and minimal upfront investment in setup, training, and deployment." This trend of End-User Software Development [23] continues and is further purused by projects like Low-comote[2][24]. Where traditional MDE methods and techniques aim to support a variety of modeling applications (*i.e.,* class

[1]https://www.iop.rwth-aachen.de

[2]https://www.lowcomote.eu/

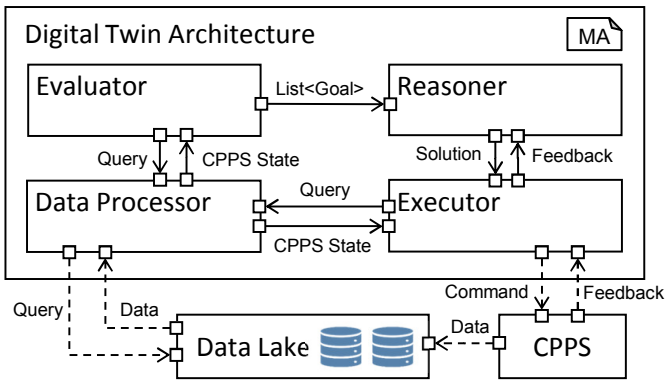


Figure 2: Digital twin base architecture [18] in MontiArc (MA) [39].

diagrams, the epsilon transformation language, or Xtend are not tied to specific applications), low-code development platforms leverages tightly integrated modeling platforms, can provide immense added value to modelers for very specific purposes, such as the development of mobile[3], web-based[4], or business applications[5]. These platforms provide graphical [25], textual [26], or form-based modeling environments [27] that focus on specific applications and comfortable UX, intelligent assistants, and well-integrated downstream tool chains [28]. Through this, the complexity of creating specific applications and the generated code are hidden from low-code modelers. Low-code development platforms are used to develop a variety of data-intensive applications. [19], [16] presents the architecture for an LCDP to develop a ditital twin.

B. Digital Twins

There is a variety of interpretations of the concept of digital twins. Some consider a digital twin a highly-precise simulation model, used at design-time of its twinned counterpart [29], [30], [31], while others understand it as software system representing the counterpart during its run-time [32], [33], [34] or something in-between [35], [36], [37]. For us, a digital twin is *"a set of models of the system, a set of digital shadows and their aggregation and abstraction collected from a system, and a set of services that allow using the data and models purposefully with respect to the original system"* [16]. Digital Shadows (DSs) are data structures that describe a dedicated view on the original system for a specific purpose. *"A digital shadow is a set of contextual data traces and their aggregation and abstraction collected for a specific purpose with respect to an original system"* [38]. The DS provides information only, whereas the DT also interacts with the system using the collected data.

Our architecture for DTs is developed using MontiArc [40], a component & connector architecture description language [41] that enables to model architectures as hierarchy of connected components. Components are black-boxes with a defined interface that consists of ports that can be either incoming, defining which types of incoming messages they can consume, and outgoing, defining which types of outgoing messages they produce. Components interact via unidirectional connectors between the ports of their interfaces. Furthermore, components can be decomposed into subcomponents or atomic. Figure 2 show the base architecture from which we derive the domain-specific DTs. The DT comprises a `Data Processor` component that connects to a `Data Lake` to aggregate data about the current Cyber-Physical Production System (CPPS) state. The `Evaluator` monitors the CPPS state and checks whether the CPPS operates as intended. In case it detects any anomalies it creates goals that specify the intended CPPS state. The `Reasoner` finds a solution for the current situation and hands this solution to the `Executor`. The `Executor` translates the provided solution into concrete CPPS settings and performs these on the CPPS. For specifying domain knowledge and the behavior of the DT, domain experts can utilize multiple modeling languages:

- UML/P class diagrams [42] to define the domain model with its elements and their relations.
- A language for Case-Based Reasoning (CBR) [43], which is a problem-solving paradigm that reuses solutions from formerly encountered situations, to find a similar solution to the current situation. A case consists of a description of the situation as conditions properties of the domain model, its solution in terms of actions to be performed, and the situation the solution intends to produce. A simple case could be that once the injection mold reaches a low temperature, it needs to be heated again, which should yield a situation in which a specific temperature is reached again. CBR models are interpreted during runtime of the DT and can be updated and exchanged during its runtime as well.
- An event language that domain experts can utilize to describe events based on situations in the real system based on values of monitored parameters contained in the domain model. Actions define reactions of the DT to an event, e.g., sending a goal for the reasoner to evaluate. Events and actions are linked by rules.
- A communication specification language which enables defining communication of data types from a specified endpoint and with a defined protocol. Currently, the architecture supports communication via OPC-UA and MQTT.
- An expression language for aggregating data. It includes the most commonly used functions such as maximum, minimum or summation functions. Aggregation is needed for the DS when collecting data.

C. MontiGem

We use MontiGem, the generator framework for information systems [44], as a model-driven platform to create and (re-)configure DTs using low-code techniques. This includes a set of modeling languages to create the DT: a language to

[3]https://www.salesforce.com/campaign/lightning/
[4]https://www.claris.com/
[5]https://www.quickbase.com/

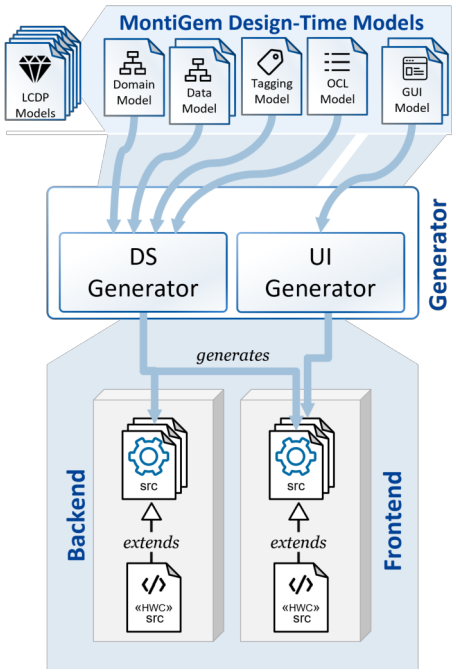


Figure 3: MontiGEM Generator framework.

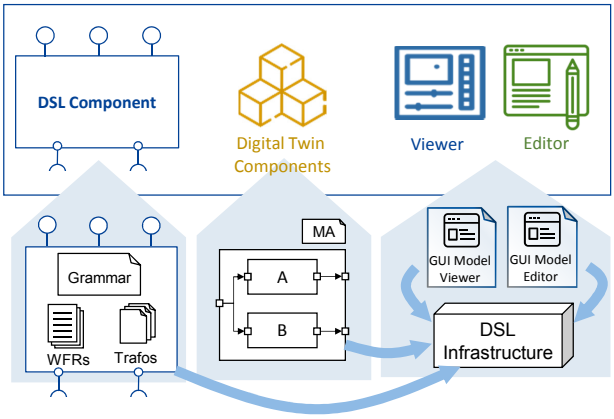


Figure 4: A detailed view on a Low-Code Development Platform Language Plugin consisting of a DSL language component, containing language constituents, a MontiArc component model for integrating the plugin's functionality into the DT architecture, and viewer and editor GUI models for editing models of the language in the LCDP.

define data structures based on UML class diagrams, namely CD4A, a language to define graphical interfaces, namely GUI-DSL [45], [15], a tagging language [46], [47], an event language [18] and a language to define goals. All used modeling languages are based on the language workbench MontiCore [48]. These languages are described with context-free grammars which integrate concrete syntax with abstract syntax. Additionally, they use well-formedness rules implemented in Java ("context conditions"). Models are parsed by a corresponding generator (Figure 3) and transformed into an abstract syntax tree. This abstract syntax is transformed again to the target syntax tree and provided to a template engine, that produces source code in the configured programming language.

This code generator has already proven its capabilities in real-world projects for financial management [49], energy management systems and the engineering of wind turbines as well as in research projects on goal modeling in assistive systems [50] and privacy-preserving information systems [51].

D. Low-Code Development Platform Language Plugin

Language plugins are complex software components which can be used to encapsulate a language, components to process models of this language and editors and viewer to create and manipulate models. Figure 4 shows the elements of language plugins comprising

- a language component [52], [53], [54] featuring the MontiCore [48] grammar of the language, context conditions, and model transformations pertaining that language,
- MontiArc [40], [55], [56] components interpreting models of that language to be embedded into the `Reasoner` and/or `Executer` of the architecture of DTs configurable with the LCDP under development,
- a GUI model from which, together with the provided DSL infrastructure of the DSL component, MontiGem generates a TypeScript-based editor for models of that language to be embedded into the LCDP under development, and optionally
- an optional TypeScript-based viewer to visualize the behavior of models of that language at runtime of the DT configured with the LCDP.

The latter TypeScript editor and viewer are integrated into MontiGem via its UI hooks for HTML.

IV. Model-Driven Engineering of the Low-Code Development Platform

In a two step approach, we can generate in the first step LCDPs to generate in the second step customizable DTs that can be used to create, configure, and operate tailored DTs using the most appropriate modeling languages for different domains. The novelty of our approach comes from combining the MontiGem code generation infrastructure with our model-driven architecture for DTs and DT language plugins. Our approach needs three different *roles*:

1) the *low-code development platform engineer* creates the LCDP model-driven,
2) the *digital twin designer* uses the LCDP to configure the digital twin to be generated and
3) the *domain expert* uses the digital twin to observe and control the CPPS.

The process of generating the LCDP starts with a set of *LCDP models* and a set of *language plugins* as input for the MontiGem generators (see Figure 5 top left).

This **set of LCDP models** are five different kinds of MontiGem design-time models (see Figure 3): The *domain*

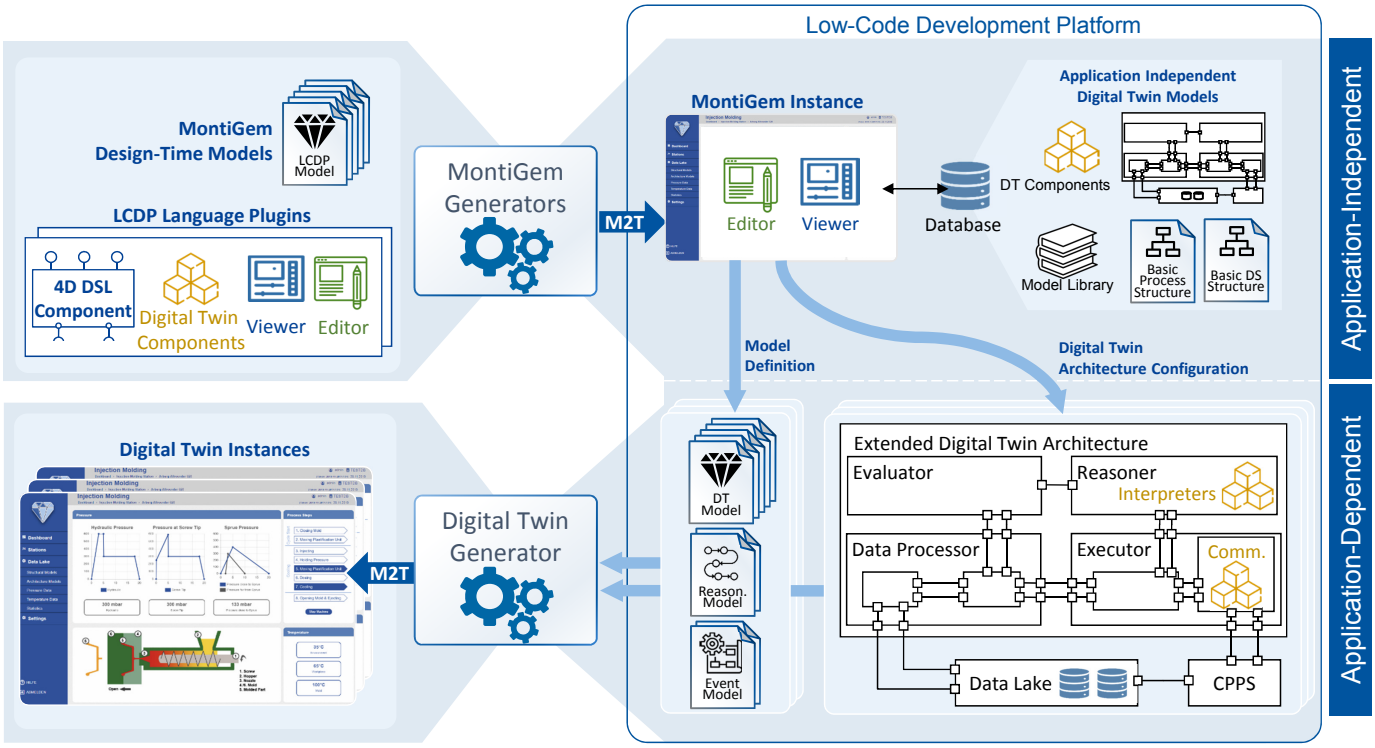


Figure 5: Using the models and language plugins (top-left), the MontiGem generators produce a tailored LCDP for creating and operating DTs (top-right). The generated LCDP consists of a web-based frontend and a database backend. In it, the languages defined by the plugins, the GUI parts defined in the GUI models, and different domain models are available such that domain experts can leverage these to configure DTs. By selecting the models to be used in the DT and the GUI models for representation of DT information, the common architecture of DTs (bottom-right) is parametrized an tailored accordingly. Using the DT generators, a new DT is produced with comprises an instance of that architecture connected to a new MontiGem instance representing information about the DT.

model using UML CD as notation describes the data structure for the generated LCDP, e.g., the class DTCommunication, with the attributes endpoint or topic or the classes DTModels or DTServices with its attributes. *GUI models* describe the graphical user interfaces of the LCDP, e.g., of the dashboard or detailed pages [45]. *Data models* define which part of the data of the domain model should be shown in each GUI, e.g., DT services, DT models or a ModelList with entries. *OCL models* describe input validators for input fields in the LCDP GUIs, e.g., to validate if the address of an endpoint follows a certain pattern. *Tagging models* add database technology specific information to the domain model, e.g., which attributes are unique in the database, or define rights and roles, e.g., the admin or LCDP configurator.

Besides MontiGem design time models that define the general appearance and functionality of the LCDP, **language plugins** allow a digital twin designer to create and edit models in the LCDP. Each of the *plugins* comprises a Domain-Specific Language (DSL) component which encapsulates a complete domain-specific language in form of a grammar defining its concrete and abstract syntax, as well as well-formedness rules, additional transformations and code-generators. These language plugins come along with components for the DT architecture, capable to process the models of this DSL, viewers, editors for the models, and a set of predefined models. By selecting language plugins, the LCDP engineer can determine which languages will be available for digital twin designers to use in the LCDP.

The MontiGem generator (Figure 5 top middle) takes all the predefined and application tailored models as input to compute a LCDP (Figure 5 right), which is provided to digital twin engineers as web application. First, from the domain model, most of the internal infrastructure is defined and generated. A relational database is build which holds the data structure. Additional to the database, several classes are generated to allow database access, abstract from database tables to allow a more general use of the data structure and allow an easier communication of the web platform and the server backend. Moreover, the MontiGem generator produces editors/viewers and the DSL infrastructure for models of the LCDP language plugin's DSL.

This LCDP is then provided to digital twin designers who have the aim to create one or more digital twins in connection to a certain CPPS and tailored for their specific needs.

V. Low-Code Development Platform

The generated low-code development platform provides application-independent parts, which can be used by a digital twin designer to configure the digital twin to be created. The selected configuration for one specific digital twin is then the application-dependent part of the LCDP.

A. Provided Application-Independent Parts of the Low-Code Development Platform

The LCDP provides application-independent parts, which can be used by the domain experts for creating different digital twins. It already includes viewers and editors for the models of the usable DSLs and a set of application-independent models (see Figure 5 top right).

The LCDP provides a **model library** (see Figure 6) with a set of different domain-specific models. This enables users to store and reuse models used in previous projects. Moreover, the models can be exported and imported between projects or for use with different tools if required.

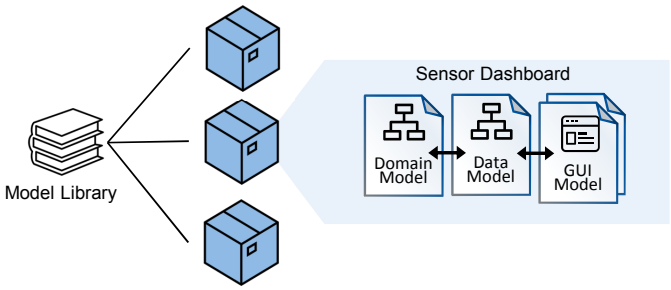


Figure 6: Model library, containing predefined models.

Also, the LCDP provides a basic process structure [57] as well as other structures of all relevant runtime languages as a class diagram. These basic structures include all concepts of the grammar of these languages. This allows us to store their models during runtime and enabled their runtime handling.

An architectural model with extension points describes the structure of the DT to be generated. These extension points can be filled with predefined DT components for five distinct parts that can be adapted: the data processor, the evaluator, the reasoning, the executor, and the communication (see Figure 7). Each of those parts are meant to be used during run-time of the DT and are further explained in Sec. VI. The feature diagram of Figure 7 shows various possibilities for configuration. The required features provide the services needed for a minimal working DT. The additional optional services can be used for more sophisticated behavior, such as action planning with models of the Planning Domain Definition Language (PDDL). Some of the digital twin services are able to handle models during runtime, others need models during design time. The editors and viewers of the LCDP language plugins allow to define these models in the required languages.

The LCDP comes along with the basic DS structure to be able to manage and store them during runtime of the digital twin. The conceptual model in [38] describes what concepts are needed for engineering a DS. The digital representation is modeled as a class diagram and consists of the composition of the *asset* the DS stands for, *data traces*, its unique *purpose* (e.g., "minimize waste in injection molding"), the system's underlying models, and a link to other DSs. The *data traces* originate from a *source* (e.g., a specific sensor), consists of *data points* and can be enriched with *metadata*. This way, the meta-model for the DS is adaptable and extendable to conform to every use-case. We use that meta-model in our implementation to stay consistent with the definitions of the Cluster of Excellence IoP.

B. Low-Code Configuration of DTs through the Platform

The LCDP comes with an intuitive user interface through which the DT designer configures the DT. The DT designer can be any person that knows the domain and the physical entity. For realizing a DT, no advanced programming skills are required. The low-code DT configurator guides the DT designer through the configuration process. It ensures that the minimal required specifications are given and that the designer is provided with meaningful options to configure the system step by step. The process of configuration creates the application-dependent parts (see Figure 5 bottom right) of the low-code development platform.

Definition of the domain. The DT designer has to start with the definition of the domain model. The *domain model* describes the context and CPPS with which the DT interacts in UML/P CD [42] with integrated OCL/P. The DT designer can (1) set up it's own model, (2) he can reuse models defined in the model library, and (3) in case he uses predefined models, he can adapt them to the exact needs of the desired DT. Our DT low-code platform is specifically designed for CPPS, thus providing a set of standard concepts in the model library that are often required to describe CPPS, like temperature sensors. The LCDP allows also to store and reuse own models which were defined in previous configurations.

Definition of constraints. Once the domain is defined, the designer can apply constraints on the concepts in the domain model, e.g., defining limits and boundaries on specific parameters. In order to enrich the domain model with further information, a tagging language can be used to apply tags to classes and attributes.

Digital shadow type definition. The *types of the digital shadows* encapsulate data about the physical entity that the DT requires for service provision. There are various types of DS, each tackling special information needs. The DT gathers information from the system and other external services (CPS, CPPS, apps) via the DSs. During the configuration of the LCDP a DT designer defines DS types which describe how the different DSs are created. First, the connection to external services and especially to the CPPS over the data lake has to be configured. This is done by specifying the incoming data structure in a class diagram and tagging it with API information such as a REST service. This allows for referencing of classes and attributes and how they should be aggregated. From that, the DT designer chooses the now discoverable data

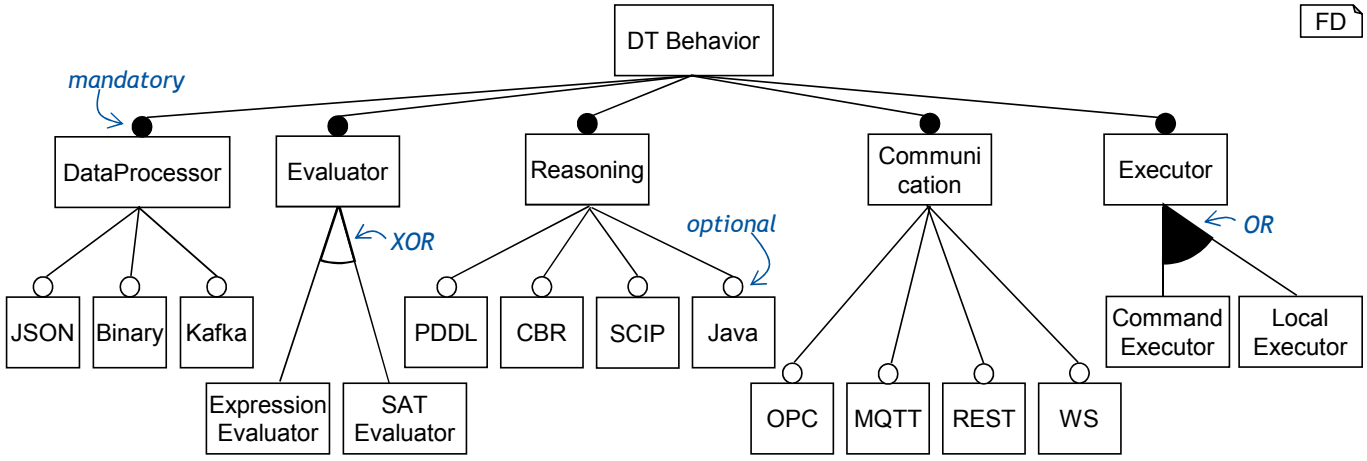


Figure 7: Feature diagram (FD) for the digital twin behavior.

points they want to include for the data traces. Next, the DT designer specifies the classes defining the data source and system asset in the domain model and implement the DS meta model by extending the corresponding classes. Now they can choose the DS's structure from those, the system's models, other DSs and the data traces. In addition, the very specific purpose for this DS is assigned. With all that given, all data can be aggregated with some description from the models. A standard aggregation language is provided which holds most functions a DT designer might need. Additional functions can be provided and implemented with an extension mechanism.

Digital twin service selection. The LCDP offers different *digital twin services* to interact with the underlying physical entity and influence its behavior. Thus, the DT low-code platform offers the possibility to define which services should be active in a specific DT and state the service's behavior explicitly. For this purpose, we provide graphical interfaces where the DT designer can choose available services for the generated DT (see Figure 8). Besides selection of the desired behavior and services (top left) and communication protocols (top right), additional information can be added for each selection, *e.g.,* for case-based reasoning a set of predefined cases can be uploaded, or MQTT requires to define an endpoint and the subscribed topics.

Digital twin roles. As different kind of users should be able to have different kinds of interaction and different views of the data, a *role and permission* infrastructure is used. That allows for different user groups, *e.g.,* a machine operator who only wants to see the information and interact with a specific CPPS. Thus, the user does not need to have access to the data of other machines. We can use tagging models to map roles and permissions to the domain models.

In addition, such restrictions can also be used for different DT services. This enables us to allow only specific user groups to add new communication endpoints or new reasoning models. Similar to the use of the tagging language in the CD, the service configuration allows the digital twin designer to connect roles and permissions to the different services.

Select and describe the visualization. Finally, the digital twin designer can (1) select predefined, (2) automatically derive, and (3) define new user interaction models. The model library already includes various GUI models in relation to the provided domain and data models that can be selected by the DT designer. Preexisting views and components are used to systematically show the DT data. Another possibility is to automatically generate the *user interface models* from the domain models, as the important information such as data structure or needed permissions are already defined (see for an approach in [15]). Moreover, the LCDP allows the digital twin designer to define handwritten GUI models. The designer can define the GUI via drag-and-drop of graphical components or in a textual model. The LCDP provides an editor to define how the data from the domain is displayed in the DT (*e.g.,* usage of tables and charts) in GUI models. This allows for individually adaptable views in a digital twin. The data models needed for the UI can be extracted automatically, as the GUI models already define what data should be shown. This is, e.g., the value of an attribute or aggregated information, such as a minimum value or the last few historic values in a time series. The LCDP already provides commonly used visual components and data aggregation methods. If those are not sufficient additional components can be added to the generated DT.

C. Generating the digital twin

Once the domain expert has selected and defined the models and selected the digital twin services, he can generate the DT instance tailored to the designer's requirements. The generated DT has a database to store data about the physical entity or the user interaction, it provides a frontend for visualizing data about the physical entity and the status of the DT. This generations step automatically integrates the corresponding digital twin components into the DT architecture. The DT services realize the DT's behavior, *e.g.,* the kinds of events that it should react to, the actions it may trigger on the physical entity, and also smartness. Furthermore, the frontend also

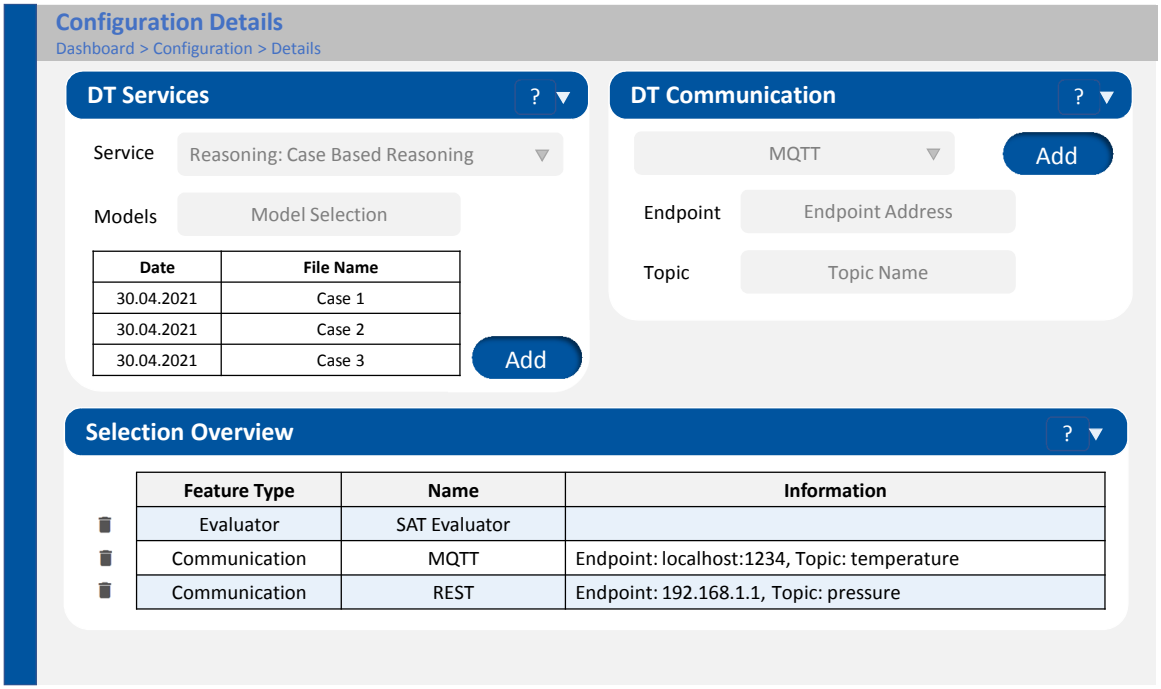


Figure 8: Low-code digital twin platform configuration view.

supports adaptation of the DT, *e.g.,* by adding or exchanging parts of the data structure.

What needs to be done when the DT has to be changed? As the DT is completely generated, each change regarding its core structure has to be made in the LCDP to recreate the DT. This has some consequences for an already existing digital twin: 1) handwritten additions to the DT might need to be adapted to a changed structure, and 2) when there is already data created in the existing DT the data might have to be migrated to the new structure. The LCDP saves the current DT configuration and provided models so the DT designer can easily continue or make changes. Nonetheless, changes which only affect interpreted models can be done during run-time. Adding new cases to the case base adapts the DT behavior while adding an additional DS type creates new DSs and can be visualized in a default GUI component.

VI. Digital Twins during run-time

The digital twin provides the functionalities specified in the digital twin services during run-time of the application. Furthermore, the application users (domain experts) can customize the DT during run-time. They can, e.g., specify new reasoning models, define new event models, set planning goals, or edit or create new DS types.

A. Functionalities of the digital twin at run-time

The digital twin continuously gathers data from the CPPS and processes it. On the one hand, it visualizes data and presents it to the users using GUI components. They can then decide to interact with the machine. On the other hand, the DT evaluates the system's current state and automatically adjusts the machine's settings if the system tends to reach a critical state.

The **DataProcessor** handles conversion between the data received from or send to external sources, *e.g.,* CPPS or other applications, and the internal structure. There are different protocols or data descriptions like binary format, JSON, or Apache Kafka streams. This allows the services to convert data and to just work on the internal data structure. The **DS types** hold all information on how to construct a DS. The DataProcessor then searches for the DS type for the correct asset with the corresponding purpose and obtains a structured DS. It adds all data points to the DS's data traces and can then calculate aggregated data based on the underlying aggregation models. The result is a structured DS for the specific purpose which contains all data from the machine and its aggregation to fulfill this purpose. The next steps then decide on how those DSs are utilized.

The behavior of the DT is realized though specific implementations of the evaluator and reasoner component in Figure 2: The **Evaluator** should handle constraints and expressions and are evaluated when checks have to be performed, *e.g.,* if a temperature value is valid or in a specific range. The Evaluator is used by other services like the Reasoner and the Executor. The **Reasoner** service handles actions, that have to be executed when specific events occurred. These components monitor the behavior of the physical entity and choose when and how to interfere.

The **Executor** is the part that performs actions which have arisen by the logic of other services. The executor creates a command that can be sent to a given endpoint. Such a

command differs between CPPS and is defined in combination with the communication.

To **communicate** with other services the DT provides a range of communication interfaces which can be configured to customize and adapt to different circumstances. Such configuration can consist of the definition of the other endpoint, such as IP/port address, the topics of the connection, *i.e.,* which information should be exchanged, and which service uses this communication endpoint. Current supported communication interfaces are MQTT, OPC UA, WebSocket, and REST. The DT itself communicates via the defined communication endpoints in the DT services or via the DSs. Each service can communicate with any given endpoint, *e.g.,* other applications, CPPS, or even with other services of the same DT. The topics fulfill different requirements and each of the DT services has already predefined topics to work with. Additional communication topics can be added to match the endpoint to communicate with. A topic can be any identifier that both endpoints agree on. The communication with topics is handled by the DT services. Data can be created and send to other registered endpoints, or the service is triggered by new data on registered topics.

The supported communication interfaces provide different kinds of registration and communication setup:

- **MQTT** uses a message broker where clients can register with specific topics and get a data update when data is sent with the registered topic. The client just has to know the central message broker and does not need to know which other endpoints exists.
- **OPC UA** provides two different modes: (1) a broadcast mode, which works similar to MQTT and (2) a peer-to-peer mode where a client and server communicate.
- **WebSockets** connect a client to a server and can be used to send any kind of data. The server endpoint has to be known to establish a connection. Topics can be realized by either have a connection per topic or send each message through the same connection but handle them with different logic depending on the content.
- **REST** provides a simple but restricted communication interface. A connection is opened with a request by the client and closed, when a response is sent back. There are extensions like REST keep-alive that retain a connection for multiple information exchanges. REST usually provides one endpoint per specific logic like a single topic.

User Access Control. Finally the DT supports role based user access control. This allows for the setup of different roles, which can be granted different permissions, *e.g.,* having a supervisor role and a operator role that are permitted to perform different tasks on the DT. The user interface can be configured to display different views based on the roles, thus getting user specific UIs *e.g.,* a machine operator may take an interest in the health state, current order, and capacity rate of one specific CPPS while the head of production at a production location rather requires an overview of current activity at this location.

B. Configuration of the DT during run-time

In contrast to the more static configuration of the DT in the LCDP, we allow for a dynamic configuration of DT services during run-time. The following aspects can be edited by platform users at run-time, through settings and editors in the user interface of the DT itself. Each of the DT services can support configurations/models which are interpreted/executed during run-time. Each of the services can have different implementations and execution logic (see Figure 7). This allows the DT users to modify the behavior of the DT. The DT needs to be regenerated, in case there are changes needed in the models mentioned before.

Define event models. Different services are able to interpret event models. These define behavior and reactions in case specific events should occur. One frequent task of DTs is reacting to changes in the physical entity or the context in which the physical entity operates. Thus, the low-code platform provides an event language for specifying events and how the DT reacts if these events occur.

Define cases for the case base. DTs of CPS often operate in changing environments where, *e.g.,* significant temperature changes can occur. In addition, their configuration depends on many variables like the raw material, wear and tear of the machine, and the produced parts. Case-based reasoning is a reasoning mechanism that resembles the way that humans deal with new situations. They remember a previous situation in the past and try to adapt the solution that worked in the past to the problem at hand. Thus, we chose case-based reasoning as a way to inject domain knowledge into the DT. For this purpose, we created a domain-specific language that helps domain experts to formulate their knowledge but in addition is machine-processable and thus can be processed by the DT.

Configure communication. The generated DT provides communication infrastructure to access, *e.g.,* a machine interface and change process parameter, or to read recent sensor values. It supports multiple different communication interfaces. Which one is used and how it is configured can be set in the DT itself.

Edit or create new DS types. If a DT user needs to investigate a different part of the machine, they can decide to adapt existing DS types or create new ones. Since visualizations and information displaying is generated from the GUI models, only a generic view on the new incoming DSs is possible. This presentation contains its structural elements and a plain view on the data traces in tables. Data aggregation is possible for interpretable aggregation models. The aggregation language, which is provided within the LCDP, is limited to the predefined functions in this case. For different aggregation models or extended functions a regeneration is necessary.

There exist different situations where and when models for the DT services are created or changed. They can be

1) provided by the DT user through the UI,
2) retrieved from external sources, or
3) automatically adapted by the DT services.

An initial knowledge base could be provided by the user. and then get adapted by the execution of a reasoner. This allows for constant adaption and optimization of the DT.

New models at run-time can influence the application by being added to the list of available models. The new models can be used in the next iteration of the model execution. This also allows for multiple people to add new models simultaneously.

Model checks and validation. To make sure that a user supplied configuration or models are valid, the application contains logic that checks such models and provide errors. Depending on the kind and usage of a model it might be necessary to check some basic models of the DT such as the domain CD.

Aspects which can not be changed during runtime are especially the domain model, as it defined the database of the digital twin. This requires a regeneration process and, thus, has to be done in the LCDP. Furthermore, it requires a data migration of the digital twin database.

VII. Case Study

This example guides through the model-driven development process of the LCDP for digital twins for the use case of a digital twin for injection molding. The example starts with the configuration and generation of the LCDP, then presents the configuration of the digital twin via the generated LCDP, and finally shows the resulting DT for injection molding and modifications possible at runtime.

A. Modeling a low-code development platform

Modeling the low-code development platform for digital twins consists of 1) defining MontiGem design-time models that describe the domain, the data structures, and the appearance of the LCDP, and 2) selecting appropriate language plugins.

For generating the LCDP configuration view (see Figure 8) with MontiGem, a domain model and GUI model is necessary. An excerpt of the domain model is shown in Figure 9. It shows a class diagram model `LCDP_data` (l. 1) that amongst others, defines classes for `DTServices` (ll. 2-5), that enable users to get an overview over all available DT services and associated models, and `DTCommunication` (ll. 6-8), that lets users choose from available communication kinds and configure them via the LCDP. The service and communication domain classes are referenced by a corresponding GUI model. From this, MontiGem generates a UI dialogue similar to the one in Figure 8.

In this example, the digital twin uses case-based reasoning to react to occurring events, thus, users of the LCDP configuring the DT should be able to formulate models of the case-based reasoning language in the platform. Language plugins bundle language definitions via DSL components, their editor and viewer in the LCDP, and a MontiArc component for processing models of the language in the digital twin architecture. Figure 10 shows the language plugin for case-based reasoning. At the top, the DSL component `CBR` defines the

```
1  classdiagram LCDP_data {
2    class DTServices {
3      List<Service> serviceList;
4      List<Model>   modelList;
5    }
6    class DTCommunication {
7      List<CommunicationService> communicationServiceList;
8    }
9    class Service {…}
10   // further LCDP domain classes
```

Figure 9: Excerpt of the class diagram for injection molding.

language constituents of the case-based reasoning language. It references the grammar of the language (l. 2) defining the syntax of case-based reasoning models and lists elements the language provides, for instance, the production for defining cases (l. 3), a Java generator (l. 4), and a set of well-formedness rules (l. 5) for checking the correctness of models of the language. Additionally, the plugin contains a MontiArc digital twin component `CaseBasedReasoner` that can be integrated into the digital twin architecture as a subcomponent of the `Reasoner` to which a generated dispatcher subcomponent connects to. The `Reasoner` comprises a subcomponent `Reasoner` translating incoming lists of goal instances obtained from the `Evaluator` into lists of the required goal subtype. This exploits that the `Dispatcher` is generated after knowing all language plugins, and, hence, available goal subtypes. Using this knowledge, it checks the types of incoming goals, transforms, and dispatches these to the respective specialized subcomponent, here `CBRReasoner` expecting a list of `CBRGoal` instances.

It features a distinct input port accepting a list of `CBRGoal` instances. The `Dispatcher` subcomponent takes a list of Goal instances (the super class of all specialized Goals) and uses reflection over the known Goal subclasses to (i) identfiy which subcomponent should receive the goals and (ii) transforms the List<Goal> into a list of the required subclass instances. This is possible due to the `Dispatcher` being generated after knowing all supported language plugins, i.e., its behavior is an if-then-else block as illustrated for the example of supporting CBR and PDDL.

The implementation of the CBR component can interpret models of the case-based reasoning language. To enable users to view and edit CBR models, the plugin also contains GUI models from which the MontiGem generator produces an editor in the LCDP platform (compare the bottom of Figure 10).

B. Using the LCDP to model a DT for injection molding

After generating the LCDP, it enables the digital twin designer to model a DT, e.g., for injection molding. The following presents different models describing the digital twins structure, constraints on this structure, its communication, and its visualization in the digital twin cockpit.

Figure 11 shows a class diagram in the domain model viewer of the LCDP displaying an excerpt of a domain for a DT. The CD defines two new classes (`PlasticizingPhase` and `PlasticizingUnit`) and reuses existing classes from the DT library such as the

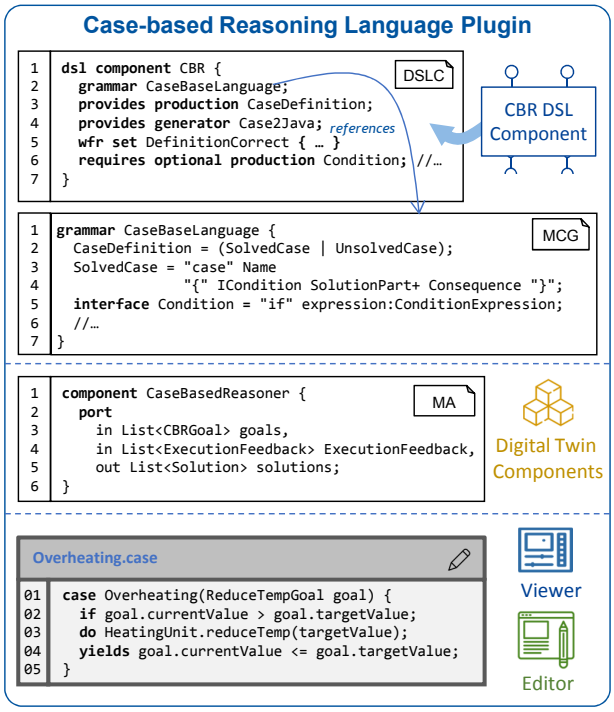


Figure 10: Excerpt of a language plugin for case-based reasoning.

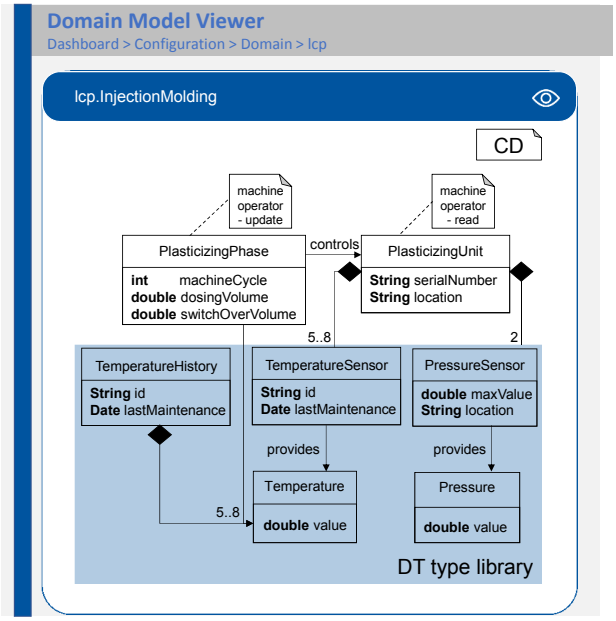


Figure 11: Excerpt of the class diagram for injection molding with tagged permissions.

`TemperatureHistory` which stores a list of recent temperatures, or `Temperature` which contains the temperature value. In addition, the diagram contains permissions that are added to the classes via tagging. These restrictions add a requirement for specific permissions when an object is accessed. Permissions can be required for reading a whole object

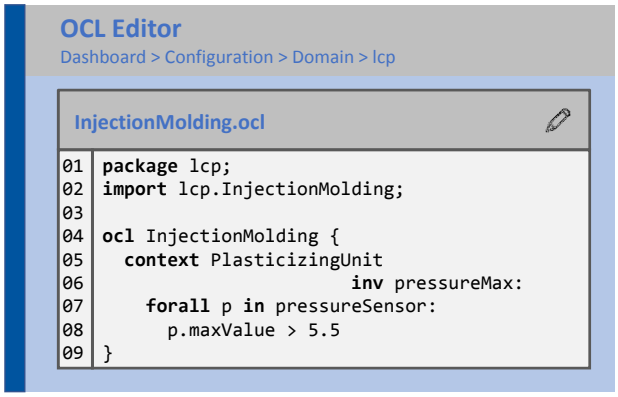


Figure 12: OCL model constraining the model from Figure 11.

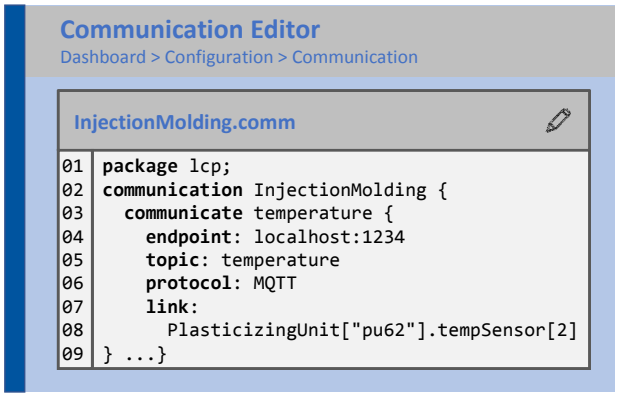


Figure 13: Communication model that specifies an MQTT communication of temperature values in an injection molding machine.

or even specific attributes. A class in the class diagram can have multiple different access permissions to provide further customization.

The model defined in Figure 12 adds additional restrictions to classes of the class diagram. Every class (l. 7), attribute (ll. 7-8), and association can be navigated, referenced, and extended with some constraints. Those constraints can be used for run-time validation checks, *e.g.,* the target pressure value should always be greater than 5.5 psi. The generator creates checks in the user frontend and application back-end of the DT to provide checks for the data input by the user or retrieved from other endpoints.

Figure 13 specifies the communication endpoint for the channel (l. 4) and the topic through which to communicate changes in the temperature value (l. 5). The communication protocol is MQTT (l. 6). Also, the communication model is linked to the physical entity that produces these values (l. 8), so in the example model, the temperature originates from the second temperature sensor in the `PlasticizingUnit` with serial number `pu6235`. Every attribute of the given domain class diagram can be linked and sent or received from the specified endpoint.

After the communication is established and the domain

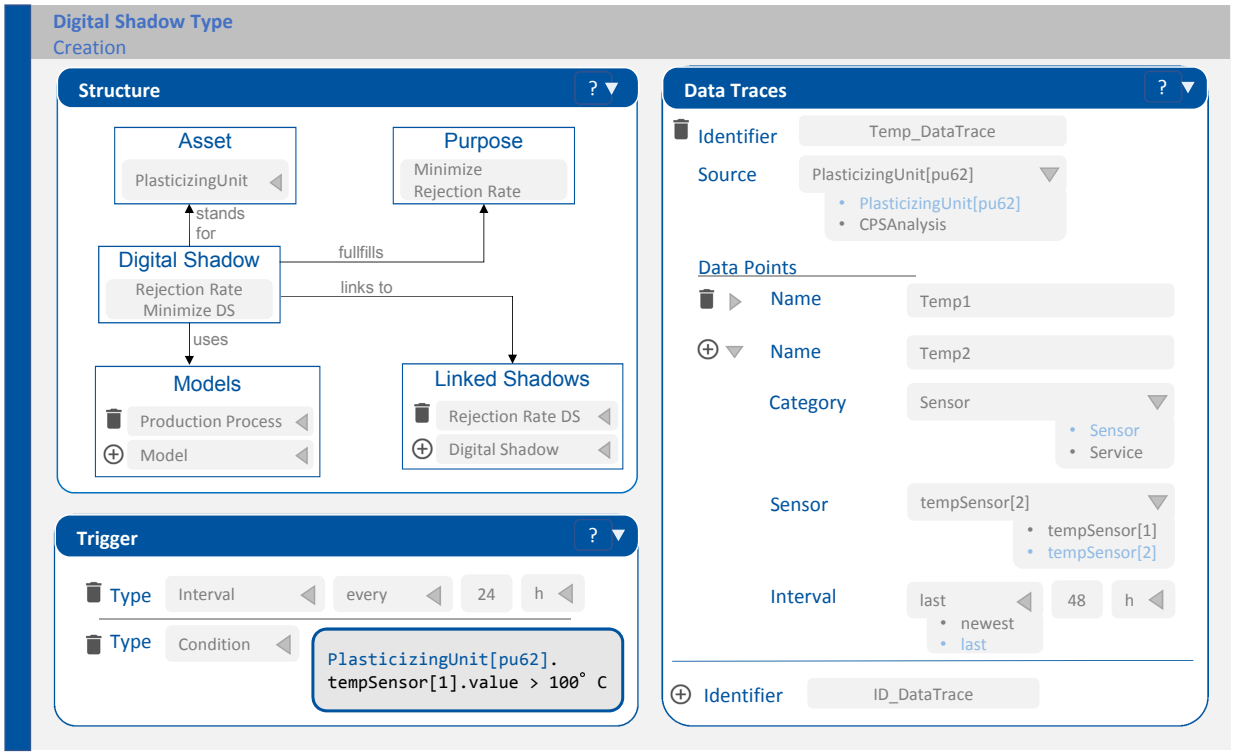


Figure 14: Configuration of the DS structure, including its defining asset, purpose, models, linked shadows and data traces. A trigger can optionally be given.

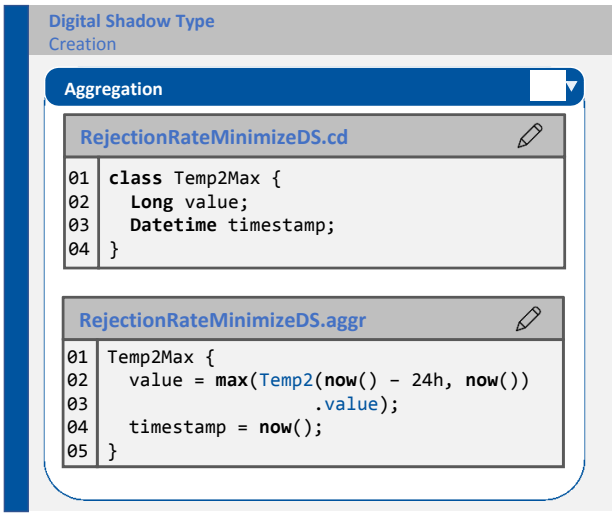


Figure 15: Aggregation of the machine's second temperature sensor using an aggregation language.

model implements the Digital Shadow meta model, the DT designer models the DS types to link the CPPS to the DT. Figure 14 shows exemplary the configuration of a DS type. First, on the left side, she chooses the asset the shadow stands for, the models that are used or delivered together with the DS, other linked shadows and a purpose. In this case, the DS stands for the injection molding machine and its unique purpose is to minimize the machine's rejection rate for temperatures too high. One of its data traces contains data points of the machine's temperature sensors. In addition, triggers in form of a schedule or a trigger condition decide when a new instance of this DS is constructed. To act purposefully the DS type needs some form of data aggregation. Figure 15 shows how the given aggregation language is utilized to model the sensor's maximum value of the past 24 hours. The upper class diagram specifies the return type, the lower aggregation model uses a maximum function over the time interval. Commonly used functions are included but the language is also extendable by self-implemented functions as well.

A given textual GUI model (Figure 16 a) from the model library (see Figure 6) is used to create the graphical view (b) for the user interface. The GUI language provides an easy-to-use abstraction to describe the graphical alignment of components. MontiGem already provides standard components like the *card* (ll. 1-10) and *datatable* (ll. 3-9). The GUI model connects different values (*eventStart*, *eventEnd*, ...) to columns of the datatable to indicate which values should be shown. The datatable then loads the information and displays a row for each event received.

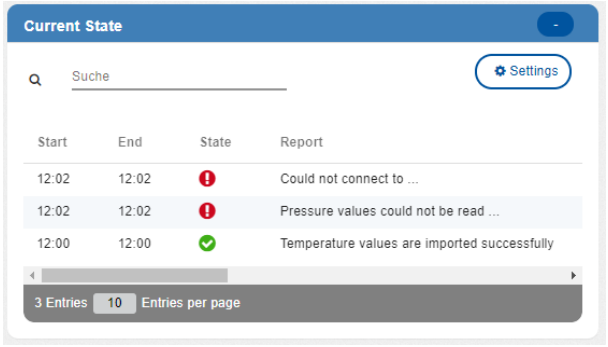
In addition to defining DT models, the digital twin architecture can be configured. Figure 8 shows the architecture configuration view of the low-code development platform. The view facilitates choosing from different DT services that should be available in the digital twin instance. In our case, we select case-based reasoning as a service for reasoning about

GUI Model Editor
Dashboard > Configuration > GUI > DT

DTStateGUI.gui

```
01 card "State" {
02   head { label "Current State" }
03   body {   datatable "State" {
04     rows < states {
05       column "Start",  time(eventStart);
06       column "End",    time(eventEnd);
07       column "State",  colorBox(state);
08       column "Report", message;
09   }}}
10 ...}
```

(a) GUI model for a state logger for the injection molding DT.



(b) Graphical result of the GUI model in the frontend

Figure 16: excerpt of a GUI with (a) the textual GUI model and (b) the corresponding graphical representation.

Event Editor
Dashboard > Events > Add Event

TemperatureTooHigh.event

```
01 behavior ControlPlasticizingTemperature {
02   event TempTooHigh for PlasticizingPhase {
03     checkTemp@(-1) > 500
04       && checkTemp@(0) > 500
05   }
06   action ReduceTemp for PlasticizingPhase {
07     new ReduceTempGoal(checkTemp@(0), 500);
08   }
09   rule TempTooHigh => ReduceTemp;
10 }
```

Figure 17: Specification on how to react to temperatures too high for the system. Here, the evaluator returns an action goal to reduce the temperature. The current temperature and the upper limit are forwarded as goal parameters.

Case Editor
Dashboard > Cases > Add Case

Overheating.case

```
01 import injectionmolding.HeatingUnit;
02
03 case Overheating(ReduceTempGoal goal) {
04   if goal.currentValue > goal.targetValue;
05   do HeatingUnit.reduceTemp(targetValue);
06   yields
07       goal.currentValue <= goal.targetValue;
08 }
```

Figure 18: Case model in the case base which specifies the reaction to the previously defined goal.

events occurring during the runtime of the digital twin. With this, the interpreter for case-based reasoning is available in the digital twin instance, and models of the language can be added or modified during runtime (see Sec. VII-C).

C. Operating the DT

After designing the DT for injection molding via the LCDP, the DT can be generated, deployed, and operated. The digital twin cockpit of the resulting digital twin instance is similar to the view displayed in the bottom left of Figure 5. This view strongly depends on the models defined in the LCDP in the previous step. During runtime, the DT is adjustable via runtime models, e.g., adding new event models for new events occurring and adding new cases to handle events to the case base. Figure 17 shows a common event in the injection molding process, which handles the overheating of the injection nozzle when its temperature is higher than 500 degrees (ll. 3-4). When the event occurs, a rule applies (l. 9), that reduces the temperature by sending a `ReduceTempGoal` parameterized with current temperature and target temperature (ll. 6-8).

Figure 18 shows a case that can handle the goal sent by the event. The case `Overheating` compares both `currentValue` and `targetValue` (l. 4) and calls a method on the heating unit of the plasticizing unit to reduce the temperature (l. 5) such that the current temperature is lower or equals the target value. Whenever a case with a condition matching the situation is found, its solution is applied to the situation and the system monitors whether the desired target situation has been reached. If no matching case is found, the reasoner searches in the vector space of conditions for a similar case. If a case is found this way, its solution is applied as well and if it yields the desired situation, a new case is synthesized based on the undesired situation and the actions of the found similar care. This way, the system learns new behavior based on existing cases to some degree.

VIII. Related Work

Related to our approach, we discuss existing low-code platforms, approaches to develop digital twins and possibilities to configure digital twins during runtime.

Model-driven digital twins. Model-driven software development is a well established software development paradigm, therefore several approaches for the development of digital twins exist [58], [59], [17], [60]. Li et al. [58] describe a model-driven digital twin for a robotic assembly. Magargle et

al. [59] describe a model-driven approach for a digital twin of a breaking system. Kirchhof et al. [17] describe how to connect a CPS with a digital twin using MDE methods. Bordeleau et al. [61] identify various MDE techniques to tackle three different challenges in digital twin development: (i) "management of heterogeneous models from different disciplines", (ii) "bidirectional synchronization of digital twins and the actual systems" and (iii) "the support for collaborative development throughout the complete life-cycle" showcasing that MDE can be used effectively to develop digital twin platforms. Similar to Bordeleau, Mandi and Lucero [62] demonstrate the benefits of using a model-based approach to develop and integrate digital twins. They however remain on a model-based approach in contrast to a model-driven one. Muñoz et al. [60] explore the use of different modeling languages to build and test digital twins of a high degree of abstraction. A prototype of that framework was validated as a proof of concept with a digital twin of a Lego car. To the best of our knowledge, they do not provide a more general approach to generate digital twins for various application domains.

Low-code development platforms. Low-code development platforms move MDE technologies from research to practice. Prominent examples are, *e.g.,* structr[6] for data modeling and service API generation through a graphical editor, VisionX[7] for developing mobile and web apps, and Mendix[8] for business applications. Book and Frank compare in [63] multiple LCDPs, similar to [64]. Highlighting the differences between platforms like A12 [65], Appian [66], Bonita Platform [67], Mendix [68], Microsoft Power Apps [69], MontiGem [70], Now Plattform [71], Out Systems [72], Pega Plattform [73], Quickbase [74] or Salesforce [75], all of these focus on providing a specific platform for specific kinds of applications.

Research also investigates advancing various aspects of LCDPs. This includes improving the user experience of by exploiting augmented reality in scenarios where modelers have to place objects in the environment is presented and evaluated [76] as well as integrating DevOps and MDE principles and practices in low-code engineering [77] to reduce learning curve for DevOps tasks. Other research investigates making LCDPs more intelligent using machine learning techniques [78]. All of these focus on specific improvements to LCDPs but not in the integrated engineering of a low-code development platform together with its connected digital twins.

Developing digital twins. Regarding the development of digital twins, various modeling frameworks have been conceived. For instance, Eclipse Ditto[9] is a framework for developing DTs that provides standardized APIs for IoT devices. In combination with the Vorto modeling language[10] for describing device interfaces, it can be used to describe digital twins that are connected to (Cyber-Physical) Systems. This framework focuses on data structures and does not provide means for specifying DT behavior. Similar commercial solutions are Amazon Greengrass[11] and Microsoft's Digital Twin Definition Language[12]. However, their main focus is on the data exchanged between the digital twin and its physical counterpart. Compared to our solution, the behavior, i.e., the evaluation, reasoning, and communication, of the digital twin cannot be defined via low-code modelling, and, instead, requires hand crafted executable code to be deployed to the respective cloud platform. Consequently, they are only suited for software engineers, whereas our solution does not require domain experts to have any programming skills. Major advantages of Greengrass and Microsoft's digital twin platform is that their digital twins are embedded into their respective cloud ecosystems accompanying data processing algorithms, machine learning capabilities, and CI/CD functionalities. By deploying the generated digital twin of our approach to one of these platforms, however, enables users to experience all these advantages of the platforms, too. Furthermore, the MontiGem digital twin language plugins could be extended with concepts of the commercial platforms to ease the deployment of our digital twin to one of these platforms.

Siemens MindSphere[13] is another commercial solution that tackles visualizing the physical entity and thus can speed up the design phase of new CPS. These solutions focus on the digital twin and do not consider the platform bringing these to life.

Configuration of digital twins during runtime. Our method also relates to using models@run.time [79], which enables the bi-directional synchronization between digital twins and actual systems [80]. For instance, Vogel and Giese [81] introduce different models@run.time to capture different system concerns for self-adaptive systems, Mayerhofer et al. [82] proposes a methodology for developing executable DSLs by providing mechanisms to describe model behavior when interpreted, or Combemale and Wimmer [83] suggest transferring design-time models to run-time for CPS development. Naturally, these approaches focus only on models of the system, not on the design time models to create the platform itself.

To the best of our knowledge no solutions for the integrated development of DTs with their LCDPs exist yet. Projects such as the Lowcomote[14] training network aim at facilitating the engineering of LCDPs for various application domains and might contribute into this direction.

IX. Discussion

Our method focuses on generating an easy to use LCDP that can be used to create and configure DTs. This includes the definition of important structures of the digital twin with common domain specific methods and the adaptation of the digital twin during the run-time. As such, various design decisions were made, which we will discuss in the following.

[6]https://structr.com
[7]https://visionx.sibvisions.com
[8]https://www.mendix.com
[9]https://www.eclipse.org/ditto/
[10]https://www.eclipse.org/vorto/
[11]https://aws.amazon.com/de/greengrass/
[12]http://www.aka.ms/dtdl
[13]https://siemens.mindsphere.io/en
[14]https://www.lowcomote.eu

The low-code development platform

The generated low-code development platform provides a web-interface for digital twin designers to configure and generate digital twins.

Graphical interfaces vs. mixed graphical and textual LCDP inputs. Our approach for a LCDP focuses on textual models which can be manipulated in editors, we provide visual representations for models in viewers and we provide forms to create and edit some of the models. Low-code approaches are often related with graphical interfaces for modeling. This makes sense for various DSLs, however, in some cases such as a case base this might not be helpful. There exists also other low-code approaches which do not consider graphical models [26]. We consider a mix between textual models and visual representations useful for the creation of digital twins. However, graphical editors could be included as extensions.

Mixing graphical and textual modeling. In our platform, we support mixing graphical and textual modeling, depending on the provided viewers and editors. While we are aware of differences between both approaches [84], we believe that the platform itself should not prescribe one way or the other. Instead, through selecting suitable language plugins, platform owners can tailor it to the needs of their users.

Usability of a textual GUI modeling language. Since it is not very common to use textual languages for the formulation of graphical user interfaces, one can ask whether the GUI language provides an easy-to-use method for describing the graphical orientation of components. We are using the GUI language since 2019 for the engineering of a real-world full-size enterprise information system within the MaCoCo project [49]. This includes more than thousand lines of code in 71 GUI models until now. As far as we have experienced it, it is a convenient way to precisely define the essentials of a GUI, which uses well defined graphical components.

Adaption of models and the code. Following Bock and Frank [85], various low-code development platforms do not provide or only partly provide adaptation mechanisms for data and process models. Our approach allows full adaptation of the models as well as of the generated application. Using an extension mechanism within the generator [48], the handwritten additions in the software code are kept during regeneration.

Modeling language modularity and evolution. To support addressing evolving challenges with a LCDP for DTs, the employed modeling languages and language infrastructure must allow for modularity within the language. This includes the combination of building blocks and the establishment of language hierarchies [86], which, for instance, is important if DTs for different application areas in smart manufacturing request different depths of, *e.g.,* event or reasoning models. As we allow to select the languages within LCDP language plugins, our approach allows to select different depths of languages with according components which are able to handle them. However, research on modeling language modularity and evolution has to be deepened.

Reuse of models. In existing LCDPs, reuse is addressed at a generic architectural level and not at a domain-specific level [85]. Our approach allows to reuse domain-specific models created for digital twins in a former configuration. Moreover, we provide a model library including different concepts that are often required to describe CPPS. This provides digital twin designers more freedom in the engineering process. Challenges when using model libraries, however, are the need to keep them in the same version as the used modeling languages as well as the need for model version management.

Collaborative Development. DTs of production systems will address multiple concerns of the twinned system and its, *e.g.,* strategic, context. Consequently, multiple stakeholders, such as shop-floor experts and managers, might interact with the DT collaboratively. Textual modeling techniques generally support this. Within the last years, a variety of collaborative modeling tools evolved as browser- or cloud-based solutions [87], [88], [89]. However, within the application domains of DTs, areas such as support for graphical modeling, identifying and resolving modeling conflicts, as well as considering roles, rights, and corresponding views for successful collaborative intra- and inter-organizational modeling remain to be investigated.

Comparison with existing IoT-platforms. Currently, the digital twin has to be defined in the LCDP and can be configured during run-time in our proprietary environment comprising of DSLs for describing aspects of the platform, the DTs and the dashboard. There already exist industrial solutions by major software companies like Microsoft and Amazon and their cloud platforms to develop digital twins, Azure IoT and IoT Greengrass respectively. Today, our environment is not integrated with these platforms, yet. However, in its current state, the LCPD as well as the digital twin created with it, could easily be deployed as applications to each of these cloud platforms and use the computational power and scalability of these platforms. This integration would be beneficial as both platforms support MQTT for communicating with the CPPS and the data lake out-of-the-box. Furhtermore, since we do not specify where the data lake is located, it could be realized via the cloud platforms storage options. Besides their infrastructure, the platforms also provide own functionality to develop and operate digital twins. For instance, the Azure digital twin provides a Digital Twins Definition Language (DTDL) that enables to specify the structure of the digital twin and its data exchanged with the physical counterpart in a DTDL model. AWS, however, has a strong focus on edge computing and provides a software that can be deployed directly on edge devices to execute applications and interact with the cloud. How to integrate and reuse these functional features of the platforms is up to future endeavours.

Challenges arising from using our framework. When using our framework challenges for the three different roles participating in our method for engineering LCDPs for DTs arise. It is necessary that the roles of subsequent LCDP development steps communicate and exchange about requirements, and that they understand the respective modeling techniques employed by the development steps they are responsible. For the LCDP engineer to model the capabilities of the low-code development platform, she needs exchange with the digital

twin designer about what she is going to require to model current and future digital twins. Once the LCDP for DTs is generated, employing changes becomes challenging because of a necessary regeneration of the whole platform whenever MontiGem design-time models are changed or LCDP language plugins are added. For both, the LCDP engineer and the digital twin designer, it is necessary to understand the modeling techniques necessary to carry out their step of our method for engineering LCDPs for DTs. The DT designer again has to communicate and exchange with the domain experts about their requirements about the developed DT.

The digital twins

Each of the generated digital twins provides a web-interface which displays information about the physical system for domain experts and allows them to influence the digital twin as well as the CPPS.

Conflicting rules at run-time. If all competing models such as event or triggers that activate on the same values are executed at the same time, then there could be conflicting goals which makes it hard to decide which trigger should be executed. Approaches to solve this are *e.g.,* LIFO or FIFO approaches or the calculation of heuristics which rule might fit best. Another possibility would be to analyze and simulate the occurrence of events and triggers with realistic values before they occur and check if such conflicts exist. However, as different DSLs can be used during runtime, these checks have to implemented for each individual combination. Clearly, this is a research topics on its own.

Limitations of the Architecture. The presented architecture is just one possible implementation for a digital twin. Clearly, there exist other digital twin architectures with a different set of components. Our approach is extensible towards other implementations as the architectural model used within the LCDP and the LCDP language plugins which are provided in the LCDP could be changed. Moreover, the digital twin can be connected via the communication interfaces to digital twin services, e.g., for optimization, process mining or artificial intelligence on digital twin data, as suggested in [85]. This allows to provide more functionalities within the implementation.

Debugging during runtime of a digital twin. Being able to debug, trace, and replay the digital twin's behavior is necessary for experimentation, identification of problems and reconfiguration. There exist various means to provide debugging for modeling languages but none of these provide generic or generative web-based interfaces. The generated digital twin could be extended for such a service.

Intelligent assistive functionalities within DTs. Another functionality which would be important to include are intelligent assistive services to support human operators. Digital twins could assist operators in making the best possible decisions when handling an increasing amount and detail of information in production processes in a goal-oriented approach [50]. Assistive services are able to analyze a current action, identify next actions and suggest their execution [90] within production steps which are not fully automated. If the monitoring of human behavior is needed additionally to the monitoring of the physical object, models could be used to connect the DT with activity recognition systems [91], [92]. Further research is needed to discuss how to incorporate this functionality within the system architecture, what modeling languages could be used for support and how the digital twin cockpit could provide this functionality on different devices.

Digital twins as part of an ecosystem

One big DT vs. multiple separate DTs. A factory can have multiple areas where they have a multitude of CPPS and it might be feasible to have multiple separate DTs. The DT developer can decide what information should be included or how the parts are separated. A DT could contain as little as a single machine, a product line, or a complete factory. However, our current implementation still leaves some aspects unsolved: It is unclear how digital twins and their digital shadows should be composed, if an own digital twin exist for these different levels. These digital twins can communicate via the provided interfaces, however, it would be helpful to generate this communication in future.

X. Conclusion

Our approach to configure and generate digital twins is based on a sophisticated tool chain for the generation of information systems and an extensible base architecture for self-adaptive digital twins. Within the configurator, a digital twin designer selects the configuration for the DT. She can set up an own domain model or select standard types from libraries, add constraints, select services which should be available in the resulting DT and is able to define or select from available visualizations. The MontiGem generator is the used to generate one or more digital twins with the selected configuration. During run-time of the digital twin, communication parameters, event models, knowledge bases and user access control can be set and re-configured according to changing environmental contexts and needs.

Even though the proposed LCDPs facilitates creation and configuration of digital twins, it has to evolve due to requirements from practice. Although the approach already enables a lot of choice, communication techniques will evolve and so should the provided communication protocols to improve *interoperability* of the digital twin with different systems of various vendors. The complexity of the domain can make it necessary for different domain experts to participate in the development of a DT, which requires the LCDP to support *collaborative development* (in the presence of CPPS being controlled by the DTs). Also, various production systems are not fully automated which requires to keep the human-in-the-loop and to provide *human-centered assistive services* [50] along with other DT services. Different application areas in production might request different depths of, *e.g.,* reasoning or event models, which requires that the used modeling languages and language infrastructure must allow for *modularity*.

Such LCDPs empower domain experts without in-depth software engineering expertise to create running systems and to operate them. To use models and model-driven engineering methods enables abstraction, analysis and automation, which are crucial for the success of LCDPs.

ACKNOWLEDGEMENTS

The authors of RWTH Aachen University were supported 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.
The authors of Universität Stuttgart were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research-Foundation) under grant no. 441207927.

REFERENCES

[1] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[2] K. Shubenkova, A. Valiev, V. Shepelev, S. Tsiulin, and K. H. Reinau, "Possibility of digital twins technology for improving efficiency of the branded service system," in *2018 Global Smart Industry Conference (GloSIC)*. IEEE, 2018, pp. 1–7.

[3] G. Zhou, C. Zhang, Z. Li, K. Ding, and C. Wang, "Knowledge-driven digital twin manufacturing cell towards intelligent manufacturing," *International Journal of Production Research*, vol. 58, no. 4, pp. 1034–1051, 2020.

[4] N. Lauzeral, D. Borzacchiello, M. Kugler, D. George, Y. Rémond, A. Hostettler, and F. Chinesta, "A model order reduction approach to create patient-specific mechanical models of human liver in computational medicine applications," *Computer methods and programs in biomedicine*, vol. 170, pp. 95–106, 2019.

[5] X. Chen, E. Kang, S. Shiraishi, V. M. Preciado, and Z. Jiang, "Digital behavioral twins for safe connected cars," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, pp. 144–153.

[6] N. Karanjkar, A. Joglekar, S. Mohanty, V. Prabhu, D. Raghunath, and R. Sundaresan, "Digital twin for energy optimization in an smt-pcb assembly line," in *2018 IEEE International Conference on Internet of Things and Intelligence System (IOTAIS)*. IEEE, 2018, pp. 85–89.

[7] N. K. Chakshu, J. Carson, I. Sazonov, and P. Nithiarasu, "A semi-active human digital twin model for detecting severity of carotid stenoses from head vibration—A coupled computational mechanics and computer vision method," *International journal for numerical methods in biomedical engineering*, vol. 35, no. 5, p. e3180, 2019.

[8] Y. Lu and X. Xu, "A digital twin reference model for smart manufacturing," 12 2018.

[9] P. Sharma, H. Hamedifar, A. Brown, R. Green *et al.*, "The dawn of the new age of the industrial internet and how it can radically transform the offshore oil and gas industry," in *Offshore Technology Conference*. Offshore Technology Conference, 2017.

[10] I. Verner, D. Cuperman, A. Fang, M. Reitman, T. Romm, and G. Balikin, "Robot online learning through digital twin experiments: A weightlifting project," in *Online Engineering & Internet of Things*, M. E. Auer and D. G. Zutin, Eds. Cham: Springer International Publishing, 2018, pp. 307–314.

[11] G. Knapp, T. Mukherjee, J. Zuback, H. Wei, T. Palmer, A. De, and T. DebRoy, "Building blocks for a digital twin of additive manufacturing," *Acta Materialia*, vol. 135, pp. 390–399, 2017.

[12] L. Wright and S. Davidson, "How to tell the difference between a model and a digital twin," *Advanced Modeling and Simulation in Engineering Sciences*, vol. 7, no. 1, p. 13, 2020. [Online]. Available: https://doi.org/10.1186/s40323-020-00147-4

[13] B. R. Barricelli, E. Casiraghi, and D. Fogli, "A survey on digital twin: Definitions, characteristics, applications, and design implications," *IEEE Access*, vol. 7, pp. 167 653–167 671, 2019.

[14] J. Cabot, "Positioning of the low-code movement within the field of model-driven engineering," in *23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20. ACM, 2020.

[15] A. Gerasimov, J. Michael, L. Netz, B. Rumpe, and S. Varga, "Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems," in *25th Americas Conference on Information Systems (AMCIS 2020)*, ser. AIS Electronic Library (AISeL), B. Anderson, J. Thatcher, and R. Meservy, Eds. Association for Information Systems (AIS), 2020, pp. 1–10.

[16] M. Dalibor, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits," in *Conceptual Modeling*, ser. LNCS, vol. 12400. Springer, oct 2020, pp. 377–387.

[17] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems," in *23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems*. ACM, October 2020, pp. 90–101.

[18] P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, and A. Wortmann, "Model-Driven Development of a Digital Twin for Injection Molding," in *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, ser. Lecture Notes in Computer Science, S. Dustdar, E. Yu, C. Salinesi, D. Rieu, and V. Pant, Eds., vol. 12127. Springer International Publishing, June 2020, pp. 85–100.

[19] J. Michael and A. Wortmann, "Towards Development Platforms for Digital Twins: A Model-Driven Low-Code Approach," in *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*. Springer International Publishing, September 2021, pp. 333–341.

[20] D. V. Rosato and M. G. Rosato, *Injection molding handbook*. Springer Science & Business Media, 2012.

[21] F. Tao, H. Zhang, A. Liu, and A. Y. Nee, "Digital twin in industry: State-of-the-art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2018.

[22] C. Richardson, J. R. Rymer, C. Mines, A. Cullen, and D. Whittaker, "New development platforms emerge for customer-facing applications," *Forrester: Cambridge, MA, USA*, 2014.

[23] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, "End-user development: An emerging paradigm," in *End user development*. Springer, 2006, pp. 1–8.

[24] M. Tisi, J.-M. Mottu, D. Kolovos, J. De Lara, E. Guerra, D. Di Ruscio, A. Pierantonio, and M. Wimmer, "Lowcomote: Training the next generation of experts in scalable low-code engineering platforms," in *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*, 2019.

[25] R. Waszkowski, "Low-code platform for automating business processes in manufacturing," *IFAC-PapersOnLine*, vol. 52, no. 10, pp. 376–381, 2019, 13th IFAC Workshop on Intelligent Manufacturing Systems IMS 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896319309152

[26] G. Daniel, J. Cabot, L. Deruelle, and M. Derras, "Xatkit: A Multimodal Low-Code Chatbot Development Framework," *IEEE Access*, vol. 8, pp. 15 332–15 346, 2020.

[27] M. Bexiga, S. Garbatov, and J. a. C. Seco, "Closing the gap between designers and developers in a low code ecosystem," in *23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proc.*, ser. MODELS '20. ACM, 2020.

[28] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178.

[29] M. Di Maio, G.-D. Kapos, N. Klusmann, L. Atorf, U. Dahmen, M. Schluse, and J. Rossmann, "Closed-loop systems engineering (close): Integrating experimentable digital twins with the model-driven engineering process." in *2018 IEEE International Systems Engineering Symposium (ISSE)*. IEEE, 2018, pp. 1–8.

[30] A. Gurjanov, D. Zakoldaev, A. Shukalov, and I. Zharinov, "Formation principles of digital twins of cyber-physical systems in the smart

factories of industry 4.0," in *IOP conference series: materials science and engineering*, vol. 483, no. 1. IOP Publishing, 2019, p. 012070.

[31] H. Sun, C. Li, X. Fang, and H. Gu, "Optimized throughput improvement of assembly flow line with digital twin online analytics," in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2017, pp. 1833–1837.

[32] S. Settemsdal *et al.*, "Updated case study: The pursuit of an ultra-low manned platform pays dividends in the north sea," in *Offshore Technology Conference*. Offshore Technology Conference, 2019.

[33] M. Uzun, M. U. Demirezen, E. Koyuncu, and G. Inalhan, "Design of a hybrid digital-twin flight performance model through machine learning," in *2019 IEEE Aerospace Conference*. IEEE, 2019, pp. 1–14.

[34] Q. Qiao, J. Wang, L. Ye, and R. X. Gao, "Digital twin for machining tool condition prediction," *Procedia CIRP*, vol. 81, pp. 1388–1393, 2019.

[35] N. Alaei, A. Rouvinen, A. Mikkola, and R. Nikkilä, "Product processes based on digital twin," in *Commercial Vehicle Technology 2018*. Springer, 2018, pp. 187–194.

[36] C. Dufour, Z. Soghomonian, and W. Li, "Hardware-in-the-loop testing of modern on-board power systems using digital twins," in *2018 International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*. IEEE, 2018, pp. 118–123.

[37] T. Ruohomäki, E. Airaksinen, P. Huuska, O. Kesäniemi, M. Martikka, and J. Suomisto, "Smart city platform enabling digital twin," in *2018 International Conference on Intelligent Systems (IS)*. IEEE, 2018, pp. 155–161.

[38] F. Becker, P. Bibow, M. Dalibor, A. Gannouni, V. Hahn, C. Hopmann, M. Jarke, 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 *40th Int. Conf. on Conceptual Modeling (ER'21)*. Springer, 2021.

[39] A. Butting, B. Rumpe, and A. Wortmann, "Embedding Component Behavior DSLs into the MontiArcAutomaton ADL," in *Globalization of Modeling Languages Workshop (GEMOC'16)*, ser. CEUR Workshop Proceedings, vol. 1731, 2016.

[40] A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language," in *European Conference on Modelling Foundations and Applications (ECMFA'17)*, ser. LNCS 10376. Springer, July 2017, pp. 53–70.

[41] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2012.

[42] B. Rumpe, *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. [Online]. Available: http://www.se-rwth.de/mbse/

[43] T. Bolender, G. Bürvenich, M. Dalibor, B. Rumpe, and A. Wortmann, "Self-Adaptive Manufacturing with Digital Twins," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Computer Society, May 2021, pp. 156–166.

[44] K. Adam, J. Michael, L. Netz, B. Rumpe, and S. Varga, "Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project," in *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, ser. LNI, vol. P-304. Gesellschaft für Informatik e.V., 2020, pp. 59–66.

[45] A. Gerasimov, J. Michael, L. Netz, and B. Rumpe, "Agile Generator-Based GUI Modeling for Information Systems," in *Modelling to Program: 2nd Int. Workshop (M2P 2020), Revised Selected Papers*, ser. Communications in Computer and Information Science, vol. 1401. Springer International Publishing, 2021, pp. 113–126.

[46] T. Greifenberg, M. Look, S. Roidl, and B. Rumpe, "Engineering Tagging Languages for DSLs," in *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. ACM/IEEE, 2015, pp. 34–43.

[47] M. Dalibor, N. Jansen *et al.*, "Tagging Model Properties for Flexible Communication," in *MODELS 2019. Workshop MDE4IoT*. CEUR Workshop Proceedings, 2019, pp. 39–46.

[48] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*, ser. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[49] A. Gerasimov, P. Heuser, H. Ketteniß, P. Letmathe, J. Michael, L. Netz, B. Rumpe, and S. Varga, "Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend," in *Comp. Proc. of Modellierung 2020 Short, Workshop and Tools & Demo Papers*. CEUR Workshop Proceedings, 2020, pp. 22–30.

[50] J. Michael, B. Rumpe, and S. Varga, "Human behavior, goals and model-driven software engineering for assistive systems," in *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, A. Koschmider, J. Michael, and B. Thalheim, Eds., vol. 2628. CEUR Workshop Proceedings, 2020, pp. 11–18.

[51] J. Michael, L. Netz, B. Rumpe, and S. Varga, "Towards Privacy-Preserving IoT Systems Using Model Driven Engineering," in *MODELS 2019. Workshop MDE4IoT*, N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer, and A. Wortmann, Eds. CEUR Workshop Proceedings, 2019, pp. 595–614.

[52] A. Butting, J. Pfeiffer, B. Rumpe, and A. Wortmann, "A Compositional Framework for Systematic Modeling Language Reuse," in *Proc. of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 2020, pp. 35—-46.

[53] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic Composition of Independent Language Features," *Journal of Systems and Software*, vol. 152, pp. 50–69, June 2019. [Online]. Available: http://www.se-rwth.de/publications/Systematic-Composition-of-Independent-Language-Features.pdf

[54] ——, "Modeling Language Variability with Reusable Language Components," in *Int. Con. on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[55] B. Rumpe and A. Wortmann, "Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures," in *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, ser. LNCS 10760, Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, Ed. Springer, 2018, pp. 383–406.

[56] A. Butting, O. Kautz, B. Rumpe, and A. Wortmann, "Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution," *Journal of Systems and Software*, vol. 149, pp. 437–461, March 2019.

[57] T. Brockhoff, M. Heithoff, I. Koren, J. Michael, J. Pfeiffer, B. Rumpe, M. S. Uysal, W. M. P. van der Aalst, and A. Wortmann, "Process Prediction with Digital Twins," in *Companion Proc. ACM/IEEE 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS-C '21)*. IEEE, 2021.

[58] X. Li, B. He, Y. Zhou, and G. Li, "Multisource model-driven digital twin system of robotic assembly," *IEEE Systems Journal*, vol. 15, no. 1, pp. 114–123, 2020.

[59] R. Magargle, L. Johnson, P. Mandloi, P. Davoudabadi, O. Kesarkar, S. Krishnaswamy, J. Batteh, and A. Pitchaikani, "A simulation-based digital twin for model-driven health monitoring and predictive maintenance of an automotive braking system," in *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, no. 132. Linköping University Electronic Press, 2017, pp. 35–46.

[60] P. Munoz, J. Troya, and A. Vallecillo, "Using uml and ocl models to realize high-level digital twins," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, 2021.

[61] F. Bordeleau, B. Combemale, R. Eramo, M. van den Brand, and M. Wimmer, "Towards model-driven digital twin engineering: Current opportunities and future challenges," in *International Conference on Systems Modelling and Management*. Springer, 2020, pp. 43–54.

[62] A. M. Madni, C. C. Madni, and S. D. Lucero, "Leveraging digital twin technology in model-based systems engineering," *Systems*, vol. 7, no. 1, p. 7, 2019.

[63] J. I. Panach, O. Dieste, B. Marín, S. Espaã, S. Vegas, O. Pastor, and N. Juristo, "Evaluating model-driven development claims with respect to quality: A family of experiments," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 130–145, 2021.

[64] G. Fulya and T. Gabriele, "Using microsoft powerapps, mendix and outsystems in two development scenarios an experience report," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, 2021.

[65] mgm technology partners GmbH. (2021) Widget showcase. [Online]. Available: https://a12.mgm-tp.com/showcase/#/

[66] Appian. (2021) Low-code application development. [Online]. Available: https://appian.com/platform/low-code-development/low-code-application-development.html

[67] Bonitsoft. (2021) Bonita platform. [Online]. Available: https://www.bonitasoft.com/bonita-platform

[68] M. Henkel and J. Stirna, "Pondering on the key functionality of model driven development tools: The case of mendix," in *International Conference on Business Informatics Research*. Springer, 2010, pp. 146–160.

[69] Microsoft. (2021) Microsoft power apps and microsoft power automate. [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/business/microsoft-powerapps

[70] K. Adam, J. Michael, L. Netz, B. Rumpe, and S. Varga, "Model-Based Software Engineering at RWTH Aachen University," in *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, ser. LNI, vol. P-304. Gesellschaft für Informatik e.V., 2020, pp. 183–188.

[71] ServiceNow. (2021) Now platform. [Online]. Available: https://www.servicenow.com/now-platform.html

[72] R. Martins, F. Caldeira, F. Sá, M. Abbasi, and P. Martins, "An overview on how to develop a low-code application using outsystems," in *2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*. IEEE, 2020, pp. 395–401.

[73] Pega. (2021) Pega platform. [Online]. Available: https://www.pega.com/products/platform

[74] Quickbase. (2021) Quickbase platform. [Online]. Available: https://www.quickbase.com/product/product-overview

[75] Salesforce. (2021) Salesforce platform. [Online]. Available: https://www.salesforce.com/products/platform/overview/

[76] L. Brunschwig, R. Campos-López, and E. G. J. de Lara, "Towards domain-specific modelling environments based on augmented reality," in *2021 IEEE/ACM 43nd International Conference on Software Engineering (ICSE)*, 2021, pp. 335–346.

[77] A. Colantoni, L. Berardinelli, and M. Wimmer, "Devopsml: Towards modeling devops processes and platforms," in *23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proc.*, ser. MODELS '20. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3417990.3420203

[78] P. Kourouklidis, D. Kolovos, N. Matragkas, and J. Noppen, "Towards a low-code solution for monitoring machine learning model performance," in *23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proc.*, ser. MODELS '20. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3417990.3420196

[79] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[86] A. N. Johanson and W. Hasselbring, "Hierarchical combination of internal and external domain-specific languages for scientific computing," in *European Conference on Software Architecture Workshops (ECSAW '14)*, U. Zdun, Ed. New York, USA: ACM Press, 2014, pp. 1–8.

[80] F. Bordeleau, B. Combemale, R. Eramo, M. van den Brand, and M. Wimmer, "Towards model-driven digital twin engineering: Current opportunities and future challenges," in *Systems Modelling and Management*, Ö. Babur, J. Denil, and B. Vogel-Heuser, Eds. Cham: Springer International Publishing, 2020, pp. 43–54.

[81] T. Vogel and H. Giese, "Requirements and assessment of languages and frameworks for adaptation models," in *Models in Software Engineering*, J. Kienzle, Ed. Springer, 2012, pp. 167–182.

[82] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable DSMLs based on fUML," in *International conference on software language engineering*. Springer, 2013, pp. 56–75.

[83] B. Combemale and M. Wimmer, "Towards a model-based devops for cyber-physical systems," in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, 2019, pp. 84–94.

[84] R. Jolak, M. Savary-Leblanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard, and M. R. Chaudron, "Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4427–4471, 2020.

[85] A. C. Bock and U. Frank, "In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms," in *Companion Proc. ACM/IEEE 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS-C '21)*. IEEE, 2021.

[87] J. Gray and B. Rumpe, "The evolution of model editors: browser- and cloud-based solutions," *Software and Systems Modeling*, vol. 15, no. 2, pp. 303–305, 2016.

[88] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Collaborative repositories in model-driven engineering [software technology]," *IEEE Software*, vol. 32, no. 3, pp. 28–34, 2015.

[89] 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.

[90] K. Hölldobler, J. Michael, J. O. Ringert, B. Rumpe, and A. Wortmann, "Innovations in Model-based Software and Systems Engineering," *The Journal of Object Technology*, vol. 18, no. 1, pp. 1–60, July 2019.

[91] H. C. Mayr, J. Michael, V. A. Shekhovtsov, S. Ranasinghe, and C. Steinberger, "A Model Centered Perspective on Software-Intensive Systems," in *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, ser. CEUR Workshop Proceedings, M. Fellmann and K. Sandkuhl, Eds., vol. 2097. CEUR-WS.org, May 2018, pp. 58–64.

[92] V. A. Shekhovtsov, S. Ranasinghe, H. C. Mayr, and J. Michael, "Domain Specific Models as System Links," in *Advances in Conceptual Modeling Workshops (ER'18)*. Springer International Publishing, November 2018, pp. 330–340.