Process-Aware Digital Twin Cockpit Synthesis from Event Logs

Dorina Bano[1], Judith Michael[2], Bernhard Rumpe[2], Simon Varga[2] and Mathias Weske[1]

[1]Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
Email: dorina.bano@hpi.de, mathias.weske@hpi.de

[2]Software Engineering, RWTH Aachen University, Aachen, Germany
Email: michael@se-rwth.de, rumpe@se-rwth.de, varga@se-rwth.de

Abstract—**The engineering of digital twins and their user interaction parts with explicated processes, namely process-aware digital twin cockpits (PADTCs), is challenging due to the complexity of the systems and the need for information from different disciplines within the engineering process. Therefore, it is interesting to investigate how to facilitate their engineering by using already existing data, namely event logs, and reducing the number of manual steps for their engineering. Current research lacks systematic, automated approaches to derive process-aware digital twin cockpits even though some helpful techniques already exist in the areas of process mining and software engineering. Within this paper, we present a low-code development approach that reduces the amount of hand-written code needed and uses process mining techniques to derive PADTCs. We describe what models could be derived from event log data, which generative steps are needed for the engineering of PADTCs, and how process mining could be incorporated into the resulting application. This process is evaluated using the MIMIC III dataset for the creation of a PADTC prototype for an automated hospital transportation system. This approach can be used for early prototyping of PADTCs as it needs no hand-written code in the first place, but it still allows for the iterative evolvement of the application. This empowers domain experts to create their PADTC prototypes.**

Index Terms—**Process-Aware Digital Twin Cockpit, Low-Code Development Approaches, Sensor Data, Event Log, Process Mining, Process-Awareness**

I. Introduction

A. Motivation and Relevance

The development, maintenance, and evolution of digital twins are still challenging research areas [1]. The original system typically comes along with a high complexity and within the engineering process of digital twin information from different disciplines is needed, e.g., (1) experts from the main domain such as civil engineering [2], avionics [3], injection molding [4], laser cutting [5], or healthcare [6], (2) people knowledgeable of the related IT systems and databases (which might be the experts from the main domain or related departments), (3) experts knowledgeable of further relevant context information such as legislation, financial affairs, business processes, or maintenance, and (4) people knowledgeable of the software engineering processes of digital twins. This results in heterogeneous views on an original system which have to be harmonized in the requirements analysis of a digital twin, e.g., via integrated, consistent collections of heterogeneous models [1].

Current research on the engineering of digital twins is moving towards more automation in the engineering process but still requires tight cooperation between domain and digital twin engineering experts with a software engineering background. Methods like model-driven software engineering and its executable models help transitions from concrete implementations for one physical object in one concrete domain towards more generalized development methods [7]. Some of these approaches might overlap with low-code development approaches, which aim to reduce the amount of hand-written code [8] and shift the power to domain experts. These methods enable domain experts to take an active role in the engineering process of the digital twin.

Whereas full automation in the engineering of digital twins and all relevant services might be a big vision, a first reachable goal is the automated engineering of Process-Aware Digital Twin Cockpits (PADTCs). A PADTC is the user interaction part of a digital twin, which provides functionality to handle explicated processes of the physical object and its' context. As any (also process-unaware) digital twin cockpit, it provides the Graphical User Interface (GUI) for visualizing data organized in digital shadows [9], models and the GUI for interaction with services of the Digital Twin (DT). As DTs digitally represent material [10], [11] and immaterial [12], [13] objects and processes from the real world, the PADTC allows to access, adapt, add information, and monitor and partially control the physical system. These relevant processes are reflected in existing data of the related software systems but are often not explicitly visible.

To support the engineering of PADTC, the process model as a main artifact in business process management [14] is needed. One option to discover such a model is process discovery, which is one of the main techniques in process mining. Any process mining technique requires as input an event log that is usually extracted from the information systems, which in case of digital twins are the software systems accompanying the physical object. Furthermore, such event logs can be extracted from sensor data, which offers already a lot of information about the physical object.

B. Research Question and Objectives

The main research question addressed in this paper is: *how to create a process-aware digital twin cockpit from event logs?*

Our objectives are (1) to propose a (semi-)automated process which allows for agile increments starting from a prototype to a full system, and (2) show its realizability in a prototype. Moreover, we show how process mining facilitates the creation of PADTCs for digital twin engineers.

C. Main Contribution

Our main contributions are a low-code development approach (Section III) that takes an event log as an input and generates a process-aware digital twin cockpit. We have developed a prototype for an automated hospital transportation system (Section V) to show the realizability of this approach and make use of automation techniques to improve this process. The low-code development approach is validated and it matured on this realization. In detail, our research contributions are:

- a definition for digital twin cockpits and process-aware digital twin cockpits,
- a method to extract event logs from sensor data, the extension of an existing method to infer a domain model from event logs [15] for the identification of necessary data types, and a method to identify roles from event logs,
- a method for a model-to-model transformation from (pre-processed) BPMN, domain, and tagging models to GUI and data models,
- a process on how to integrate the different transformation steps into an automated process, and
- an intensive discussion on how to further automate these steps as well as challenges and further extensions.

As showing the realizability of research concepts is as important for us as the method itself, our practical contribution includes the engineering of the whole process including a prototype for a process-aware digital twin cockpit. Our lessons learned during this process influenced the description of the method and resulted in additions, e.g., the step pre-processing of discovered models. We show examples of concrete pre-processing steps of models in the prototype. We had to incorporate existing methods to extract BPMN models from event logs in Python into a Java-based automation process in the prototype, which improved our understanding of a realizable system architecture. We have made profound adaptions of the existing MontiGem [16] generator framework to be able to handle process models during runtime. In detail, we implemented additions to the two generators as well as in the runtime environment.

D. Relation to previous work

Within two visionary papers [17], [18], we have already presented concepts for low-code development platforms of digital twins and the engineering of process-aware digital twin cockpits, however, the method presented in Section III is yet unpublished. Overlapping between all these papers is (1) the use of the MontiGem generator framework for cockpits but each paper with own additions needed, (2) the MontiCore Domain-Specific Languages (DSLs) used as they provide a common basis, and (3) our digital twin definition. Paper [17] presents the vision for a low-code development platform for digital twins in a 2-step generation process but (1) does not include any process mining techniques or event logs, and (2) creates models manually or by choosing them from a model library. [17] has a focus on language composition and model reuse, whereas this paper uses a data-driven automation approach. Paper [18] presents how process mining and model-based digital twins could cooperate with a specific focus on the concept of digital shadows. The differences between paper [18] and this article are that (1) this article focuses on process-aware digital twin cockpits whereas [18] focusses on process mining in interaction with self-adaptivity services in a MAPE-K loop of digital twins, and (2) this article provides automatic transformations from sensor data to models and the running system, whereas [18] presents design time models without considering their automatic transformation from other information sources. Moreover, this article provides a concrete prototype in contrast to the two visionary papers.

Paper Organization. Section II briefly discusses the main concepts and notions. We introduce a running example to explain the approach in Section IV. Section III describes the main steps to generate the digital twin from an event log. Section V shows an application use case for our low-code approach. Section VI summarizes the related work and the last section concludes this paper.

II. Preliminaries

We introduce key terms and concepts to provide a foundation for understanding the paper.

A. Background and Definitions

We provide details why we position ourselves as low-code development approach and details about the digital twin, digital twin cockpit and process-aware digit twin cockpit definitions we use.

1) Low-Code: The term low-code was initially introduced by Richardson and Rymer [19] in the context of achieving faster alternatives to implementing customer-facing apps instead of using traditional development platforms. Low-Code Development Platforms (LCDPs) need no or only a reduced amount of hand-coding to create an entirely operational application. This allows for the fast delivery of applications and helps to test business ideas with working code within days or weeks [19]. To allow for customization at a higher level of abstraction, LCDPs include graphical, textual, or form-based modeling environments.

Other low-code approaches besides extensive LCDPs are, e.g., low-code development approaches which facilitate the development of software applications with reduced code [8] These approaches might use models, data stored in schema-less XML/JSON documents, or relational databases.

Following the categorization of model-driven engineering (MDE) and low-code by Di Ruscio et al. [8], *we can position* *ourselves on level two, which describes low-code development approaches using models*. This level is an overlap between a set of low-code approaches with a set of MDE approaches that have in common that they use machine-processable models, aim to reduce the amount of hand-written code, e.g., via code generation or interpretation but are not offering deployment or lifecycle management capabilities for the created system.

Within this paper, we propose a *low-code development approach* to facilitate the creation of digital twins from event logs, which are upfront extracted from the sensor data. Using modeling and automation reduces the hand-written code dramatically and allows for faster delivery of applications. The applications we focus on within this paper are process-aware digital twin cockpits.

2) Digital Twins and Digital Twin Cockpits: There exists a plethora of definitions for digital twins,however, there is little consensus about what a DT actually is [20]. The main weaknesses about the available definitions are that they are:

1) ambiguous, by deferring to another undefined term, such as a "virtual representation" [21], "a virtual projection of the industrial facility into the cloud" [22], or a "computable virtual abstraction" [23];

2) narrow, by focusing on specific use cases, domains, or technologies, such as a "digital model of the real network environment" [24]; "product avatar" [25] which includes only the product perspective and not machines, processes, or material; or a "virtual representation based on AR-technology" [26];

3) underspecified such as [27], where a digital twin is mainly defined via its automatic data flow from virtual to physical object but falls short on discussing the role of models and additional information, characteristics, and functionalities a digital twin has over its physical object; or

4) utopian, due to all-encompassing aspirations, such as an "integrated virtual model of a real-world system containing all of its physical information" [28], or a "complete digital representation" [29].

As none of these definitions fit, researchers in the German cluster of Excellence "Internet of Production", which involves about 200 participating researchers from more than 30 colocated institutes from different disciplines, such as engineering, material science, computer science, economics, and social sciences, and more than 50 industrial partners, have come up with the following definition:

Definition 1 (Digital Twin [30])**.**
A ***digital twin*** *of a system consists of*
- *a set of models of the system,*
- *a set of digital shadows, and*
- *provides a set of services to use the data and models purposefully with respect to the original system.*

To be able to create a digital twin requires that we have observable elements in the physical world that can be monitored, sensed, or actuated and controlled.

The *set of models* helps to understand the system in the physical world, e.g., structure, behavior, physical, geometrical or mathematical models. These models can be, e.g., used to create the digital twin or to compare the current status of the twin with planned states.

Each digital shadow in the *set of digital shadows* includes "a set of contextual data-traces and/or their aggregation and abstraction collected concerning a system for a specific purpose with respect to the original system" [9].[1] Following this definition, a digital shadow is a passive set of data [31] which is an information source about a system's state and history. The shadows are collected, filtered, and reduced for their purpose in varying forms of abstractions and are purely digital artifacts produced by a (physical) system. We can use, e.g., process mining algorithms for aggregating digital shadows from observed data [31]. Moreover, shadows may contain information from different perspectives, e.g., systems (physical and organizations), processes, products, and humans [32].

The provided *services* during runtime of the digital twin might include process mining, artificial intelligence, simulation and predictive control services. Process mining techniques can be used within DTs as services to further improve and adapt the used knowledge. Artificial intelligence services help to realize real-time decisions and explainable AI within decision support helps to reduce human errors in decision making [33]. Services to control the physical object need to send execution commands via APIs to the physical object and are often related to self-adaptiveness [34].

To visualize relevant information, an interface for domain experts is needed, which leads us to the term *digital twin cockpit*:

Definition 2 (Digital Twin Cockpit)**.**
A ***digital twin cockpit*** *is the user interaction part (UI/GUI) of a digital twin. It provides the graphical user interface for visualizations of its data organized in digital shadows and models, and the interaction with services of the digital twin, and thus enabling humans to access, adapt and add information and monitor and partially control the physical system.*

A cockpit is, by definition, a part of the digital twin, and can be seen both as a special service provided by the digital twin and an integrative front-end component for various specific services that the digital twin provides. A cockpit visualizes various forms of data, which includes, e.g., digital shadows, any form of data received from third-party systems, all kinds of data and commands entered by the humans using the cockpit and models of the physical system or the operation processes of the physical system.

A specific kind of a digital twin cockpits is a *PADTC*:

[1]For more details about digital shadows, we refer the reader to [9] which presents a conceptual model, gives details on each concept and provides an example from the production domain.

Definition 3 (Process-Aware Digital Twin Cockpit)**.** *A* ***process-aware digital twin cockpit*** *is a digital twin cockpit that additionally provides functionality to handle explicated processes of the physical object and its' context.*

A PADTC is a digital twin cockpit but has a stronger focus on processes, which are explicitly defined using appropriate process definition languages. A PADTC knows the allowed processes (in the form of models), the current status of these processes (in the form of status data), and the history of these processes states and executed actions (in the forms of data lakes). Processes describe the steps needed to be carried out by (1) the physical object, (2) the digital twin, or (3) are expected to be executed by the context, which includes humans and other physical objects respectively their digital twins. A process may have several active participants, but not all of those need to be participating in each process definition, which allows various forms of automation of processes. Thus, a process step is executed automatically by the physical object, the digital twin (only), which means it is a form of data processing. In addition, it is executed by physical objects in the context, is executed by humans involved, or is executed by humans involved using the physical object.

To realize digital twins and their cockpits is an increasingly complex task that leads us to the idea of increasing the degree of automation for creating them.

B. Technologies Used

We provide an overview of event logs and their use in process mining, and introduce the generator framework that we use as the basis for generating digital twin cockpits.

1) Event Logs: The central artifact of any process mining technique is an event log. The event log is defined as a collection of events. Each event contains at least: 1) an attribute called *case identifier*, necessary to distinguish between different business process instances; 2) an *activity name*, essential to know the steps followed in the process in order to achieve a particular business goal; and, 3) the *timestamp*, which indicates the time occurrence of each event [14]. Besides that, an event log might have different optional attributes, such as the resource, which indicates the entity responsible for executing the activity or the department under which the activity is executed. All events pertaining to the same *case identifier* constitute an execution trace or a business process instance. The association of events with the case identifier establishes a perspective called *case notion* [35].

In traditional process mining, these event logs are extracted from the database or data warehouses of a given organization [36]. Therefore, the extraction process handled by the process mining experts requires a clear definition of the case notion [37]. Once the extraction process takes place, the event logs are tailored to different process mining techniques like process discovery, conformance checking, and process improvement [38]. A process discovery technique takes as input an event log and by applying one of many discovery algorithms (e.g., Alpha Miner [39], Heuristic Miner [40], Inductive Visual Miner [41]) outputs a process model. In contrast, conformance checking is used to check whether events recorded in the event log conforms to the process model or vice versa. Finally, the performance of the process model can be improved by analyzing the bottlenecks. A collection of different process mining techniques is made available in a workbench tool called ProM [42].

The databases of running information systems are not the only source for deriving such logs. Sensor data are another source of information to extract the event log [43]. In this case, the event log can provide insights regarding the activities of people, machines, and the way how they behave in a specific environment. However, discovering the process model of human behavior from sensor data is a challenging task that is caused by the gap between the sensor data, and the event log structure [44]. In addition, the sensor data have no explicit process notion awareness like activity name and execution instance. Therefore, we aim to address this challenge and use the extracted event log to automate the construction of a PADTC.

C. Digital Twin Cockpit Generation

Within [30], we have used the generator framework MontiGem [16] to create a digital twin cockpit. The generated DT cockpit relies on a client-server infrastructure and presents relevant information via different GUIs. Figure 1 shows the generation process using a variety of models and the resulting system architecture of a DT cockpit in combination with a set of services of the DT. The services communicate with the physical object, its data sources, and 3rd party applications such as related information systems via defined interfaces.

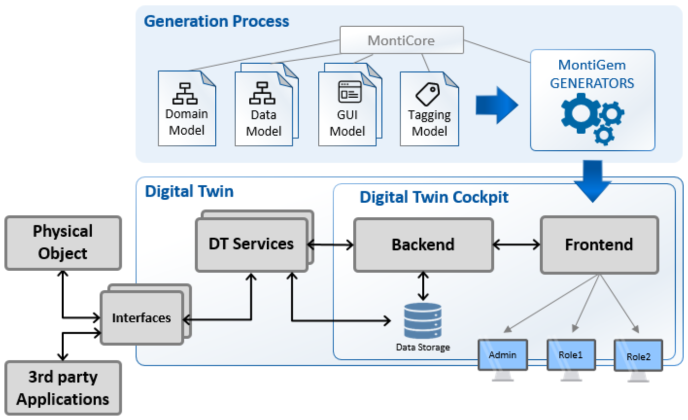


Fig. 1: Generating Digital Twin Cockpits with MontiGem

MontiGem [30] takes a set of textual models as input and creates a running information system. To define the models, we use DSLs created in the language workbench and development tool framework MontiCore [45]. The input models include two models using the UML/P class diagram notation: The *domain model* used to define concepts of the application domain resulting in the latter data structure and several *data* *models* which represent a subset of this domain model and define views on this data for each GUI. To describe graphical user interfaces, MontiGem can understand *GUI models* which are defined using a specific DSL [46]. MontiGem also allows to add optional models: Restrictions, e.g., for validation of input data, can be modeled using *Object Constraint Language (OCL)* and additional information such as platform-specific details or rights and roles could be added to the domain model using *tagging models*. Such tagging models [47] can be used to add additional (technical) semantic information onto concepts of existing models without changing the original models, e.g., tag an attribute of a class diagram with database technology-specific information.

The following approach is based on using event logs in an automated way to create process-aware digital twin cockpits. Our first steps were in the production domain, however, it soon became apparent that our approach could be applied to other application domains where physical objects are monitored, and such data exists.

III. Generating Process-Aware Digital Twin Cockpits from Event Logs

This paper aims to facilitate the engineering process of digital twins by increasing the degree of automation. In more detail, we aim to generate process-aware digital twin cockpits from event logs, which are upfront extracted from the sensor data.

Our approach includes *two main roles*: The digital twin engineer who follows the suggested approach and creates a process-aware digital twin cockpit and the domain or process expert who works with the PADTC during the runtime of the application.

From the perspective of a digital twin engineer, our PADTC creation process includes three phases (preparation, generation, adaption) and its generation result at runtime, which we consider as the fourth phase.

Figure 2 gives an overview of the suggested approach. The following description of the **four phases** (in the figure right) references the numbering of the steps in the figure. Each of these steps is discussed in more detail in the following subsections. *In Section V we present a prototype using the use case from Section IV. It covers steps 1-11 in Figure 2.*

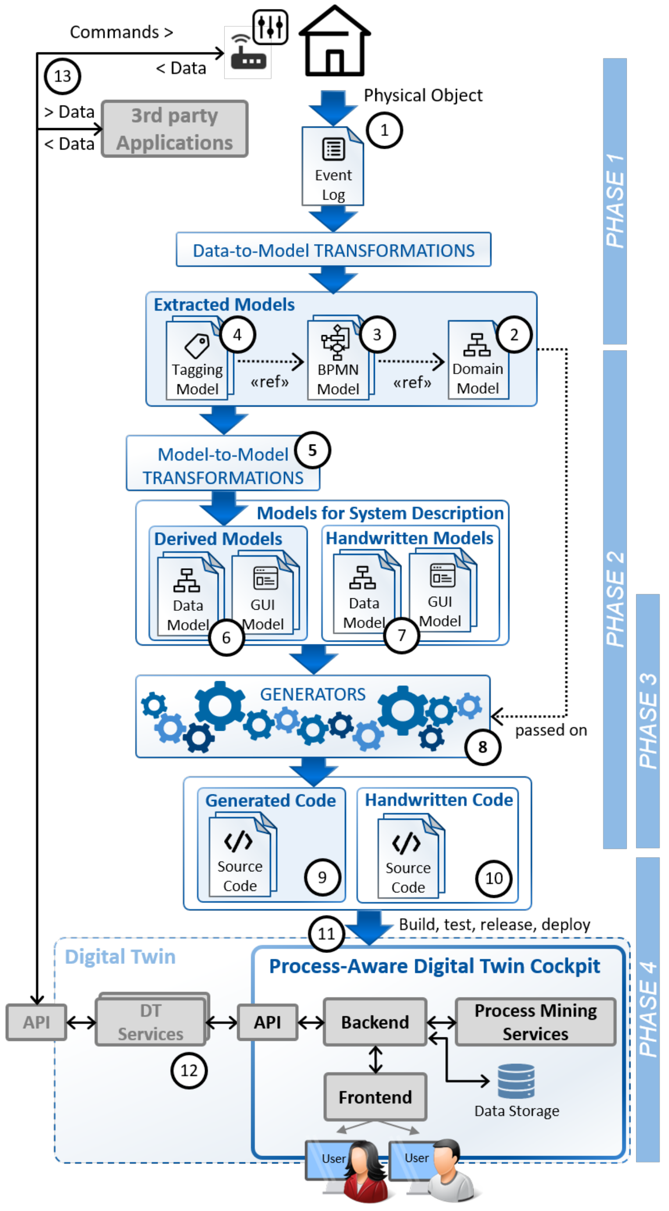


Fig. 2: Overview of the generation process and resulting Process-Aware Digital Twin Cockpit

Phase 1: Preparation. In the *preparation phase*, (1) the sensor data from a physical object is used to extract an event log, which is later used to discover (2) the domain information (e.g., Domain Model in Figure 2), (3) process models, and (4) relevant roles (e.g., Tagging Model in Figure 2) via data-to-model transformations.

Phase 2: Generation. In the *generation phase*, the models ((2), (3), and (4)) derived from the previous phase are used as an input within step (5) in Figure 2) a model-to-model transformation to create data models (views) and GUI models (6) representing the different process tasks. A generator (8) in Figure 2 could be used to synthesize (9) the source code of the process-aware DT cockpit, also referred to as model-to-code transformation.

Phase 3: Adaption. In the *adaption phase*, it is possible to make additions to the generation process and the digital twin cockpit. To allow for adaptability, additional handwritten models, (7) in Figure 2 could be used as an input for the generator (8). Handwritten code (10) can be added to the backend and frontend of the generated PADTC as an extension of the generated code (9) using the TOP-mechanism.

Phase 4: Runtime. The PADTC is built within step (11) of Figure 2, which is then tested, released, and deployed. Afterward, the connection to relevant digital twin services (12) is made available. At runtime, live data (13) from the physical object or third-party applications are used by the DT, and the DT can influence the physical object via commands. The domain users can interact with the physical object and the digital twin services via the process-aware digital twin cockpit.

A. Phase 1: Preparation

This phase aims to discover all relevant information from sensor data and transform it into models usable for generative approaches (data-to-model transformations).

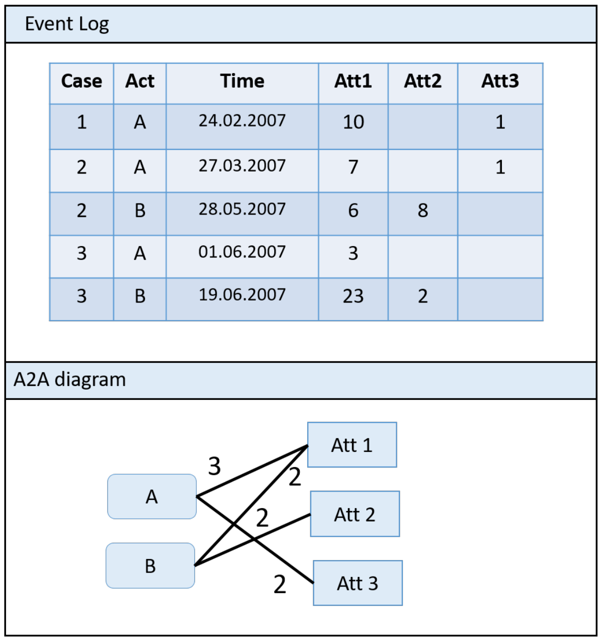
1) Event Log Extraction from Sensor Data: Extracting an event log from the sensor data can be a challenge as they are not explicitly aware of process notions like activity name and execution instance. Therefore, the challenge that needs to be tackled is to correlate the sensor data to specific process instances by having in mind a clear business goal. The fact that each process instance execution pertains to one case in the event log and each case is defined as a collection of ordered events implies the need to specify the core attributes of each event: case identifier, activity name, and timestamp.

Depending on the format of the sensor data and the business goal other approaches like in [44], [48] can be used to extract an event log.

2) Inferring the Domain Model from Event Log: Within this low-code development approach for PADTC generation we have to ensure the existence of a data structure. Therefore, we are using the previously extracted event log to infer a domain model that is later used to create the data structure capable of storing and managing a collection of data values. One could use the approach presented in [15] to discover a domain model represented as UML class diagram from an event log. The approach groups the event log attributes into the domain model classes by generation first an intermediate representation called *Activity-Attribute relationship* (A2A) diagram (see Figure 3). This diagram stores the relationship between the activities and attributes captured in the event log. Besides that, the number of access occurrences for each relation is calculated, representing the number of times an access relation between an activity and attribute holds.

An example of the A2A diagram derived for an example event log is illustrated in Figure 3. For example, the activity *A* is accessing the *Att1* three times and *Att2* two times. While *Att1* and *Att2* is accessed by the same number of times (2) from the activity *B*.

The domain model generation considers as input the A2A diagram and based on the relation between two or more attributes a set of predefined rules are applied to judge whether these attributes belong to the same domain model class or not. In addition, the relations between the generated classes



Case	Act	Time	Att1	Att2	Att3
1	A	24.02.2007	10		1
2	A	27.03.2007	7		1
2	B	28.05.2007	6	8	
3	A	01.06.2007	3		
3	B	19.06.2007	23	2	

Fig. 3: An example of the A2A diagram inferred from the event log [15]

and multiplicity is defined based on the number of relations between the activities and attributes in the A2A diagram.

For generative approaches, it is additionally interesting to identify data types when inferring Class Diagrams (CDs) from the event log. However, the approach presented in [15] neglects the data type attributes in the discovered domain model. Therefore, it could be extended in order to identify the data type for each attribute. This allows handling the domain model in an automated way and representing the GUIs data according to its type. To solve this, for each attribute inferred in the data model class the data type is defined by parsing all data values in detail.

It is complex to unambiguously identify the exact type in given data as it can be written in various styles. E.g., the natural number (integer) `712` can be written by the simple version `712` as a long value `712L`, an octal value `01310`, or a hex value `0x2C8`. Such different writing styles might also occur in event logs.

A simple approach to identify data types is to investigate existing data values in the event log, distinguish between numbers and strings, and further check if it is an integer or float. The pseudo-code in Listing 1 depicts a simple logic to recognize integer, float, date, string, and enumeration types. In addition, it is possible to subdivide them further to support more specific types, such as byte and long. The identification of *enum* variants (see Listing 1, line 10) is possible by checking if the values of an attribute can be grouped in a fixed set of possible values. The identification is possible with a threshold of 5 or 10 different values.

Listing 1: Sketch of the type recognition algorithm

```
if (isNumber) {
    if (isFloatingPoint) {
        => float
    } else {
        => int
    }
} else if(time or date format) {
    => date OR time
} else { // string
    if (isEnumeration) {
        => enum
    }
    => string
}
```

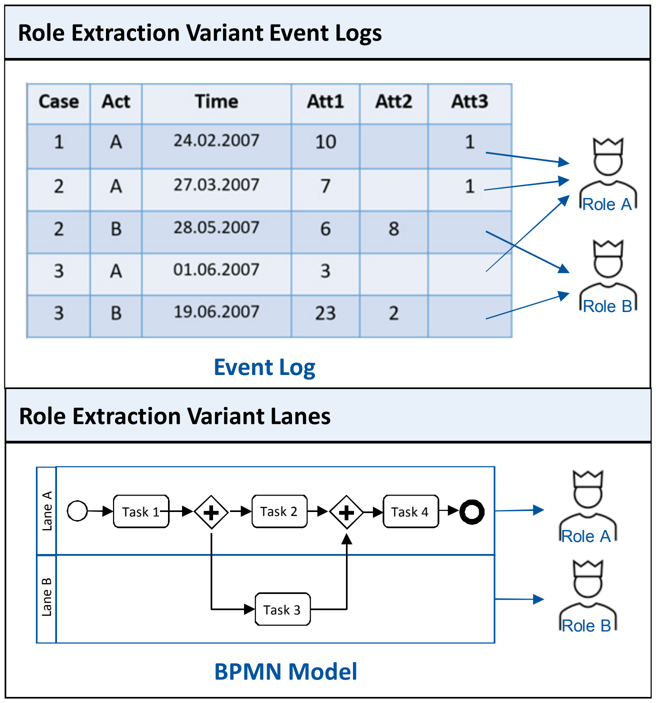
The type recognition is just a semi-automatic process such that it provides recommendations for the data model attributes. Those types always depend on the attribute values of the corresponding event log. In the end, a domain expert together with the digital twin engineer must decide if the data structure extraction is done correctly. The identified data types are added to the domain model and used to generate the process-aware digital twin cockpit.

3) Extracting BPMN Models: The generation of the process-aware digital twin cockpit requires the process model. Different process mining techniques (e.g., Inductive Visual Miner [41], Heuristic Miner [40]) exist to discover a process model from an event log in the process mining area. Our language of choice to show processes in digital twins is Business Process Model and Notation (BPMN) [49], [50]. For measuring the quality of the discovered process model, one of the existing quality metrics can be used (e.g., replay fitness, precision) [51]. The first one quantifies the degree to which the discovered process model can reproduce the event log traces. In contrast, precision measures the fraction of the behavior that is possible in the discovered process model, but it can not be seen in the event log.

4) Extracting Roles for a Tagging Model: We use the information provided in the event log to automatically extract roles for the PADTC, which reduces the manual effort to achieve such extraction. One option is to make use of the tagging models [52], used to access specific data onto concepts in the domain model. The tagging models allow us to add additional information to existing models while keeping the original models, in this case, the domain model, untouched. Within the generation process, roles in tagging models define possible roles in the generated process-aware digital-twin cockpit.

Figure 4 shows the principle process about how to get a tagging model for roles. For an automated approach, there exist two possible ways to identify roles:

- The first extraction possibility depends on which information is included in the event log: If specific user groups could be identified from the event log, e.g., by using a role discovery algorithm [53] which groups actors into roles, this information can be used for creating the tags.

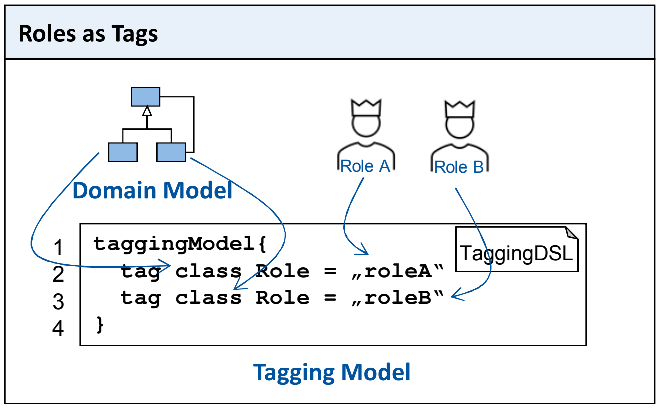


Case	Act	Time	Att1	Att2	Att3
1	A	24.02.2007	10		1
2	A	27.03.2007	7		1
2	B	28.05.2007	6	8	
3	A	01.06.2007	3		
3	B	19.06.2007	23	2	

Fig. 4: Variants of role extraction

- The second approach only works if the BPMN model includes lanes: Each lane in a BPMN model defines different user groups and can be extracted as a role with the lane name as the role name.

In a second step (see Figure 5) these identified roles are used to add tags to classes from the domain model. We know which class to tag with which role as we use information from the event logs: Actors are related to specific attributes, and we have discovered in the first step to which class each attribute belongs to.



```
taggingModel{
  tag class Role = „roleA“
  tag class Role = „roleB“
}
```

Fig. 5: How to create a tagging model with roles and information from the class diagram

Following this method, we now have a domain model, one or more BPMN models, and a tagging model, which can be used in the next step – the generation of the PADTC.

B. Phase 2: Generation of the PADTC

The overall generation process is split up into two main parts, including a model-to-model and a model-to-code transformation (see Figure 2). The model-to-model transformation ((5) in Figure 2) uses the models discovered from an event log ((2), (3), and (4) in Figure 2) to generate additional models (e.g., data models, GUI models (6)). While the generator (8) syntheses the source code (9) of the process-aware digital twin cockpit from models (model-to-code transformation) and provides a runnable PADTC.

1) Preprocess Discovered Models: Dependent on the discovery algorithms in the former steps, it might be necessary to transform the discovered models to another representation in order to use it as an input for a generator. For example, PM4PY [54] creates a BPMN in XML format, and the transformation uses another textual BPMN DSL as input. Another option is to change the BPMN XML format to fit the input format specified by a process engine such as Camunda[2]. To the best of our knowledge, most tools are not compatible in the first place. Therefore, we expect this step to take place in several implementations.

2) Discover Additional Models: Our aim in this step is to provide all needed models to start the generation of the PADTC. Therefore, we need a model-to-model transformation, which takes the discovered and preprocessed models, e.g., domain, BPMN, and tagging model in Figure 2, as input and creates data models and GUI models.

For each activity in the BPMN model, one GUI model can be generated. This GUI model allows us to represent the data needed within this activity graphically, provides input fields for data input, and ensures navigation possibilities to mark an activity as completed. One data model is created for each GUI model, which defines the necessary data to be shown and stored in a corresponding GUI. This data always represents a subset of the concepts in the domain model and can be seen as similar to views used in the database.

With these models in place, we can move further and start the generation process of the PADTC.

3) Code Synthesis: The code for the PADTC is synthesized from the provided models. This requires (1) to have or create a code generator which is able to handle the different models, namely the domain, process, role tagging, GUI and data models and (2) to provide a runtime environment (RTE) to generate the code into, as not all software code is generated. This runtime environment should be able to handle process models during runtime in order to be process-aware.

A PADTC is typically realized as a client-server architecture with a web-interface for users. This requires to generate the data structure within a database, the persistence logic in the backend, the GUIs in the frontend, and the communication infrastructure between backend and frontend.

[2]https://camunda.com/products/camunda-platform/bpmn-engine

The discovered process models contain information relevant for the source code of the PADTC. Participants of the business process and the tagging models (steps (3) and (4) in Figure 2) can be used to create roles and test users in the generated application, while the data model (6) and the domain model input (2) should be used to create the data structure of the database and the communication between frontend and backend of the PADTC. The discovered process models should be additionally added as data into the generated application, this way they can be used during runtime.

The synthesized PADTC code from the provided models requires: (1) the creation of the code generator that can handle different models, namely the domain, process, role tagging, GUI, and data models; (2) to provide a runtime environment to generate the code into, as not all software code is generated. This runtime environment should be able to handle process models during runtime in order to be process-aware.

A PADTC is typically realized as a client-server architecture with a web interface for users. Therefore, it is necessary to generate the data structure within a database, the persistence logic in the backend, the GUIs in the frontend, and the communication infrastructure between backend and frontend.

The discovered process models contain information relevant for the source code of the PADTC. Participants of the business process and the tagging models (steps (3) and (4) in Figure 2) can be used to create roles and test users in the generated application, while the data model (6) and the domain model input (2) should be used to create the data structure of the database and the communication between frontend and backend of the PADTC. The discovered process models should be additionally added as data into the generated application. In this way, the discovered process can be used during runtime.

The result of the generation process is a PADTC with basic GUIs for the discovered processes. The PADTC should be a runnable application that allows domain experts to validate the discovered processes, roles, and data structures. The models and generated code can be manually extended by the digital twin engineer in the next step.

C. Phase 3: Adaption

To apply this approach in practice, the infrastructure has to adapt and extend the generation process and the resulting application. In our approach, this can be achieved in two ways: 1) by using (6) the generated model and (7) additional hand-written models as an input for (8) the generator; 2) by adding (10) handwritten code as an extension to (9) generated code.

1) Add Handwritten Models: A digital twin engineer can add handwritten models, e.g., with domain information, for graphical user interfaces and additional data models to further customize the process-aware digital twin cockpit.

The additional domain model contains information, which is useful for the domain and visualizations but could not be extracted from the event log, e.g., attributes that are relevant during runtime but were not captured in the event logs. Such additions could also include structures to be able to handle some runtime models, e.g., the concepts of the meta-model of digital shadows or basic process structures [18]. Defining them in addition to the domain model enables this approach to generate the concepts in the data structure of the resulting PADTC. This enables us to store the models during the runtime of the application.

GUI models should be customizable for the specific use-case and come along with related data models for representing the relevant part of the data. Typical added GUI models are, e.g., domain-specific dashboards with statistical information or user interfaces containing pictures or other media showing the physical object, which the digital twin stands for.

2) Add Handwritten Code: Further adaptations can be accomplished by adding handwritten code extensions ((10) in Figure 2). To remain compatible with the generated code, handwritten code is extending generated code. Typical handwritten additions are artifacts that occur only once in the code and are not interesting for generation processes. Usually, they refer to business logic, e.g., the duration calculation from two given points in time or summarizing data for specific categories. Other relevant additions are in the case of a PADTC connectivity information to digital twin services, which means the implementation of according APIs, e.g., REST[3], WebSockets[4], or MQTT[5].

After all, additions are added, the PADTC (11) can be automatically built and tested. After some manual usability checks, the digital twin can release and deploy it.

D. Phase 4: Runtime

During runtime, data in the PADTC (11) can be initialized. In addition, the cockpit is connected via APIs to DT services and able to receive and handle live data via them. The users during runtime are the domain or process experts.

1) Initialize Data: Typically, the generated system comes along with an empty database. As we already have event logs containing physical objects (1), the resulting PADTC can be automatically (initially) populated with information from the event logs, namely the historical data. Additionally, the roles and test users for each role can be initialized.

2) Runtime Models: The PADTC includes the technology to handle process models during runtime, e.g., via a process engine. During runtime, we can add additional runtime models, e.g., instances of digital shadows or a process model for planned processes (cf. [18]). Which runtime models could be added during the application's runtime dependent on which DT services should be connected with the PADTC. These could be, e.g., AI services, Process Mining services, or services of a self-adapting loop. As an example, if we connect it with the self-adapting MAPE-K loop presented in [30], we should be able to define event models or case-based reasoning models during runtime.

3) Live Data: As we know from the event logs, which data is interesting for us, the relevant data structures exist in the PADTC, and we have already generated the backend functionality to store or update data. This infrastructure is used for the data received from the physical objects during runtime.

It is possible to dynamically visualize the live data in the graphical interfaces, e.g., via regularly updated charts. This depends on the information defined within the GUI models or what was added in the handwritten code. Moreover, it is interesting to have the functionality to visualize a process model step by step using live data during execution and to compare the live data with the already discovered process models.

In addition, based on the live data, the discovered process model from the extracted event log can change over time, and its visualization helps the domain or process expert get process models updates based on these changes. The low-code development approach presented in this paper can be applied to different application domains.

IV. Use Case: Hospital Transportation System

We use the following use case inspired by a real-world healthcare process to explain our approach further. Transportation of patients is an essential aspect outside as well as within hospitals [55], [56]. Suppose that the patients are admitted for diagnoses and treatments into a hospital. During the time in the hospital, the patient movements are monitored based on the sensors applied to each wheelchair or bed. Whenever a patient arrives at the hospital a wheelchair or a hospital bed is occupied by him, which is then used to capture its movement in the hospital premises.

One wheelchair or bed can be assigned only to one patient. The patient has to move through different departments to receive the necessary treatments from different specialized doctors. Suppose that each sensor of the wheelchair or bed signals whenever the patient enters and leaves a department. This way, it is possible to track and monitor the movements of all patients from the moment they are admitted to the hospital till the moment they are discharged.

As it is illustrated in Figure 6, the patient arrives at the hospital and occupies a wheelchair with sensor number 1 and visits department A. In this department, the patient will be treated by doctor A. The sensors attached to the wheelchair and the department's doors keep track of the department name whenever the patient enters and leaves the corresponding department. The patient might need to visit several departments for further diagnoses and treatments, and all his movements are tracked based on the corresponding sensors.

The emitted sensor data are stored in some text files and contain the following information (see Table I): the time when

[3]https://restfulapi.net
[4]https://websockets.spec.whatwg.org
[5]https://mqtt.org

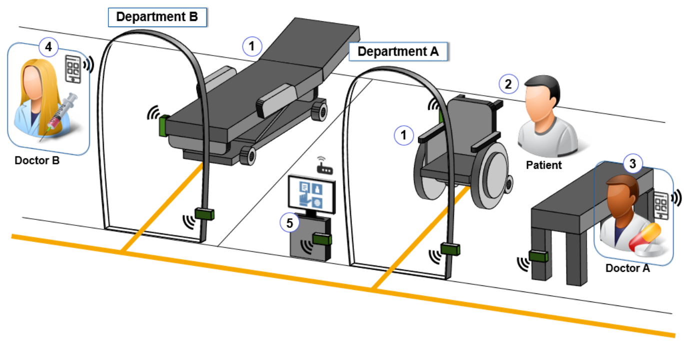


Fig. 6: An automated transportation system for patients

the patient enters (i.e., in-time column) and leaves (i.e., out-time column) the department, the sensor id used to identify the activated sensor. Based on this information, it is possible to derive the department identification number (i.e., used to identify between several departments within the hospital) because the configuration of sensors is known beforehand.

in-time	out-time	department id	sensor id	...
12/7/2021 9:49:00	12/7/2021 11:06:22	Emergency Department	B23	...
13/1/2021 7:32:34	13/1/2021 10:07:52	Neurology	A03	...
13/1/2021 11:08:34	13/1/2021 11:52:52	Emergency Department	B23	...
13/1/2021 7:32:34	13/1/2021 8:07:52	Medicine	A07	...
13/1/2021 9:32:34	13/1/2021 12:07:52	Medicine	A07	...
...	...	...	...	...

TABLE I: Example of sensor data

If the hospital aims to create a PADTC, various disciplines have to share effort in the engineering phase: We need experts from the medical department and experts from the IT and maintenance department. They have to analyze and identify relevant data, design the PADTC, which includes to manually extract the needed data structures from existing applications, manually describe the relevant processes, e.g., using process modeling languages, design the system architecture of the PADTC and start implementing its main components, including the visualization, process management, and data exchange. After several iterations and feedback from domain users, the PADTC can be used. The following section shows how to reduce the time needed for the first steps by increasing the automation and reducing hand-written code.

V. Prototype

We realize the presented approach using a) data from the MIMIC III database [57], b) models of DSLs created in the language workbench and development tool framework MontiCore [45] and c) the generation framework MontiGem [16]. The generated application is a client-server architecture. The used programming languages are Java for the backend, TypeScript and HTML within the Angular 6 framework for the frontend.

A. Phase 1: Preparation

To build a prototype of our low-code approach, we are using the MIMIC III database [57] which contains electronic healthcare records of patients admitted to the critical care unit at the Beth Israel Deaconess Medical Centre (BIDMC). Each entry in the database table is considered as one event triggered from the sensor (in the same format as in Table I).

Event Log Extraction from Sensor Data. One of the earliest steps in our approach is to extract an event log from the sensor data. The format of the sensor data (the same as the one explained in Section IV) is necessary but not sufficient to identify such event attributes because they contain domain activation time stamps (general case), and patient-related information is missing. In addition, this information is about the incoming, outgoing time, and department identification number (Table I). The sensor identification number can be derived from the department number because it is known which senor belongs to which department. Besides that, it is necessary to set up mapping with the patient data. Such data might include the patient identification number, name, age, gender. This connection is created whenever the patient is admitted to the hospital and occupies the sensor wheelchair or bed and is lost whenever the patient is discharged from the hospital in order to allow for a new connection when a newly admitted patient occupies the same seat. In this way, we will be able to append the patient-related information to the raw sensor data and use such information to extract an event log.

The selection of the case notion strongly depends on the domain, and the business process model goal [58]. Considering the format of our sensor data and our final goal, we select the *Patient Admission* as a case notion of our event log. In this way, each case in the event log contains events regarding the patients' visits to the respective departments. The same patient can be admitted several times into the hospital; therefore, a new case is defined in the event log for each admission.

One option for identifying the event log activities from the sensor data is considering the department names. As for the timestamp of the event, the time tracked by the sensor (e.g., in-time, out-time) can be used, representing the time when the patient enters and leaves the corresponding department. Events related to the time when the patient is admitted (e.g., the same time when the patient occupies the wheelchair or bed) or discharged from the hospital are named the same for each event log case: the first event is called *admit*, and the last one is called *discharge*. For each pair of events, each pertaining to one department, the event with the earliest timestamp is the named admit plus the corresponding department name, and the latest one is named discharge plus the department name, which corresponds to the time when the patient is admitted and discharged from the department. In addition, information related to the patients can be added as optional attributes in the event log.

Following the approach presented in Section III an example of the extracted event log is illustrated in Table II. The *case* *id* equal to 1 contains 4 events. Each activity represents the department that the patient has visited from the moment they are admitted (i.e., the first event starts with *admit*) to the moment they are discharged (the last event is called *discharge*) from the hospital. Therefore, the first event in each case starts with *admit* and the last one ends with *discharge*. The *admit* event corresponds to the time when the patient arrives at the hospital and occupies a wheelchair or hospital bed. In contrast, the *discharge* event corresponds to the time when the patient is discharged from the hospital and released from the wheelchair or hospital bed.

The event log extracted from the MIMIC database contains 25.463 events and 6.455 cases executed during one year. For security reasons, the time in the MIMIC database is anonymized. Therefore, the events pertaining to our event log belong to the year 2153.

Following the next step of our approach, the domain model for the previously extracted event log has to be discovered.

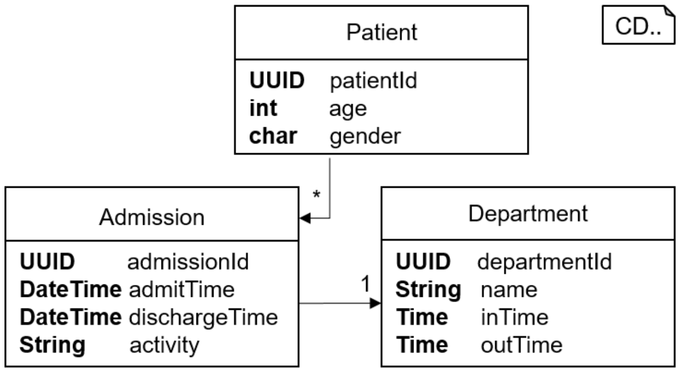


Fig. 7: Excerpt of the class diagram describing the use-case

Inferring the Domain Model from Event Log. The domain model is discovered from the event log extracted from the MIMIC database. The initial discovery defines each attribute as String data type, which is in principle usable in the generation process. To support addition datatypes and give it is required to run the type recognition algorithm described in Listing 1. Figure 7 shows the data structure, including the discovered data types needed for the digital twin generation represented as a UML class diagram in a graphical notation. The defined structure can be read as follows: 1) a patient can have any number of admissions 2) an admission always happens at a specific department 3) an admission corresponds to a visit of a specific patient in a department (as denoted by the given cardinalities). Supporting different data types allows users to specify different input fields, e.g., data time, text, integer.

During our prototype implementation, we use a textual notation for the discovered domain model (Figure 7) in the UML/P [59] DSL Class Diagrams for Analysis (CD4A), which includes the concepts class, attribute, and association. These concepts are sufficient to generate the data structure in the PADTC database.

Extracting BPMN Models. Once the event log is extracted from the sensor data in the next step, the process model is discovered by applying the Inductive Visual Miner algorithm [41]. We are using the PM4Py [60] library to discover the process model from the event log. Our language of choice for represent the process model is BPMN [50], [61] (see Figure 8). For simplicity reasons, the threshold of the discovered process model is set to 90%.

The process starts with the *admit* activity, which represents the time when the patient is admitted to the hospital and occupies the wheelchair or bed. Afterwards, the patients is either admitted to the *ED Emergency Department* or *Labor & Delivery*. The discharged patient from ED Emergency Department (e.g., *discharge ED Emergency Department*) are admitted to the Medicine department (e.g., *admit Medicine*). While, the patients discharged from the *Labor & Delivery* department are moved to the *Obstetrics (Postpartum & Antepartum)* for further postpartum treatment. Finally, the patient is discharged, and the wheelchair or the hospital bed is made available for another patient.

Since the quality metrics strongly depend on the algorithm used to discover the process model and the variety of variables involved in the process [62], we argue that finding the best discovery algorithm is out of scope for this application scenario.

Extracting Roles for a Tagging Model. To specify which roles should be able to fulfill a certain task in the digital twin, we can extract a tagging model from either the BPMN model or the event log. Our event log does not explicitly include actors but participating departments, so our extraction algorithm assumes that one role for each department exists. Additionally, one role for the group of patients is added. The example tagging model is depicted in Listing 2, which denotes the name of the tagging model and the BPMN model that is getting tagged (l. 1 `Transportation`, the model in Figure 8). Then each line (ll. 2-4) contains a tag `Role` with a value of what role is associated with what part of the model.

Listing 2: Roles tagging model

```
1 tags ResourcesTags for Transportation {
2  tag Transplant Role = "transplantDoctors";
3  tag EmergencyAction Role
4        = "emergencyDoctors";
5  tag Patient Role = "patient";
6 }
```

B. Phase 2: Generation

Within this phase, we first have to preprocess the discovered models to use them in the transformation, transform them, and then use them for the generation of the digital twin cockpit. We already have a first prototype of the PADTC at the end of this phase.

case id	activity	time	age	gender
1	admit	22/03/2153 10:02	22	F
1	admit ED Emergency Department	22/03/2153 12:30		
1	discharge ED Emergency Department	22/03/2153 14:33		
1	admit Medicine	22/03/2153 22:44		
1	discharge Medicine	22/03/2153 23:45		
1	discharge	22/03/2153 23:59		
2	admit	13/08/2153 01:07	56	F
2	admit Labor-Delivery	13/08/2153 02:36		
2	discharge Labor-Delivery	13/08/2153 15:33		
2	admit Obstetrics(Postpartum-Antepartum)	13/08/2153 16:04		
2	discharge Obstetrics(Postpartum-Antepartum)	14/08/2153 20:44		
2	discharge	15/08/2153 22:14		

TABLE II: Simplified example of an event log extracted from the sensor data

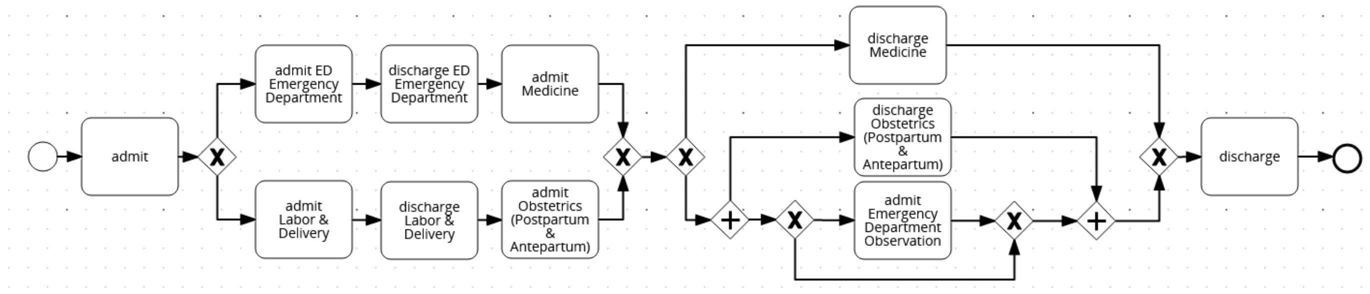

admit
admit ED Emergency Department
discharge ED Emergency Department
admit Medicine
admit Labor & Delivery
discharge Labor & Delivery
admit Obstetrics (Postpartum & Antepartum)
discharge Medicine
discharge Obstetrics (Postpartum & Antepartum)
admit Emergency Department Observation
discharge

Fig. 8: The discovered process model illustrated as BPMN after applying the Inductive Visual Miner algorithm. For simplicity reason several activities are excluded from the process model

Preprocess Discovered Models. As expected, the BPMN model discovered using PM4PY in XML format is not directly compatible with, e.g., the Camunda BPMN XML structure. Thus, we have to transform this BPMN XML model automatically into Camunda BPMN models and another textual representation, namely our own MontiCore [45] BPMN DSL.

The transformation requires a preprocessing step as we use parts of the Camunda library to access the input BPMN XML. The structure of the discovered BPMN model from PM4PY includes a `diagram` and a `process` part. The Camunda library expects them in the opposite order, resulting in the need to swap these parts in the XML. Another aspect was, e.g., that PM4PY does not include the direction of gateways which refers to whether the gateway is a split (called diverging in the XML) or join (called converging in the XML). Counting the incoming and outgoing connections can be automatically detected and added in the XML.

The transformation to our own MontiCore BPMN DSL starts with reading the BPMN XML into an internal data structure (similar to the meta-model of the BPMN language), parsing it to the target language, and printing the transformed model. This has the negative aspect that large models require more memory store but has the advantage that the order of elements in the input BPMN XML is irrelevant, the printing process can be optimized as we know how many elements of which type should be printed and the internal structure is reusable if other transformations will be implemented in future. The transformation creates a textual BPMN model in the MontiCore BPMN DSL.

Discover Additional Models. The domain, tagging, and transformed BPMN models are then used as input models within the *model-to-model transformation* step. The resulting models are a GUI and a data model for each of the activities in the BPMN model. We also derive an additional class diagram for commands that specifies the process-specific commands, e.g., the command to start the respective process. To realize the model-to-model transformation, we use FreeMarker[6] templates that take relevant information from the input models from the created abstract syntax tree and replace defined placeholders in the templates.

Table III gives an overview of the transformations for mapping BPMN models to GUI models. Figure 9 illustrates the generated GUI structure from a user task in a BPMN model based on these transformations. That is, transformation 1 defines that one GUI model is generated per user task in the BPMN model. The generated GUI model consists of a card element as known from most UI frameworks. Transformation 2 defines that the name of the user task, `taskName`, is used as the heading of this card element. Additionally, the task name is a part of the name of the GUI model file, the name of the webpage element in the GUI model and the name of generated input and output forms which are needed in Angular for user input and output.

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

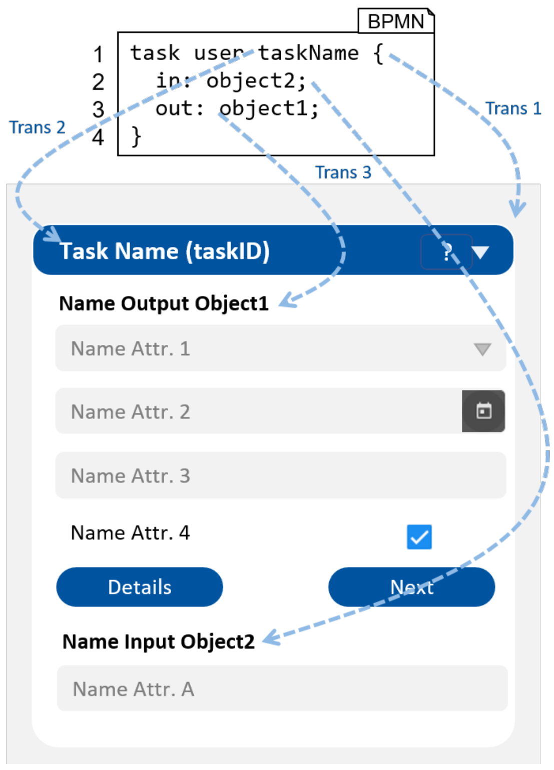


Fig. 9: Relationship between the GUI and the BPMN model

Nr	Concept in BPMN	Realization in the GUI
(1)	task with type user task	one GUI model per task
(2)	task name	integrated in the name of the GUI model file, name of the webpage element, label on top of the first card in the GUI model, name of input and output forms
(3)	input and output objects of one user task	name as heading, name of the generated domain form group template (see Figure 10)
(4)	attributes of input and output objects of one user task	a field for each attribute including support for datatypes, e.g., a dropdown for enums, calendar for dates
(5)	split gateway	method to navigate to the next task based on attribute values

TABLE III: Transformation from BPMN concepts to GUI

With transformation 3, the names of the input and output objects, `object1` and `object2`, are used as the headings of the generated form groups of these objects within the card element. The types of these objects, namely `Object1` and `Object2`, are further used as the names of the generated domain form group templates, as shown in Figure 10.

With transformation 4, for each of the attributes of the input and output objects, e.g. `attr1`, `attr2`, `attr3`, and `attr4` of `object1` and `attrA` of `object2`, one respective input field is generated considering its data type. For example, text input is generated for a simple text, a dropdown input is generated for an enumeration, a checkbox is generated for booleans, or a calendar is generated for the date attribute. Transformation 5 defines that a split gateway in a BPMN model is used to generate a method to navigate to the next task based on attribute values. The method is called when the according button is pressed.

In our approach, we use a 2-step generation process to generate the GUI models (see Figure 10): Firstly, we use a handwritten `FormGroup.ftl` template which instantiates handwritten field templates and generate generates domain form group templates for each in and out object of the task (see Figure 10, two generated `DomainFormGroup.ftl` templates marked in red according to the in and out object types in the BPMN model in Figure 9). Each `DomainFormGroup.ftl` covers the fields of one input or output object needed in the UI.

Secondly we are generating the GUI model: For each user task, one GUI model is created using the `TaskForm.ftl`. For each input and output of the task, the `TaskForm.ftl` instantiates a template matching the type of the data element, e.g., for the output of a task with type Object1, the template `Object1.ftl` is instantiated. The same occurs for the input with type Object2 and the template `Object2.ftl`. The template `Object1.ftl` is made up of the fields of the class Object1, and corresponding field templates are added, e.g., three times a `TextField.ftl` with the corresponding data type of the input field and one `CheckBox.ftl`. The fields from the input data are disabled if they are not to be changed within this activity.

The buttons in Figure 10 are defined in the `TaskForm.ftl` and call methods within the backend of the PADTC. If these task GUIs are just used to step through the process, a simple next button is enough to navigate to the next task. If the process steps should also be replayable within the digital twin, a *claim*, *disclaim* and *complete* task should be included, which is then reflected in the task list of the current user.

A BPMN model might also include aspects that are not reflected in a GUI, e.g., service tasks that are performed without user interaction. In this case, we generate a Java delegate implementation, which the developer must extend to provide the task's business logic. The process engine can call this implementation.

Moreover, this model-to-model transformation generates a commands.cd, which includes classes to start new instances of a process and complete user tasks, and a data model is generated, which includes process-specific Data Transfer Objects (DTOs) for the inputs and outputs of each user task. These DTOs can be sent between the backend and frontend of the process-aware digital twin cockpit.

Adaptions within the runtime environment of the application. Our generated code has a connection to manually written code, predefined components, and the runtime environment, altogether also referred to the target system [45]. To handle processes during the runtime of the application,

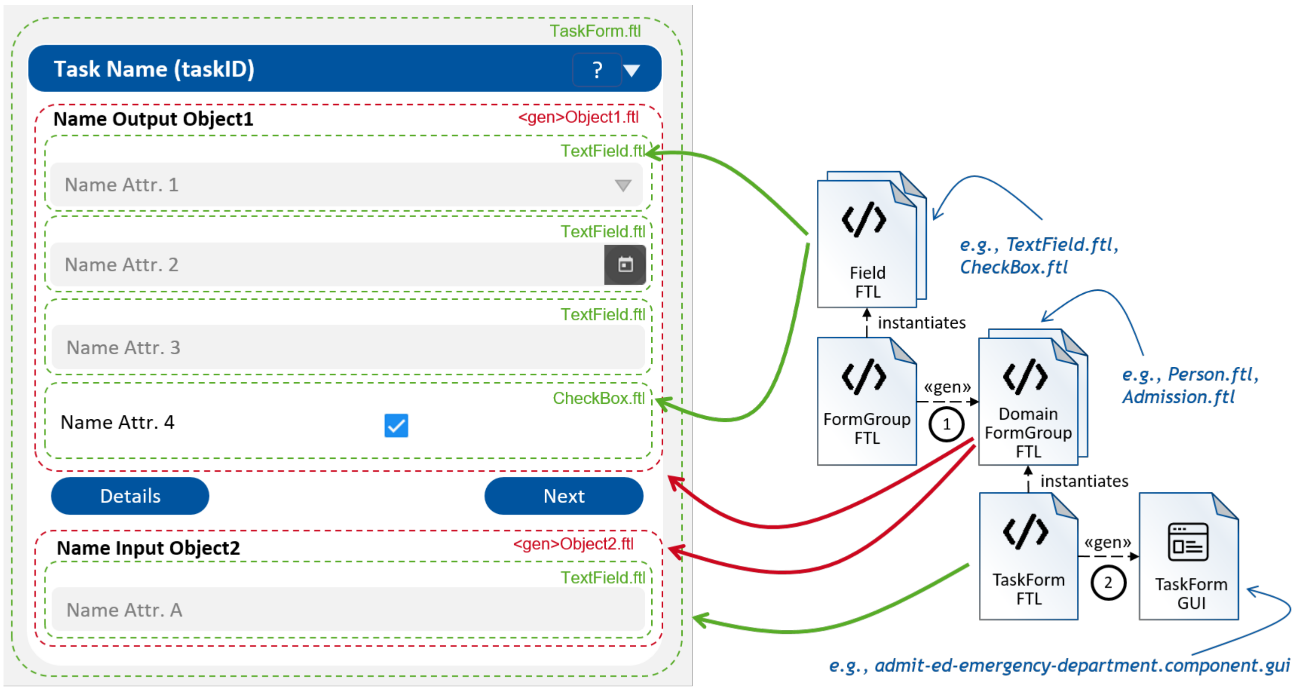


Fig. 10: Structure of generated GUI models with the FreeMarker template structure

we had to make some additions within the RTE of the target system, namely our process-aware digital twin cockpit. These additions were only made once and remain for every process-aware digital twin cockpit independent from the application domain.

As we want to use process models during the application's runtime, we had to add a process engine in the RTE. In the current implementation, we are using the Camunda process engine[7]. However, this might also be replaced by handwritten code, which can handle processes when no predefined component should be used. Therefore, it is possible to use the BPMN models in XML format during runtime of the application, which are discovered in the data-to-model transformation from event logs and are preprocessed.

Another aspect that is needed in every PADTC is the visualization of task lists. Our current realization of the runtime environment includes code and models, which exist in every implementation. We have added the corresponding GUI models for task lists, including running and completed tasks.

Code Synthesis. Having this initial set of models prepared, we can start the first model-to-code generation iteration. The MontiGem generator takes the models as input, transforms them into an internal representation which is transformed into an output internal representation, uses templates for the Java backend and the TypeScript and HTML frontend, and generates the code (see [63] or [46] for details). The generated system without hand-written additions consists of the database, the backend infrastructure (to handle database objects and data shown in the frontend), the communication infrastructure (between backend and frontend), and some basic views for the process steps.

[7]https://camunda.com/products/camunda-platform/bpmn-engine/

C. Phase 3: Adaption

As hand-written additions for this evaluation, we have added maintenance and event log meta-data to the domain model, added a picture of the floor plan in .svg, created GUI models and data view models for the dashboard, and some other pages. By adding hand-written models and hand-written code (see (7) in Figure 2), more complex visualizations are possible.

Add Handwritten Models. The additions to the domain model are all separate class diagram files, which will be merged before they are used as input for the generator framework. Listing 3 shows an excerpt of the additions in a domain model regarding the meta-structure of an event log. The attributes can be identified from the event log used at the beginning of this approach.

Listing 3: Meta-Structure of Event Logs as addition to the domain model

```
classdiagram eventlogs extends domain {
  class EventLog {
    int caseId;
    String activity;
    ZonedDateTime time;
    int age;
    String gender;
  }
9 }
```

Listing 4 shows an excerpt of the additions in a domain model regarding maintenance tasks for devices. The devices (l.3-6) have a particular type (l.8-10) and are related (l.12) to one concrete maintenance plan (l.19-24). Each maintenance plan includes several operations (l.30). These operations (l.19-24) define what has to be done in which interval (l.26-28) and under which regulation. Each of these operations can be executed (l.36). In this case, the start and end times of the execution will be stored together with a derived attribute that calculates the duration of the execution (l.32-37). Aspects such as changes in maintenance plans and operations have to be considered together with a more complex regulatory structure. These aspects are omitted here to keep the example compact.

Listing 4: Maintenance of devices as addition to the domain model

```
1  classdiagram maintenance extends domain {
2
3    class Device {
4      int deviceId;
5      DeviceType type;
6    }
7
8    enum DeviceType {
9      WHEELCHAIR, BED;
10   }
11
12   association [1] Device <->
         ↪ MaintenancePlan [1];
13
14   class MaintenancePlan {
15     int mPId;
16     String description;
17   }
18
19   class MaintenanceOperation {
20     int mOId;
21     String description;
22     MaintenanceInterval neededInterval;
23     String regulation;
24   }
25
26   enum MaintenanceInterval {
27     DAILY, WEEKLY, MONTHLY, ANNUALLY;
28   }
29
30   association [1] MaintenancePlan <->
         ↪ MaintenanceOperation [*];
31
32   class MaintenanceExecution {
33     int mEId;
34     ZonedDateTime startTime;
35     ZonedDateTime endTime;
36     /long duration;
37   }
38
39   association [1] MaintenanceOperation <->
         ↪ MaintenanceExecution [*];
40 }
```

Handwritten Tagging Models. It is also possible to add further information such as privacy or security concerns with tags. In our current implementation, it is possible to define data restrictions in the generated activity forms, e.g., disabling and hiding individual form fields. To include this feature, an additional tagging model is required.

Handwritten GUI Models. It might be necessary to also include the handwritten GUI models for the dashboard of the application, e.g., as shown in the excerpt in Listing 5. In a GUI model, we have to define which data is needed (l.2) to be shown, e.g., in a data table: In l.12, we use the entries objects in the object TransportDataTable to get each row with its data. We can add layout information, e.g., container (l.4) or rows (l.17) and use predefined graphical components, e.g., a card with head and body (l.5,6,9), labels (l.7), data tables (l.11-16) or buttons (l.18-20). The data table (l.11-16) has columns for each relevant attribute, where we can specify the presented name in the GUI and the name of the attribute as it is called within our data model. For buttons (l.18-20), it is possible to define a method executed when the button is pressed. The concrete implementation of this method has to be added as handwritten code, as only the method declaration is generated.

Listing 5: Excerpt of the handwritten GUI model for the dashboard

```
1  webpage Dashboard(
2   all TransportDataTable tdt, ...){
3   ...
4   container(45%){
5    card {
6     head {
7      label "Transport Devices in Operation"
8     }
9     body {
10     ...
11     datatable "TransportData" {
12      rows < tdt.transportDataEntries {
13       column "ID", id;
14       column "Location", location;
15       column "Duration", time(duration);
16     }}
17     row(r){
18      button "Historical Transport Data" {
19       click -> navigateToEventLogPage()
20      }
21     }
22  }}}
23 }
```

Handwritten Code for the Application. The current implementation of the generator framework MontiGem requires some handwritten additions, e.g., the logic to load and aggregate data from the database into the objects transferred to the frontend for visualization. Other additions were needed in the method bodies in the frontend, which should be executed when pressing a button.

Furthermore, the current runtime environment includes APIs using REST and WebSockets as technologies. If it is needed to connect our process-aware digital twin cockpit with DT services, we can add handwritten code, which helps us to handle transferred and received data in the backend of the PADTC. To test this, it would be possible to implement, e.g., a mock third-party application that sends maintenance information.

To be able to *use process mining algorithms during runtime*, we have added an additional docker container that can run Python code and provides a Flask REST API to access its functionalities. The event log data is stored in the database of the PADTC and can be accessed by the Python implementation via the backend to run process mining algorithms on this data. We have implemented, e.g., the visualization of the Directly-Follows-Graph (DFG) from the given event log data, both visualized as an SVG and as a table with filtering options.

We had to add handwritten GUI models for handling the process mining results during the application's runtime. This included two pages, one showing the SVG of the DFG and one for the textual representation. If a user opens the GUI page, the DFG implementation in PM4Py is called. In the additional Docker container, the JSON response is parsed and populated into a DTO in the backend and then sent to the frontend GUI page. The SVG is shown directly in the frontend.

Generated Process-Aware Digital Twin Cockpit. Figure 11 shows an example GUI for the transportation data and process from a department perspective in the DT cockpit. The card top left visualizes the different locations of departments and allows users to use the picture as navigation to details. The other cards show information related to the selection, the Medicine department. The bar chart bottom left shows the transport capacity using real-time and historical information. The card top right presents some critical indicators regarding wheelchairs and transport beds maintenance. The dashboard below allows the selection of a time period of interest, and by clicking the *Show the Process* button, the discovered process can be shown. The process expert might learn the patient path through the hospital. The process and critical key indicators are shown on a separate page. The card on the bottom right shows data from transportation devices in a selected time period and allows access to detailed historical information via a button. Based on the considered models and the final business goal of the digital twin, other information can be visualized in Figure 11.

D. Phase 4: Runtime

The created PADTC allows for live monitoring as sensor information or lives data from the physical objects or third-party applications can be handled in digital twin services and visualized in the cockpit. Moreover, the users can control the digital twin via the process-aware digital twin cockpit by using execution services that translate user commands to machine or control commands and send them to the physical object, more precisely the cyber-part controlling the physical object. The latter is not implemented in our prototype; for example, we refer the reader to [34].

Initialization. We automatically add the data from the event logs used for model extraction into our PADTC database. This allows us to make this information already available when the process-aware digital twin cockpit is started for the first time.

We have also added the discovered roles and a dummy user for each role to be able to test the application. As part of the existing runtime environment, the digital twin cockpit already provides a settings menu where all roles and users are listed. This means that our domain or process expert is can see this data via the settings menu.

Runtime Models. We can use the models at runtime to step through the discovered processes. The discovered process might help the process expert to understand how the patient moves through the different departments in the hospital. In addition, from the discovered process, it is also possible to understand which departments are visited exclusively to each other. For example, considering the discovered process model depicted in Figure 8 the patients which go to the *admit Labor & Delivery* department never go to the *admit Emergency Department*.

Process Mining during Runtime. Our prototype allows running process mining algorithms during runtime of the PADTC. This enabled the process expert to analyze event logs or parts of them, e.g., to see if specific departments are always accessed from the same former departments, which can be used to organize the distribution of the transport beds.

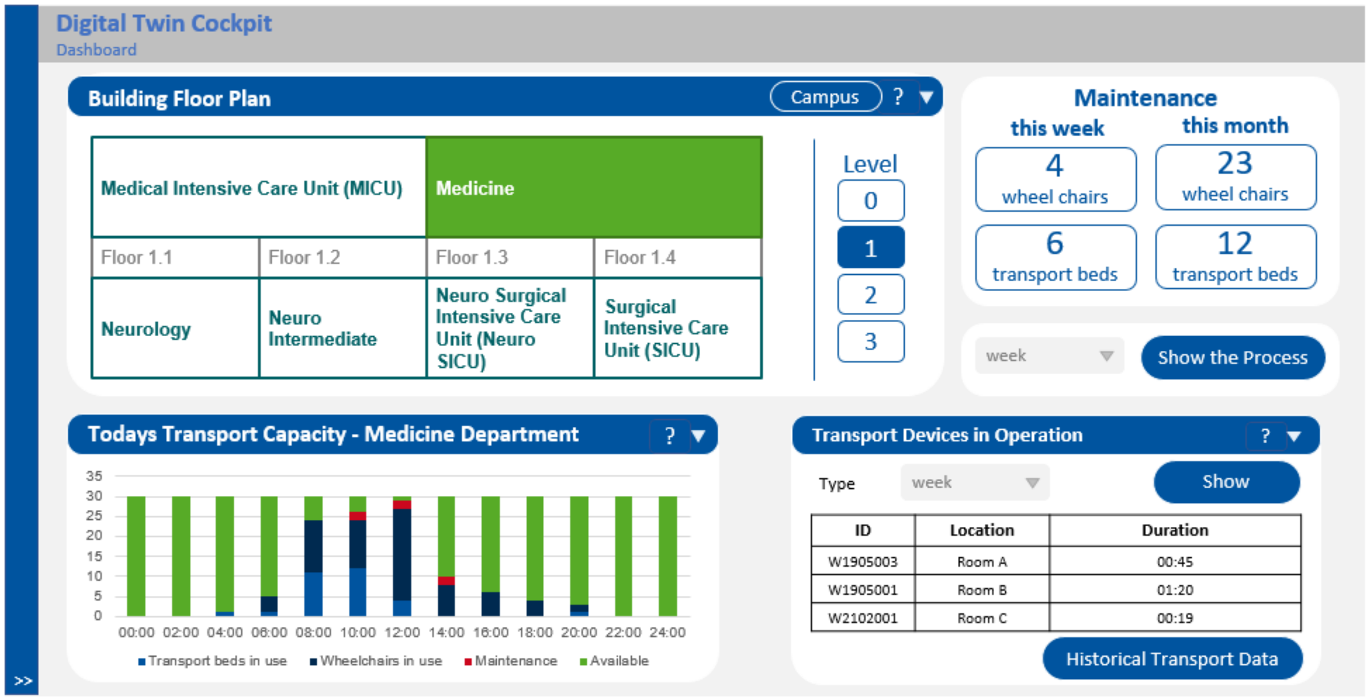
Live Data. We have implemented a WebSockets connection, which allows us to visualize the live data in the graphical interfaces in a dynamic way, e.g., via regularly updated charts. Furthermore, it is possible to use this live sensor data to create new process models step by step, which can be again visualized. Another alternative is to use an already discovered process model and highlight the occurring process steps.

Moreover, it is possible to control the physical objects via the PADTC: maintenance needs could call services for rerouting and automatically guide the physical objects to the technical department, errors on the "cyber" side of the physical object might require remote maintenance, which can be organized via the PADTC, and bottlenecks in the transport capacity of departments can be handled by automatic rerouting of beds.

This application scenario has provided us insights into the power of this approach and aspects where a higher degree of automation is desired and would help to lower the number of handwritten models and code.

VI. Related Work

We relate our approach to digital twin engineering approaches in general and discuss different aspects comparable to



ID	Location	Duration
W1905003	Room A	00:45
W1905001	Room B	01:20
W2102001	Room C	00:19

Fig. 11: Process-Aware Digital Twin Cockpit

our approach, as there exists only a low number of publications about low-code development approaches for process-aware digital twin cockpits.

(Low-Code) Digital Twin Engineering: Commercial Solutions. In recent years, various commercial platforms for the engineering of digital twins were developed, e.g., Eclipse Ditto[8] for developing DTs that provides standardized APIs for IoT devices could be used in combination with the Vorto modeling language[9] for describing device interfaces connected to (cyber-physical) systems. Similar solutions provide Amazon Greengrass[10] or Microsoft's Digital Twin Definition Language[11]. Other products including 2D and 3D visualizations are AWS IoT TwinMaker [12], the low-code data science platform Gramex [13] or the ASCon Digital Twin Product Suite [14]. The ASCon Digital Twin Product Suite might come closest to a PADTC, as it includes process models and a workflow engine in contrast to the others. Gramex and the ASCon Digital Twin Product Suite state that they provide a low-code experience. To the best of our knowledge, all of these solutions do not provide an automated mechanism to extract models from given sensor data or event logs, which means the digital twin engineer still has a high effort for defining the models or making configurations manually within the platforms. In some of these approaches, software engineering knowledge is needed to define the required information. Moreover, most of them allow manually connecting data with predefined visualization components, but they do not generate GUI models from existing data.

[8]https://www.eclipse.org/ditto/
[9]https://www.eclipse.org/vorto/
[10]https://aws.amazon.com/de/greengrass/
[11]http://www.aka.ms/dtdl
[12]https://aws.amazon.com/de/iot-twinmaker
[13]https://gramener.com/gramex
[14]https://ascon-systems.de/en/product-suite/application-engine/

(Low-Code) Digital Twin Engineering: Exemplary Academic Contributions. The data sources which are interesting to consider for digital twins are diverse and spread over heterogeneous applications. Faber et al. [26] discuss digital twins for the monitoring and analysis of wind farms using cloud technologies. They show different types of data sources for wind farm monitoring, which come from heterogeneous data sources and applications, e.g., Enterprise Resource Planning- (ERP), Customer Relationship Management (CRM), or Supply Chain Management(SCM) systems. In contrast to our approach, they do not provide an automated mechanism to extract models from given data or use models to define graphical user interfaces.

Govindasamy et al. [64] have created a digital twin for air quality management. Their digital twin prototype measures CO2, temperature, and humidity values of rooms within a building. The purpose of collecting this data in a digital twin is to improve work productivity and reduce the risk for virus infections. Furthermore, they suggest three services to be added to their implementation: a visualization, a physical simulation, and a prediction service. For processing the data received from the physical twin of the room the conceptual schema is presented. Similar to our approach, these models provide the required digital twin functionality. Unlike our approach, the models used as input for the framework are manually created and not automatically generated for the input data.

Bibow et al. [4] present a digital twin for an injection molding process. The presented approach aims to engineer digital twins while supporting domain-specific customization and at the same time automating the essential activities based on the model-driven reference architecture. The realization requires defining a domain model, a tag model to link concepts in the domain model with database information, and defining event models that occur in the physical system. Their generated, executable digital twin is still extendable using handwritten additions. Unlike our approach, the input data are queried from the so-called Data Lake, an extensive data storage containing multiple databases or other data providers. In addition, the Data Lake is responsible for data preparation and processing in case the repositories contain unstructured data. Similar to our approach, they use the domain model to describe the data structure the digital twin-component needs to exchange with each other. In contrast, the domain model is manually created with the help of the domain expert and is not automatically inferred from the data stored in the Data Lake. Moreover, the following steps and the prototype in [4] are different from our approach and prototype as they use, e.g., architectural models as input of a code generator.

Our presented low-code development approach does not offer built-in deployment and lifecycle management facilities for the created process-aware digital twin cockpit. The proposed approach could be embedded into a low-code development platform to enable that. In [17], we have presented a vision towards LCDPs for digital twins where domain experts can select language plugins and, according to architectural components, generate a digital twin in the next step. The low-code development approach proposed in this article could be integrated into such a LCDP for digital twins, as this would allow domain experts without software engineering knowledge to create the first version of their process-aware digital twin cockpit. However, this goes beyond the current implementation.

To sum up, in comparison to our proposed approach, the approaches for digital twin engineering do not provide an automated mechanism to extract models from given sensor data or event logs, which means the digital twin engineer has a higher effort for defining the models manually. Thus, our approach allows for a higher degree of automation in digital twin construction. Besides that, none of these approaches aim to generate the process-aware digital twin cockpit from the sensor data. Moreover, most of them do not consider BPMN models during runtime within a workflow engine or generating GUI models from process models.

Event log extraction from sensor data. Our approach starts in step 1 (see Figure 2) with the event log extraction from given sensor data. There exist several approaches in literature discussing such extractions.

Eck, Sidorova, and van der Aalst [44] have applied process mining on sensor data in order to discover the process model based on human behavior. First, an event log is extracted from the sensor data subsequently analyzed using different process mining techniques. A smart product designers use the insights gained from the discovered process. The first challenge that the authors emphasize is the mapping between the sensor measurements to human activities, and the second one is grouping these activities into the process instances. Unlike our approach, the sensor data are not known from which sensor they are coming from. Therefore, the first step of their approach is to divide the sensor data into windows. After making sure that each window contains measurements from a single sensor. Afterward, the domain knowledge comes into play to label them. The choice to determine the main characteristics of the process instances is also made by domain knowledge.

Janssen et al. [43] are applying unsupervised learning techniques in the form of clustering to transform the sensor data into event data. The main goal of this step is to identify the sub-traces and afterward group them. The authors use a combined time- and quantity-based similarity measure to find similar traces. After the clusters of similar traces are discovered in the next step, the activity labels are defined, which requires the involvement of humans. In addition, they are addressing the challenges of concurrent behavior between activities. Their sensors attached to smart homes or factories cannot identify entities. This is different from our setup scenario in which we are assuming that the sensors are logging the department name whenever they become active.

Senderovich et al. [48] provides a transformation of sensor data (e.g., Real-Time Locating System data) into the event log data based on the interaction notion, which is defined as an intermediate knowledge layer. Under the assumption that interactions correspond to an activity instance, the authors propose an optimal matching solution to map interactions to event log activity labels. A set of interaction-to-activity mapping complements the event log extraction. Based on the domain of our running example, the interactions correspond to business entities such as patients or doctors. If two entities share the exact location indicates an interaction that can be part of a particular activity instance. Similar to our sensor data, interactions between entities are known and can be captured only within a department.

Process Mining and Digital Twins. In resent years, the use of process mining for and within digital twins increased. van der Aalst [65] emphasizes that the creation of the digital twin in one organization requires the need to define the fabric of the real operational process. In addition, he argues that process mining is an excellent option to move towards the digital twin. However, this publication does not provide an approach regarding the construction of a digital twin. In more recent work, Park and van der Aalst [66] present how to create a digital twin of an organization. They have implemented a web service for building, updating, and visualizing digital twin interface models, which are object-centric Petri nets. Their implementation could be a part of a process-aware digital twin cockpit which allows only hand-written code additions where our approach allows to add hand-written models. Moreover, our approach provides additional visualizations for non-process related data which is generated from GUI models.

Brockhoff et al. [18] show a vision towards the use of process mining techniques during the run-time of a self-adaptive digital twin in combination with digital shadows. They also include process-aware digital twin cockpits as one part which is generated but focus on process mining in interaction with self-adaptivity services in a MAPE-K loop of digital twins. They present design time models without considering their automatic transformation from other information sources and they use event logs within digital twin services but not as source for deriving models for the generation process.

Lugaresi and Matta [67] use a process mining approach to discover manufacturing systems automatically. They automatically retrieve relevant characteristics of a production system from data logs and create digital twins that can estimate system performance. Again, we would not call the derived result a digital twin but digital twin services which could be used in a digital twin. Moreover, they do not generate GUIs from event log data and do not provide a process engine component.

Similar approaches exist, mainly using artificial intelligence approaches for model discovery from event logs instead of process mining. Yang et al. [68] provide an approach where they use event logs to create transition systems. These transition systems train a neural network that creates a digital twin that predicts the remaining cycle time in the manufacturing process. In their understanding, a digital twin is a set of pictures in smart manufacturing processes to conduct experiments. From our understanding, we would not call the derived result a digital twin but digital twin services which could be used in a digital twin. Their approach generates no source code and provides no process engine component; thus, it can not be seen as a process-aware system.

To sum up, various approaches cover different aspects of our approach, especially the commercial digital twin engineering platform providers, but not in the given combination.

VII. Discussion

This section discusses the strong points and the limitation of our method and the presented use case application.

Challenges in event log extraction. Usually, the requirements for an event log that is subject to process mining are easily defined when they are extracted from *process-aware* information systems that operate based on process models [69]. However, it might be necessary to extract such event logs from process-unaware information systems, which implies that the event log might lack the information required for the process mining. Therefore, the association of the sensor data with the process instances is a challenging task [70].

Challenges in the event log extraction strongly depend on the sensor data format and the final business goal. Our final goal is mainly focused on the patient flow within the hospital. If it is necessary to extract the event log, keeping in mind a different business goal, the process discovery must be adjusted to meet these goals. Another challenge of working with sensor data is to correlate them into event log cases, representing process instances since they are not outputs from process-aware information systems.

Correctness of the Transformations. Within the prototype, the data-to-model, model-to-model, and model-to-code transformations were tested using a test infrastructure, a set of models covering all concepts of the DSL grammars, and a comparison with the expected outcome. This approach works fine for the model-to-model and model-to-code transformations, as we know what outcome to expect. It might be challenging to know what is expected in the data-to-model transformation as event logs and the resulting BPMN models could be quite large. However, it is also feasible to choose individual cases from the event log and compare them with their occurrence in the resulting BPMN model.

Using process models for generation. Our process in Fig. 3 shows that we use BPMN models for the system generation, which requires the regeneration of the system in case new processes are discovered. However, as this step could be automated together with continuous integration and deployment strategy, this is not a strong limitation. Another possible realization is to use BPMN models only at runtime. However, this would require a higher amount of handwritten code for GUIs.

Extension of the extracted BPMN models. Dependent on the aim of the process-aware digital twin cockpit and, consequently, what should be shown in the user interfaces, it might be interesting to make additions to the derived BPMN models ((3) in Figure 2). Our current implementation would, e.g., allow specifying input and output data of process steps. Thus, the discovered GUI models ((6) in Figure 2) could include input forms with different fields to add data as a user within a process step. This is only possible if the needed information exists within the BPMN model, requiring a handwritten extension. In the current implementation, this is not realized.

Increase the degree of automation. The proposed approach relies on its higher degree of automation in comparison to manual engineering processes of PADTCs and the aim to reduce the handwritten code. However, this could be even further improved:

- The **structure from event logs** which is used as an additional handwritten domain model, could be automatically discovered from the given event log as well. This would result in an additional data-to-model transformation.
- The current approach allows to add a **domain model** which includes additional concepts from the domain, e.g., the maintenance information for CPS in our example. This information is typically included in other applications, e.g., SAP Plant Maintenance (PM). **Analyzing third-party applications** could help to detect which

domain information might be relevant for the digital twin. Additionally, the **analysis of engineering models** for the creation of the physical object under investigation might also reveal important information usable within the domain model.

- The **structure of digital shadows and process structures** could be already provided in some model libraries for application-independent models, such as suggested in [18] and automatically added in the first generation step. To derive the concepts for the used DSLs during runtime can be easily realized for MontiCore [45] DSLs, as the grammar is already mapped to a class diagram which could be directly used.
- **GUIs for processes and event logs** could be already predefined and provided as application-independent models. This would allow the digital twin engineer to reduce effort and directly use them within the generation process.
- **GUI models** related to the domain can be generated in a similar way to the approach in [46], where visualizations for each data class and their relations are generated. To stay flexible, it is possible to change or extend generated GUI models with additional handwritten ones following one of the approaches in [71]. If we are able to generate large parts of the GUI models would significantly reduce the number of needed handwritten ones.

Extendable architecture. As mentioned by Bock and Frank [72], it is interesting to provide an adaptable software architecture with predefined components or services. Our approach is highly extendable as it is possible to connect DT services with the process-aware digital twin cockpit. Furthermore, we allow making handwritten extensions of the code, which means this handwritten code can be larger, existing application parts which are reused.

DT services for process prediction and forecasts. Current approaches in research move towards process model forecasting [73] and prediction. To include such approaches could also be an interesting aspect for PADTC, as they provide the domain experts additional insights into the future of their physical objects.

Plugin system for digital twin services. The current implementation requires adding the digital twin services via APIs, which requires handwritten code to handle the sent and received data. It might also result in the need to start a regeneration process if specific data received from these services has to be stored in the database, but no according to structure exists. This would also require adding GUI and data models. A better approach to allow the connection with DT services is to realize a plugin system for DT services together with a generic structure to be able to store data. This would not result in the need for regeneration.

Connection to Low-Code Development Platforms for Digital Twins. In principle, it would be possible to take the first steps of our approach and realize it as an extension of a low-code development platform for digital twins. However, to integrate such an extension in existing platforms would require (1) contacts to the platform developers as they are closed systems and provide, if any, only means for restricted extensions or (2) to develop your low-code development platform [17].

Generalizability of the approach. The general approach is domain-independent, and it can be used for different domains where a PADTC is needed for a physical object. The information in the described models is needed to generate as much as possible. However, the described languages are replaceable, e.g., to use other users interface languages or process modeling languages. Replacing languages would result in other data-to-model and model-to-model transformations. If another process language than BPMN is used, e.g., Petri nets, a process engine tailored for this language [74] is needed. The largest effort in realizing this approach is needed to implement the code generator.

VIII. Conclusion

This paper presents a low-code development approach to generate a process-aware digital twin cockpit from event logs and show a prototype for an automated hospital transportation system. We suggest using process mining techniques to extract an event log from sensor data and then apply data-to-model transformations to infer the data model, and to discover the process model and roles. We suggest using model-to-model and model-to-code transformation techniques to automate the digital twin generation process. The discovered models are used as input for a code generator. We applied our approach to real-life sensor data in an automated hospital transportation system use case and show that it is possible to achieve a high degree of automation in the process-aware digital twin cockpit engineering process.

In future work, it would be interesting to integrate the proposed low-code development approach in a low-code platform such as described in [17], where domain users are empowered to guide remaining semi-automatic steps via a GUI and make additions needed to define a more sophisticated process-aware digital twin cockpit. This requires further analysis of the hand-written parts of the application to make suggestions for abstractions.

Acknowledgements

This work was 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

References

[1] 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.

[2] F. Jiang, L. Ma, T. Broyd, and K. Chen, "Digital twin and its implementations in the civil engineering sector," *Automation in Construction*, vol. 130, p. 103838, 2021.

[3] V. Zaccaria, M. Stenfelt, I. Aslanidou, and K. G. Kyprianidis, "Fleet monitoring and diagnostics framework based on digital twin of aero-engines," in *Turbo Expo: Power for Land, Sea, and Air*, vol. 51128. American Society of Mechanical Engineers, 2018, p. V006T05A021.

[4] 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 *Advanced Information Systems Engineering*, ser. LNCS, S. Dustdar, E. Yu, C. Salinesi, D. Rieu, and V. Pant, Eds., vol. 12127. Springer, 2020, pp. 85–100.

[5] J. Lipp, S. Sakik, M. Kröger, and S. Decker, "LISSU: Integrating Semantic Web Concepts into SOA Frameworks," in *23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, INSTICC. SciTePress, 2021, pp. 855–865.

[6] Y. Liu, L. Zhang, Y. Yang, L. Zhou, L. Ren, F. Wang, R. Liu, Z. Pang, and M. J. Deen, "A novel cloud-based framework for the elderly healthcare services using digital twin," *IEEE Access*, vol. 7, pp. 49 088–49 101, 2019.

[7] 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, 2019.

[8] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, "Low-code development and model-driven engineering: Two sides of the same coin?" *Software and Systems Modeling*, 2022.

[9] F. Becker, P. Bibow, M. Dalibor, A. Gannouni, V. Hahn, C. Hopmann, M. Jarke, I. Koren, M. Kröger, J. Lipp, J. Maibaum, J. Michael, B. Rumpe, P. Sapel, N. Schäfer, G. J. Schmitz, G. Schuh, and A. Wortmann, "A Conceptual Model for Digital Shadows in Industry and its Application," in *Conceptual Modeling, ER 2021*, A. Ghose, J. Horkoff, V. E. Silva Souza, J. Parsons, and J. Evermann, Eds. Springer, 2021, pp. 271–281.

[10] K. Bruynseels, F. Santoni de Sio, and J. van den Hoven, "Digital twins in health care: Ethical implications of an emerging engineering paradigm," *Frontiers in Genetics*, vol. 9, p. 31, 2018.

[11] F. Biesinger, D. Meike, B. Kraß, and M. Weyrich, "A digital twin for production planning based on cyber-physical systems: A case study for a cyber-physical system-based creation of a digital twin," *Procedia CIRP*, vol. 79, pp. 355–360, 2019.

[12] S. A. P. Kumar, R. Madhumathi, P. R. Chelliah, L. Tao, and S. Wang, "A novel digital twin-centric approach for driver intention prediction and traffic congestion avoidance," *Journal of Reliable Intelligent Environments*, vol. 4, no. 4, pp. 199–209, 2018.

[13] F. Lima, C. N. de Carvalho, M. B. S. Acardi, E. G. dos Santos, G. B. de Miranda, R. F. Maia, and A. A. Massote, "Digital manufacturing tools in the simulation of collaborative robots: Towards industry 4.0," *Brazilian Journal of Operations & Production Management*, vol. 16, no. 2, pp. 261–280, 2019.

[14] W. van der Aalst, *Data Science in Action*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–23. [Online]. Available: https://doi.org/10.1007/978-3-662-49851-4_1

[15] D. Bano and M. Weske, "Discovering data models from event logs," in *Conceptual Modeling - 39th International Conference, ER 2020, Vienna, Austria, November 3-6, 2020, Proceedings*, ser. Lecture Notes in Computer Science, G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, and H. C. Mayr, Eds., vol. 12400. Springer, 2020, pp. 62–76. [Online]. Available: https://doi.org/10.1007/978-3-030-62522-1_5

[16] 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., May 2020, pp. 59–66.

[17] 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.

[18] 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 *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. ACM/IEEE, October 2021, pp. 182–187.

[19] C. Richardson and J. Rymer, "New development platforms emerge for customer-facing applications," 2014. [Online]. Available: www.forrester.com

[20] D. Jones, C. Snider, A. Nassehi, J. Yon, and B. Hicks, "Characterising the digital twin: A systematic literature review," *CIRP Journal of Manufacturing Science and Technology*, vol. 29, pp. 36–52, 2020.

[21] A. Ardanza, A. Moreno, Á. Segura, M. de la Cruz, and D. Aguinaga, "Sustainable and flexible industrial human machine interfaces to support adaptable applications in the industry 4.0 paradigm," *International Journal of Production Research*, vol. 57, no. 12, pp. 4045–4059, 2019.

[22] A. Martins, H. Costelha, and C. Neves, "Shop floor virtualization and industry 4.0," in *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE, 2019, pp. 1–6.

[23] A. S. Ullah, "Modeling and simulation of complex manufacturing phenomena using sensor signals from the perspective of industry 4.0," *Advanced Engineering Informatics*, vol. 39, pp. 1–13, 2019.

[24] R. Dong, C. She, W. Hardjawana, Y. Li, and B. Vucetic, "Deep learning for hybrid 5g services in mobile edge computing systems: Learn from a digital twin," *IEEE Transactions on Wireless Communications*, vol. 18, no. 10, pp. 4692–4707, 2019.

[25] J. Ríos, J. C. Hernández, M. Oliva, and F. Mas, "Product Avatar as Digital Counterpart of a Physical Individual Product: Literature Review and Implications in an Aircraft," in *Volume 2: Transdisciplinary Lifecycle Analysis of Systems*, ser. Advances in Transdisciplinary Engineering, 2015, pp. 657 – 666.

[26] H. Pargmann, D. Euhausen, and R. Faber, "Intelligent big data processing for wind farm monitoring and analysis based on cloud-technologies and digital twins: A quantitative approach," in *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. IEEE, 2018, pp. 233–237.

[27] 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, 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018.

[28] K. T. Park, Y. W. Nam, H. S. Lee, S. J. Im, S. D. Noh, J. Y. Son, and H. Kim, "Design and implementation of a digital twin application for a connected micro smart factory," *International Journal of Computer Integrated Manufacturing*, vol. 32, no. 6, pp. 596–614, 2019.

[29] C. Mandolla, A. M. Petruzzelli, G. Percoco, and A. Urbinati, "Building a digital twin for additive manufacturing through the exploitation of blockchain: A case analysis of the aircraft industry," *Computers in Industry*, vol. 109, pp. 134–152, 2019.

[30] 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, 2020, pp. 377–387.

[31] P. Brauner, M. Dalibor, M. Jarke, I. Kunze, I. Koren, G. Lakemeyer, M. Liebenberg, J. Michael, J. Pennekamp, C. Quix, B. Rumpe, W. van der Aalst, K. Wehrle, A. Wortmann, and M. Ziefle, "A Computer Science Perspective on Digital Transformation in Production," *ACM Trans. Internet Things*, vol. 3, pp. 1–32, 2022.

[32] A. Mertens, S. Pütz, P. Brauner, F. Brillowski, N. Buczak, H. Dammers, M. Van Dyck, I. Kong, P. Königs, F. Kordtomeikel, N. Rodemann, A. K. Schaar, L. Steuer-Dankert, S. Wlecke, T. Gries, C. Leicht-Scholten, S. K. Nagel, F. T. Piller, G. Schuh, M. Ziefle, and V. Nitsch, "Human digital shadow: Data-based modeling of users and usage in the internet of production," in *2021 14th International Conference on Human System Interaction (HSI)*, 2021, pp. 1–8.

[33] P. Brauner, R. Philipsen, A. C. Valdez, M. Ziefle, and R. Philipsen, "What happens when Decision Support Systems fail? – The Importance of Usability on Performance in Erroneous Systems," *Behaviour & Information Technology*, vol. 38, no. 12, pp. 1225–1242, 2019.

[34] 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.

[35] W. M. P. van der Aalst, A. Adriansyah, A. K. A. De Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. Van Den Brand, R. Brandtjen, J. Buijs *et al.*, "Process mining manifesto," in *International Conference on Business Process Management*. Springer, 2011, pp. 169–194.

[36] K. Diba, K. Batoulis, M. Weidlich, and M. Weske, "Extraction, correlation, and abstraction of event data for process mining," *Wiley*

Interdiscip. Rev. Data Min. Knowl. Discov., vol. 10, no. 3, 2020. [Online]. Available: https://doi.org/10.1002/widm.1346

[37] W. M. P. van der Aalst, "Process mining - discovery, conformance and enhancement of business processes." Springer, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-19345-3

[38] W. van der Aalst, "Process mining: Overview and opportunities," *ACM Trans. Manage. Inf. Syst.*, vol. 3, no. 2, Jul. 2012. [Online]. Available: https://doi.org/10.1145/2229156.2229157

[39] B. Mikolajczak and J.-L. Chen, "Workflow mining alpha algorithm — a complexity study," in *Intelligent Information Processing and Web Mining*, M. A. Kłopotek, S. T. Wierzchoń, and K. Trojanowski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 451–455.

[40] A. Weijters, W. Aalst, and A. Medeiros, "Process Mining with the Heuristics Miner-algorithm," BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven, 2006.

[41] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Process and deviation exploration with inductive visual miner," in *Proc. of the BPM Demo Sessions 2014 Co-located with the 12th Int. Conf. on Business Process Management (BPM 2014)*, ser. CEUR Workshop Proceedings, vol. 1295. CEUR-WS.org, 2014, p. 46.

[42] B. Dongen, A. Medeiros, H. Verbeek, A. Weijters, and W. M. P. van der Aalst, "The ProM framework: A new era in process mining tool support," vol. 3536, 06 2005, pp. 444–454.

[43] D. Janssen, F. Mannhardt, A. Koschmider, and S. J. van Zelst, "Process model discovery from sensor event data," in *Process Mining Workshops ICPM 2020*, ser. LNBIP, S. J. J. Leemans and H. Leopold, Eds., vol. 406. Springer, 2020, pp. 69–81.

[44] M. L. V. Eck, N. Sidorova, and W. V. Aalst, "Enabling process mining on sensor data from smart products," *Int. Conf. on Research Challenges in Information Science (RCIS)*, pp. 1–12, 2016.

[45] K. Hölldobler, O. Kautz, and B. Rumpe, *MontiCore Language Workbench and Library Handbook: Edition 2021*, ser. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021. [Online]. Available: http://www.monticore.de/handbook.pdf

[46] 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), August 2020, pp. 1–10.

[47] M. Dalibor, N. Jansen, J. C. Kirchhof, B. Rumpe, D. Schmalzing, and A. Wortmann, "Tagging Model Properties for Flexible Communication," in *Proc. of MODELS 2019. Workshop MDE4IoT*, N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer, and A. Wortmann, Eds. CEUR Workshop Proceedings, 2019, pp. 39–46.

[48] A. Senderovich, A. Rogge-Solti, A. Gal, J. Mendling, and A. Mandelbaum, "The ROAD from sensor data to process instances via interaction mining," in *Advanced Information Systems Engineering - 28th Int. Conf. (CAiSE'16)*, ser. LNCS, S. Nurcan, P. Soffer, M. Bajec, and J. Eder, Eds., vol. 9694. Springer, 2016, pp. 257–273.

[49] Object Management Group, "Business Process Model and Notation (BPMN), v2.0.2," http://www.omg.org/spec/BPMN/2.0.2/.

[50] M. Weske, *Business Process Management: Concepts, Languages, Architectures*, 3rd ed. Berlin Heidelberg: Springer-Verlag, 2019.

[51] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *Int. J. Cooperative Inf. Syst.*, vol. 23, no. 1, 2014. [Online]. Available: https://doi.org/10.1142/S0218843014400012

[52] 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.

[53] A. Burattin, A. Sperduti, and M. Veluscek, "Business models enhancement through discovery of roles," 04 2013, pp. 103–110.

[54] A. Berti, S. J. van Zelst, and W. M. P. van der Aalst, "Process mining for python (pm4py): Bridging the gap between process- and data science," *CoRR*, vol. abs/1905.06169, 2019. [Online]. Available: http://arxiv.org/abs/1905.06169

[55] E. Meephu, S. Arwatchananukul, and N. Aunsri, "A Framework for Development of an Intra-Hospital Patient Transfer Using Queue Management System," in *Global Wireless Summit (GWS)*, 2018, pp. 300–303.

[56] T. Hanne, T. Melo, and S. Nickel, "Bringing robustness to patient flow management through optimized patient transports in hospitals," *Interfaces*, vol. 39, no. 3, pp. 241–255, 2009.

[57] A. E. Johnson, T. J. Pollard, L. Shen, L. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, and R. G. Mark, "Mimic-iii, a freely accessible critical care database," *Scientific data*, vol. 3, p. 160035, 2016.

[58] E. G. L. de Murillas, H. A. Reijers, and W. M. P. van der Aalst, "Case notion discovery and recommendation: automated event log building on databases," *Knowl. Inf. Syst.*, vol. 62, no. 7, pp. 2539–2575, 2020. [Online]. Available: https://doi.org/10.1007/s10115-019-01430-6

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

[60] A. Berti, S. J. van Zelst, and W. M. van der Aalst, "Process mining for python (pm4py): Bridging thegap between process- and data science," in *ICPM Demo Track 2019*, A. Burattin, A. Polyvyanyy, and S. J. van Zelst, Eds. CEUR-WS.org, 2019, pp. 13–16.

[61] OMG, *Business Process Model and Notation (BPMN), Version 2.0*, Object Management Group Std., Rev. 2.0, January 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[62] D. Pérez-Alfonso, O. Fundora-Ramírez, M. S. Lazo-Cortés, and R. Roche-Escobar, "Recommendation of process discovery algorithms through event log classification," in *Pattern Recognition*, J. A. Carrasco-Ochoa, J. F. Martínez-Trinidad, J. H. Sossa-Azuela, J. A. Olvera López, and F. Famili, Eds. Cham: Springer International Publishing, 2015, pp. 3–12.

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

[64] H. S. Govindasamy, R. Jayaraman, B. Taspinar, D. Lehner, and M. Wimmer, "Air quality management: An exemplar for model-driven digital twin engineering," in *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 229–232.

[65] W. M. P. van der Aalst, "Concurrency and objects matter! disentangling the fabric of real operational processes to create digital twins," in *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, ser. Lecture Notes in Computer Science, A. Cerone and P. C. Ölveczky, Eds., vol. 12819. Springer, 2021, pp. 3–17. [Online]. Available: https://doi.org/10.1007/978-3-030-85315-0_1

[66] G. Park and W. M. Van Der Aalst, "Realizing A Digital Twin of An Organization Using Action-oriented Process Mining," in *3rd International Conference on Process Mining (ICPM)*, 2021, pp. 104–111.

[67] G. Lugaresi and A. Matta, "Automated manufacturing system discovery and digital twin generation," *Journal of Manufacturing Systems*, vol. 59, pp. 51–66, 2021.

[68] M. Yang, J. Moon, J. Jeong, S. Sin, and J. Kim, "A novel embedding model based on a transition system for building industry-collaborative digital twin," *Applied Sciences*, vol. 12, no. 2, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/2/553

[69] W. M. P. van der Aalst, "Process-aware information systems: Lessons to be learned from process mining," in *Transactions on petri nets and other models of concurrency II*. Springer, 2009, pp. 1–26.

[70] S. Goel, J. M. Bhat, and B. Weber, "End-to-end process extraction in process unaware systems," in *International Conference on Business Process Management*. Springer, 2012, pp. 162–173.

[71] A. Gerasimov, J. Michael, L. Netz, and B. Rumpe, "Agile Generator-Based GUI Modeling for Information Systems," in *Modelling to Program (M2P)*, A. Dahanayake, O. Pastor, and B. Thalheim, Eds. Springer, March 2021, pp. 113–126.

[72] A. C. Bock and U. Frank, "In search of the essence of low-code: An exploratory study of seven development platforms," in *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 57–66.

[73] J. De Smedt, A. Yeshchenko, A. Polyvyanyy, J. De Weerdt, and J. Mendling, "Process model forecasting using time series analysis of event sequence data," in *Conceptual Modeling*, A. Ghose, J. Horkoff, V. E. Silva Souza, J. Parsons, and J. Evermann, Eds. Springer International Publishing, 2021, pp. 47–61.

[74] S. Pellegrini and F. Giacomini, "Design of a Petri Net-Based Workflow Engine," in *3rd Int. Conf. on Grid and Pervasive Computing - Workshops*, 2008, pp. 81–86.