Springer Nature Computer Science manuscript No.
(will be inserted by the editor)

Model-Driven Engineering of Process-Aware Information Systems

Imke Drave · Judith Michael* · Erik Müller · Bernhard Rumpe · Simon Varga

Received: date / Accepted: date

Abstract Enterprise information systems created with model-driven software engineering methods need to handle not only data but also business processes in an automated way. This paper shows how to engineer process-aware information systems following the model-driven and generative software engineering paradigms. Existing approaches realize either the generation of automated or manual activities but do not employ model-driven engineering of all system aspects through systematic language composition. A generative approach that additionally uses process modeling languages allows developers to evolve generated data-centric information systems into process-aware information systems. To be usable within our generation process, we have developed a textual BPMN version and according language tooling to check the soundness of the models. We have included these process models into the generation process of an information system together with other domain-specific modeling languages, e.g., for data structures, and generate an extendable, process-aware information system that is open for continuous regeneration and hand-written additions. This approach allows us to lift a generated data-centric information system to a process-aware information system. Agile development enabled through the opportunity to validate assumptions automatically and adapt changes efficiently, enhances the engineering process as well as the generated systems themselves.

Keywords Model-Driven Software Engineering · Business Processes · Domain-Specific Modeling Languages · Code Generation · Process-Aware Information System.

Imke Drave
Software Engineering, RWTH Aachen University, Germany
E-mail: drave@se-rwth.de

✉ Judith Michael
Software Engineering, RWTH Aachen University, Germany
E-mail: michael@se-rwth.de

Erik Müller
Software Engineering, RWTH Aachen University, Germany
E-mail: erik.mueller@rwth-aachen.de

Bernhard Rumpe
Software Engineering, RWTH Aachen University, Germany
E-mail: rumpe@se-rwth.de

Simon Varga
Software Engineering, RWTH Aachen University, Germany
E-mail: varga@se-rwth.de

1 Introduction

Motivation and relevance. Model-Driven Software Engineering (MDSE) uses models as primary artifacts to derive code, tests and documentation and has established as a paradigm in software engineering throughout the past decades. Therein, the use of models instead of, for example code, narrows the conceptual gap between the problem domain and the solution domain [26].

Enterprise Information System (EIS) are software systems that collect, store, and assimilate data and information, and also provide feedback [68]. By nature, these systems are highly complex and evolve continuously making them a predestined application domain for MDSE. Part of the complexity of engineering such systems arises from the fact that there are multiple and heterogeneous aspects [18] when it comes to engineering EISs. Examples for such aspects are the Graphical User Interface (GUI) and the data structures. Here, MDSE enables engineers to use tailored modeling languages for specifying these aspects, while model-to-code transformations enable generating an application that inte-

[DMM+22] I. Drave, J. Michael, E. Müller, B. Rumpe, S. Varga:
Model-Driven Engineering of Process-Aware Information Systems.
In: Springer Nature Computer Science Journal, Volume 3, Springer, November 2022.
www.se-rwth.de/publications/

grates all modeled aspects automatically. Since 2016, we are developing a full-size real world application for the financial controlling of small and medium sized university chairs called Management Cockpit for Controlling (MaCoCo) [28]. It is used by more than 160 chairs of the university and has a code base of app. 9.000 Line of Code (LOC) in models, app. 390.000 LOC generated and 115.000 LOC hand-written code. Employing MDSE methods has proven to increase development efficiency and quality of the final application significantly.

However, through the demand for new or adapted functionalities, not only the system evolves continuously but also the set of modeling languages and code generators employed in the development process [18]. Technology stacks and development processes therefore need to be amenable for change. New requirements, especially for larger organizations, have triggered the transition from data-centric EIS to Process-Aware Information System (PAIS), which apart from data, provide support for structured processes. Processes have therefore become another aspect of the application under development and integrating model-driven process development into existing model-driven development processes of EISs has become subject to ongoing research. Doing so, requires suitable modeling languages for specifying processes as a chain of tasks or steps performed by humans or other systems with specific roles in the organization, that enable to reference elements of the existing data structures to specify the inputs and outputs of the process tasks or steps. Common languages to describe such processes are, e.g., Business Process Model and Notation (BPMN) or UML Activity Diagrams, typically employed using the graphical notation proposed by their respective standards [57,56]. To support the transition process from an EIS to a PAIS, process models and respective generators need to be integrated in the technological landscape and methodology of model-driven EIS development. However, generating information systems using behavior models and integrating them with hand-written code is still a challenge. This paper proposes a textual notation of BPMN and its integration into an existing technology stack employed in the ongoing EIS development of MaCoCo [28].

In this paper, we show *how to engineer process-aware information systems following the model-driven and generative software engineering paradigms.* Our approach combines both, the generative use of process models, e.g., written in BPMN, as well as the interpretation of these models at run-time.

Approach and main results. This paper presents our generative approach to engineering PAIS that uses, among others, process models as input to generate the code base that integrates a process engine in the system architecture. For the implementation of all languages mentioned in this paper, we used MontiCore [37], a language workbench for Domain-Specific Languages (DSLs), which generates language infrastructure from a context-free grammar and provides mechanisms for integrating hand-written code. Thus, we propose a textual notation for BPMN, suited for code generation with MontiCore. Our BPMN DSL covers all relevant concepts for code generation, which is 88.4% of the common executable BPMN elements and enables the definition of additional data structures. For the generation process of the PAIS, we have extended the generator framework MontiGem [28,2], which, so far, has used structural models to generate a data-centric application such as MaCoCo. The extension allows to generate a PAIS from UML Class Diagrams (CDs), Object Constraint Language (OCL) expressions, models for the GUIs, tagging models, and BPMN models. This approach makes it possible to create a PAIS automatically from a set of models and still allows for hand-written extensions and continuous re-generation of the resulting application.

Outline. The next Section 2 shows our vision of generating PAIS. Section 3 discusses relevant preliminaries: the language workbench MontiCore for creating different DSLs, the generator framework MontiGem to create the PAIS, the used DSLs such as the process language BPMN and our running example from the quality assurance process of a manufacturing plant. Section 4 presents our textual BPMN DSL and the process for model validation. Section 5 describes the process-related extensions to the run-time environment as well as the generation process using the generator framework MontiGem and our BPMN DSL. Section 6 shows example models, the generation results as well as an overview of the generated and hand-written artifacts. Section 7 discusses related work. Section 8 shows limitations, and strengths of our approach and the last section concludes this paper.

2 Towards Generated Process-Aware Information Systems

Nowadays, the focus of EIS development has shifted from data to process orientation [66] which gives rise to so called PAISs. According to [1,19], a PAIS is a software system that uses process models to manage and execute operational processes involving people, applications and/or information sources. When it comes to defining business processes, their implementation and integration within an existing EIS is a major challenge. MDSE provides means to overcome this challenge by

introducing models as the main development artifacts that serve as a communication basis and enable code generation at the same time.

A Vision Towards Generating PAIS. Process modeling languages, such as BPMN enable to describe business process models by abstracting from the implementation platform. Further, there exist various techniques to analyze, interpret and transform business process models based on mathematical theory in the literature, e.g., [77,75,81]. Tailoring PAISs for a specific application domain thereby becomes much easier, because it enables engineers to translate customer requirements into a model that abstracts from the implementation platform that is also amenable for automatic processing such as code generation. Therefore, we *envision* to utilize models in a formal process modeling language for generating the process-related functionality of a PAIS.

As in MaCoCo, engineering PAISs will most likely be brownfield development, where an existing (data-centric) EIS needs to be "lifted" to a PAIS that integrates the process related functionality. Following the trend of entrepreneurial software to integrate existing pieces of software to obtain an implementation of an application, rather than to implement an entire application from scratch, we *envision* to obtain a generative approach to engineering PAISs by integrating generative engineering of business processes with generative engineering of EISs.

Requirements for Process Modeling. In MDSE, models are the primary development artifacts. To enable this, the models must be comprehensive and intuitive for all model users. At the same time, the models that are used for communication and those that are used for code generation should be the same or obtained from automatic transformations. Standardized modeling languages such as BPMN have established in the domain of business process engineering [12]. Stakeholders whose background is not necessarily related to computer science are therefore familiar with the language and able to read and understand it. Further, various techniques to analyze, interpret and transform business process models based on mathematical theory exist in the literature, e.g., [77,75,81]. These techniques provide a conceptual basis for generative engineering of PAISs. Hence, we *require* to reuse the BPMN standard for defining the modeling language and existing techniques or tools to process BPMN models.

For modeling the input and outputs of tasks, process models need to reference the data classes modeled in a data model of the system and the generation process must assure that the type of a task's input is given by a corresponding class generated from the data models. These are two respective aspects in the sense of [18] the system and both may be relevant for other aspects. To keep models readable and to maintain their purpose, these aspects need to be modeled using appropriate languages. We *require* a composed process modeling language. The infrastructure generated by MontiCore, for example, offers means to compose languages efficiently [34,37].

Requirements for Code Generation. Generating code from a process model will not produce an expected outcome if the model does not adhere certain Context Conditionss (CoCos) [37] and it will fail, e.g., if the transformation of the model into code will produce a deadlock. To enable efficient modeling, we *require* automatic verification of soundness and such CoCos. Generating the entire code base of any software system from models is not possible [67]. In general, the generated code base needs handwritten extensions to provide a fully functional system. This *requires* mechanisms that enable developers to integrate handwritten code with the generated code. MontiCore, for example, offers mechanisms to integrate handwritten code efficiently [30].

The next section introduces the generator framework MontiGem which already provides a rich set of code generating functionality for EISs and thereby provides a powerful tool for efficient model-driven and generative engineering of these systems. Furthermore, it allows to implement extensions such as generators and languages for process models, efficiently. Reusing MontiGem, therefore, follows our vision of integrating generative EIS engineering with generative engineering of business processes. Developing and integrating code generators for process models into the code generation procedures of MontiGem further enables to reuse existing methodologies for generative engineering of EISs, that have proven effective [2].

3 Preliminaries

In the following sections, this paper presents an approach that enables agile generative engineering of PAISs and follows the vision proposed previously. For implementing the modeling languages, we used the language workbench MontiCore and adapted the generator framework MontiGem [2,3]. The process-awareness of the generated application is established by including BPMN models during the generation process. This section, therefore, provides fundamentals on the concepts of MontiCore, MontiGem, BPMN and

a running example which will serve to illustrate our approach throughout the paper.

3.1 MontiCore

The language workbench MontiCore [33,35] is a tool for engineering compositional, textual (modeling) languages. Therein, engineers specify a language's concrete and abstract syntax as context-free grammars in an integrated way. Model checking and transforming models in the language into code are greatly facilitated by the model processing infrastructure, which includes, e.g., a parser, generated by MontiCore.

So far, MontiCore languages have been applied for MDSE in multiple domains including automotive, cloud, smart home, robotics and software engineering. The UML/P (UML/P) language family [64], a subset of the UML that is suited for code generation, has been implemented with MontiCore. The UML/P together with the methodologies proposed in [65], provide the linguistic and methodological foundation for pervasive generative engineering of software products.

To support the adaption of generated code, MontiCore provides the TOP-mechanism [37]. The mechanism relies on the object-oriented principle of inheritance to include handwritten extensions of generated classes. The TOP-mechanism checks for such handwritten files during the generation process and generates the code such that the handwritten code is always used instead of the prior generated code. In detail, it checks if there is already a handwritten source for a given class and renames the generated file. Thus, the application always uses the handwritten extension. Using MontiCore, allows to separate the generated code from the handwritten code. Continuous re-generation without loss of information is thereby possible.

3.2 MontiGem and its DSLs

MontiGem [2,3], a generator for EISs, combines multiple transformations and code generators to create a widely functional EIS from a set of input models. It uses templates in the target language(s), i.e., Java, Typescript and HTML, as well as models from different DSLs as input. Supported languages are UML/P CDs [64] to describe data structures, the OCL (OCL/P (OCL/P)) [64] to specify restrictions on the data, GUI models [29] to specify user interfaces, or the Tagging Language [31] to enrich model elements with additional information, e.g., platform specific data to concepts from the domain model (see Figure 1).

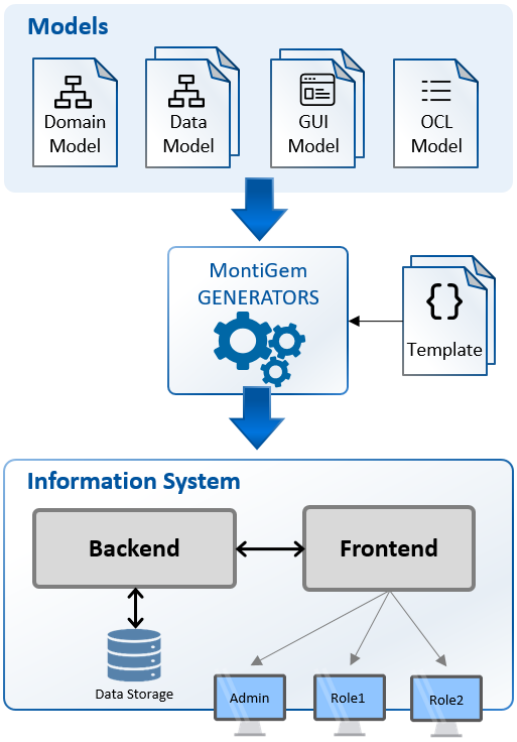

Fig. 1 The generation process with MontiGem.

The framework supports generating code from models in these DSLs: It generates the code that represents data structures from UML/P CDs, the code that implements the functionalities for data validation from OCL/P constraints, the code that implements GUI pages in HTML and Typescript as described in the GUI models, and the functionality for communication between the Java back-end and HTML/Typescript front-end [28]. MontiGem provides a Run-time Environment (RTE) to support the basic infrastructure for the application. This includes, e.g., GUI components, communication infrastructure, a security manager and database access. The RTE can be configured to allow for customization for generated applications. MontiGem uses a multitude of MontiCore languages to generate a range of application elements and provides means to adapt generated code with MontiCore's TOP-mechanism [37]. The use of this combination of multiple languages can produce a variety of different application parts and minimize the effort as less handwritten code is needed.

MontiGem is used in the real-world project MaCoCo for financial management [28], for creating digital twin cockpits [15], and to support the engineering process of wind turbines with digital twin cocpits for parameter management [50]. We use it in projects to create low-code development platforms for digital twins [14], on

goal modeling in assistive systems [48], and privacy-preserving information systems [47].

3.3 Business Process Model and Notation (BPMN)

Business Process Management supports the design, enactment, management, and analysis of business processes [74] which enables agile and efficient adaptation to market needs and changes. The de-facto standard [12] BPMN [57] provides a graphical notation that is intuitive to business users yet expressive enough to capture the technical details of complex business processes.

BPMN [57] categorizes its graphical elements as *flow objects*, *connecting objects*, *data*, *swimlanes*, and *artifacts*. Flow objects are the main building blocks of BPMN models and are linked through connecting objects. They encompass *activities*, *gateways*, and *events*. Data captures the physical or digital items that are created, accessed, or updated during a process. Swimlanes act as containers to organize and categorize activities, e.g., by functional departments or organizational roles. Artifacts display supporting information, such as comments. Each basic category has variations to cope with the complexity of business processes. For modeling data and expressions, BPMN foresees the use of XML Schema[1] and XPath[2].

3.4 Running Example using BPMN

In the following sections, we consider a *quality assurance process of a manufacturing plant* as a running example for considering processes in an information system. This can be seen as an extension of a data-centric information system that provides staff and contract management of a mechanical engineering department of the university as well as the material and resource management of the associated demo factory. The running example is used to explain the textual version of BPMN and in an example application for validation purposes of our generative approach.

During the daily commissioning of a manufacturing plant for gear shafts, samples are produced for quality assurance. After powering up the manufacturing plant, the engineer needs to adjust the control parameters of the plant. These parameters influence the quality of the manufactured goods. The engineer determines the parameters by running simulations of the production process. Once a suited set of parameters has been determined, the plant produces the shafts and bearings seats. Meanwhile, the engineer records the calculated parameters. The plant then measures the produced samples. If the tolerances are not met, the engineer must re-evaluate the parameters and new samples must be produced. If the tolerances are met, and if it is Friday, the engineer creates a weekly report before the plant goes into regular operation. Figure 2 shows a model of the process in the graphical notation of BPMN 2.0.

[1] `https://www.w3.org/standards/xml/schema`, last accessed: 31.07.2020

[2] `https://www.w3.org/TR/xpath20/`, last accessed: 31.07.2020

4 A Textual BPMN Notation for MontiCore

To make BPMN models amenable for code generation with MontiCore, we developed a textual notation for BPMN. The notation covers *private* (executable) BPMN processes, i.e., processes within a single organization (as opposed to processes spanning multiple organizations, which are modeled by *public* BPMN processes). In addition to the graphical elements, the textual notation includes non-graphical attributes usable for code generation such as formal conditions. BPMN designates XML Schema and XPath as the default data modeling and expression language [57]. This hinders code generation, as, e.g., types therein are tied to the lifecycle of the parent process or sub-process. Our approach, therefore, explicates constraints on classes and associations as UML/P CDs accompanied by OCL/P constraints [64,65]. Therein, types persist beyond the scope of the process, which enables to reference data items by their names within a textual BPMN model which eliminates the graphical notation's need for *data associations*.

To implement this within the textual BPMN, we took advantage of MontiCore's mechanisms for systematic language composition [33,35]. Listing 1 shows the running example introduced in Section 3.3 in the textual BPMN notation. The main difference are the use of defined data types (l. 2, 14, and 25), the separation of the tasks (ll. 5-18 and 22-29) and the control flow (ll. 32-48).

Listing 1 Example process in the textual BPMN notation.

```
1  process ProduceGearShaft {
2    store params:ShaftParameters;
3
4    lane Engineer {
5      task manual PowerUpPlant;
6      task user AdjustParameters {
7        out: params;
8      }
9      task script SimulateParameters until:[true] {
10       in: params;
11     }
12     task user RecordParameters;
```

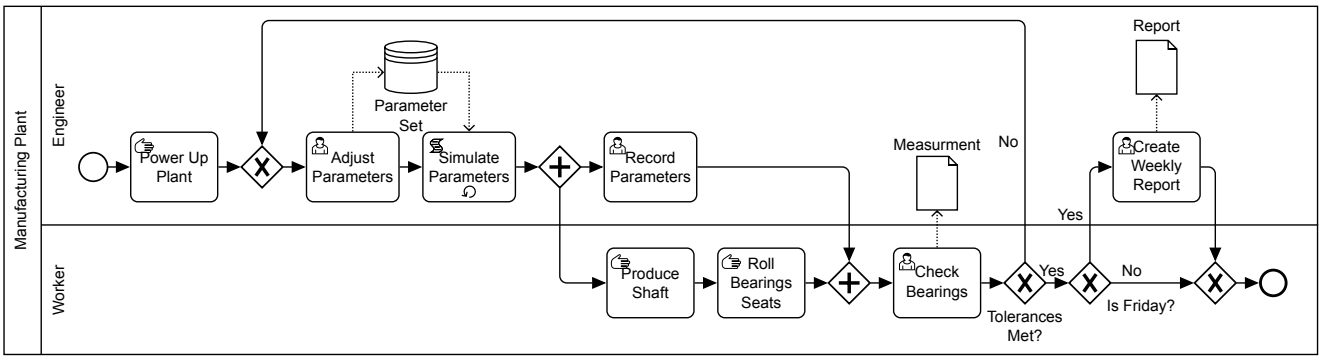

Fig. 2 Running example: daily commissioning of a manufacturing plant.

```
13
14      data report:WeeklyReport;
15
16      task user CreateWeeklyReport {
17        out: report;
18      }
19    }
20
21    lane Worker {
22      task manual ProduceShaft;
23      task manual RollBearingSeats;
24
25      data measurement:BearingMeasurements;
26
27      task user CheckBearings {
28        out: measurement;
29      }
30    }
31
32    merge xor RepeatIfAccuracyNotMet;
33
34    split xor CheckIfFriday;
35    merge xor CheckIfFridayMerge;
36
37    event start -> PowerUpPlant ->
          ↪ RepeatIfAccuracyNotMet -> AdjustParameters
38      -> SimulateParameters -> split and -> {
39      RecordParameters,
40      ProduceShaft -> RollBearingSeats
41    } -> merge and -> CheckBearings -> split xor -> {
42      [measurement.result  > 0.02]
            ↪ RepeatIfAccuracyNotMet,
43      [measurement.result <= 0.02] CheckIfFriday
44    };
45
46    CheckIfFriday -> [false] CheckIfFridayMerge;
47    CheckIfFriday -> [true ] CreateWeeklyReport ->
          ↪ CheckIfFridayMerge
48      -> event end;
49  }
```

In our particular setting, the textual notation had several advantages over graphical notations, to which, among others, belong, enhanced conciseness, integratability with existing developer tools and better support for version management systems.

4.1 Syntax

A `process` (see l. 1 in listing 1) contains the elements of the process and may use `lanes` to group elements (l. 4). Activities use the keyword `task` (atomic activity, l. 5) or `sub-process` (compound activity). Tasks may specify a task type (user, service, etc.), a looping behavior for loop or multi-instance activities, or further task-specific attributes. Regular sub-processes, as well as their variants (event-based, transaction, and ad hoc), are supported. Gateways either `split` (l. 34) or `merge` (l. 35) sequence flows. Mixed gateways, which merge and split paths at the same time, are not supported. The gateway type, i.e., the split or merge behavior, is specified by the keyword `xor` (exclusive), `ior` (inclusive), `and` (parallel), `event` (event-based), or `complex`. Events use the keyword `event` (l. 38) and may specify an event type (start or end; intermediate if omitted). The event behavior is controlled by the keyword `receive` (catch event) or `send` (throw event), followed by the trigger that is being received or sent. BPMN supports data objects (`data`, l. 25) tied to the life-cycle of the parent process and data stores (`store`, l. 2) which persist beyond the scope of the process. Data and payloads carried by event triggers (messages, signals, errors, etc.) have a name and a type. Types are captured as a UML/P CD [64], which, as mentioned above, makes them persist beyond the lifecycle of the parent process or sub-process.

Activities and events, then, specify inputs and outputs by referencing the corresponding data items by their names.

A sequence flow connects two flow objects (l. 46). It specifies the name of the source node and the name of the target node, separated by an arrow '->'. Multiple sequence flows can be chained to create a path (ll. 38–44). In the case of a conditional flow, the condition is specified within curly braces next to the target of the sequence flow.

While activities can only be referenced by name, events and gateways can also be defined in-lined in the sequence flow (l. 38). In-lined elements are anonymous, i.e., they do not have a name and cannot be referenced by other sequence flows. Lastly, so-called block structures enable the definition of structured process parts. A block consists of multiple branches. The branching behavior is controlled by the flow node preceding the

block, e.g., if a parallel gateway precedes the block, all branches are executed (in parallel). In contrast, if an exclusive gateway precedes the block, only one branch is executed. Branch conditions are evaluated to determine which branch should be executed. In case a block is not preceded by a gateway, BPMN uncontrolled flow semantics apply [57, p. 32]. Similarly, the flow node following the block controls the synchronization behavior of the branches. By combining sequence flow chaining, in-lined events and gateways, as well as block structures, it is thus possible to describe complex and arbitrary structured sequence flows in a concise manner.

4.2 Model Validation

CoCos impose restrictions on a language's set of valid sentences [37], e.g., the source and target of a sequence flow. BPMN specifies relationships between elements, but does not define a formal notion of *soundness* [76]. The concrete and the abstract syntax of the textual BPMN notation are specified using MontiCore, which generates model-processing infrastructure, including support for checking CoCos [37]. Based on this, we check BPMN models in three stages for (1) well–formedness, (2) structural, and (3) behavioral CoCos. A stage is only executed if the previous stage passed without errors since checks in a stage may require properties checked in a previous stage. Moreover, later stages are computationally more expensive.

(1) Well-formedness CoCos subsume the BPMN interaction rules and syntactic constraints [57], e.g., flow conditions must evaluate to a Boolean value, referenced elements must exist and the type of a data element must exist. Moreover, the restrictions specified by the BPMN standard are checked, e.g., when the use of one element or attribute requires or prohibits the use of another element or attribute, restricted in its number, or when only certain elements can be connected by a sequence flow. More than 50 CoCos restrict the set of valid BPMN models.

(2) Structural CoCos detect violations of the interaction rules and structural anomalies. Static analyses suffice to check this type of context condition, i.e., executing or simulating the BPMN model is not necessary. Structural anomalies can be classified as *deadlocks*, *lack of synchronization*, *infinite loops* and, *dead activities* [40]. They typically result from a mismatch between an upstream split gateway and a downstream merge gateway. For example, a deadlock occurs if a parallel gateway is used to merge flows that have previously been split using an exclusive gateway, thus causing process execution to block partly or entirely. In contrast,

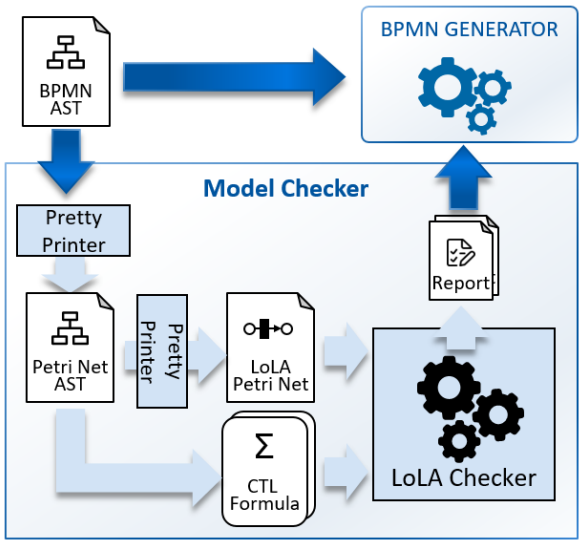

Fig. 3 Checking soundness for BPMN models.

failing to join (parallel) flows leads to duplicated execution of downstream process parts, referred to as lack of synchronization. Anti-patterns typically can only be used with block-structured processes [42]. Our implementation detects anomalies by scanning the process graph for *anti-patterns* (see [59,43,41]) and supports arbitrarily structured processes. We use an extended detection algorithm that eliminates false positives and false negatives. For example, it correctly detects that an exclusive gateway lying on a path from a parallel split gateway to a parallel merge gateway leads to a deadlock at the merge gateway as the execution may exit the path and, thus, not reach the merge gateway (false negative), see Figure 4.

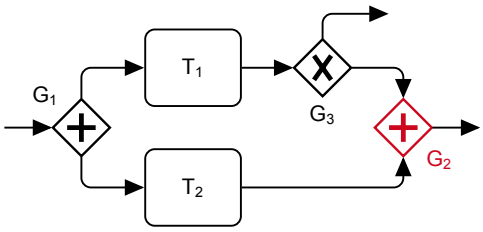

Fig. 4 Structural CoCos: Exclusive split gateway G_3 can cause the flow to exit the parallel control structure causing a deadlock at the parallel merge gateway G_2.

(3) Behavioral context conditions apply Petri net-based model checking to ensure formal soundness [32] of the BPMN model. Petri nets [63,52] are a natural candidate for this task due to the flow-oriented nature and the (informal) token-based semantics of BPMN [76]. The notion of soundness defined in the context of Petri nets [60,63] has been transferred to BPMN [81]. A pro-

cess is *sound* if (i) a process instance can always complete, (ii) once a process instance completes, all activity instances have completed, (iii) there exist no activities that can never be reached [76].

To prove soundness, we transform the BPMN model to a Petri net and generate a set of Computation Tree Logic (CTL) formulas [13] that ensures soundness of the Petri net and, thus, of the BPMN model (see Figure 3). Formally the set of CTL formulas that need to be fulfilled are similar to [24, 23]. Their fulfillment implies the absence of deadlocks and livelocks which implies soundness properties (i), and (ii) as well as liveness of the WF-net which implies the soundness property (iii). The BPMN model is, therefore, sound, iff the Petri net obtained from the BPMN model satisfies the generated CTL formulas. Essentially, verifying condition (i) comes down to verifying soundness comes down to verifying liveness of the final marking and absence of dead transitions. If either one is not satisfied by the WF-net, the BPMN model is not sound. We use the model-checker LoLa [79] for verification. The result enables to identify errors in the BPMN model that cause dissatisfaction of the set of CTL formulas.

The transformation is an adaptation of [17] for BPMN 2.0 and enables the independent checking of sub-processes as depicted in Figure 3. The transformation takes a BPMN model as input, which is parsed to obtain its Abstract Syntax Tree (AST) which is then transformed into a Petri net AST. More precisely, the resulting Petri net is a WF-net [72, 71], i.e., a specific kind of Petri net that is commonly used to formally represent and verify correctness of workflow processes [76]. The transformation algorithm is an extension of [17] that supports also, e.g., non-interrupting boundary events, which were introduced in BPMN 2.0 [57]. From the WF-net AST, we use a pretty printer to obtain a LoLa-specific representation which is structurally very similar to the Petri net AST generated from the BPMN AST, as well as the CTL formulas that assure soundness. The implementation hands the LoLa-specific WF-net representation and the generated CTL formulas to the LoLa checker. The result is a Boolean, telling whether the input WF-net, and thus, the BPMN model, are sound, i.e., whether the WF-net satisfies the input CTL formula.

4.3 Language tooling

The implementation of the textual BPMN includes additional tooling that facilitates developing functionalities to generate code from textual BPMN models. The class `WorkflowTool` in Figure 5 encapsulates the functionality provided to code generator developers.

By means of method chaining, developers decide which steps are necessary for processing BPMN during a model-to-code transformation. The methods provide the following functionalities for processing BPMN models:

- **Loading the model:** The method parses a provided BPMN model, and creates the corresponding AST as well as the symbol table [37].
- **Check CoCos:** This method performs an automatic check of the CoCos explained in the previous section.
- **Transformation:** Using this method, the application developer can modify, add, or delete nodes in the AST of a BPMN model. An example for such a transformation is the replacement of sub-processes by (atomic) tasks.
- **Adding Imports:** This method allows to add import statements to the generated code, e.g., for integrating a workflow execution engine.
- **Get AST or Symbol Table:** This method simply returns the created AST or symbol table.
- **Export to XML:** This method stores the BPMN AST as a model in the BPMN 2.0 XML exchange format.
- **Writing Auxiliary Models:** Writes additional models, such as CDs as specified in the BPMN model.

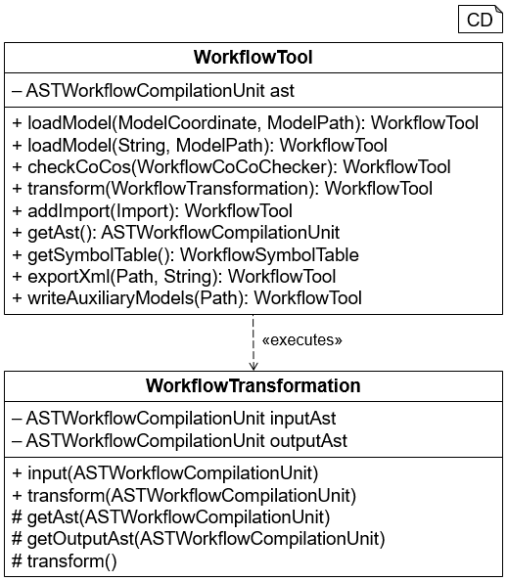

Fig. 5 The central classes for loading and manipulating BPMN models.

The class `WorkflowTransformation` operates on an input AST of a BPMN model and its symbol

table resulting in a transformed output AST which can be used in further steps, e.g., for code generation.

The next section introduces a generative approach that integrates the BPMN DSL and its tooling to create a PAIS. Full coverage of the BPMN standard is not necessary for this generation process. Minimally, we require the BPMN DSL to include human and automated tasks, i.e., user and service tasks, as well as basic gateways and events.

5 Generating Information Systems from Process Models

An EIS includes multiple different application parts, which need to be implemented. To better support the users with their tasks, a process model defines viable behavior of the system and the user during the process. For generating the code base of a PAIS these models can be used in two different ways: (1) interpreting the model during run-time and (2) using the model's information during compile-time. The interpretation of the model yields a process which the application executes while offering the user a guideline, what operations are viable in the current state according to the process model. Process engines are the common choice for automatically executing process models. In a generative approach to PAIS development, a generator uses the information provided by the model to create infrastructure, add resources needed during a process step, GUI pages, and provide access to the application's data structure and data storage. The generated PAIS supports both human and automated activities. Still, the developer needs to provide the implementation of the business logic.

This section outlines how to extend MontiGem to enact process models and extended the generated information system by workflow functionality. The extension includes additional transformations in the existing generation process to allow for interoperability with the existing components and extendability of MontiGem. With these additions, a *process-aware* information system can be generated.

5.1 Architecture

The generated PAIS is split into a 3-tier architecture [2] which is illustrated in Figure 6. The back-end (1) comprises the application logic (1a) and the Camunda workflow engine (Camunda BPE)[3] (1b) for enacting the modeled processes, providing the execution of service tasks that can be automated. The BPMN models are exported to the BPMN 2.0 XML exchange format and executed by the process engine at run-time. The application logic queries the process instances and engages with them via the services offered by the process engine. The process engine is responsible for steering the process instances. Camunda BPE is embedded as a dedicated component, so it can be *easily exchanged* with similar process engines. The business logic is part of the application logic (1a). When the process execution reaches a service task, the process engine calls the appropriate implementation within the separated application logic. There exists an application database (2a) and an independent database for the process engine (2b) to store its internal state, including the state of process instances. Persisting the state of the process engine enables to restart or pause the back-end and resume the execution of process instances started earlier.

The generated PAIS supports both human and automated tasks, i.e., user, and service tasks, as well as basic gateways and events. It supports the evaluation of OCL conditions and makes use of the application data. To support user interaction, the front-end (2) contains parts that supply status information of the current running process and requests additional information provided by the user. The front-end features a process list, a task list, and task pages to provide the outputs of user tasks. Tasks can be assigned to either individual users or user groups. The user can select tasks and provide the required data. The communication between front-end and the process engine is channeled through the application logic. Thereby the front-end remains independent from the process engine during run-time, and the application logic can apply further filter or validation logic. To transmit the data and trigger process-related actions, the extended MontiGem generator framework generates the corresponding Data Transfer Objects (DTOs) and commands in front-end and back-end.

5.2 Generation process

We separate the *generation process* in two main parts: Figure 7 shows the main steps including two separate generators in detail. A two-step generation strategy has several advantages over a direct generation of the final artifacts. At the conceptual level, it enables the reuse of existing (platform) abstractions and, thus, facilitates the specification of the system. It facilitates handling the resulting models in generator one, as those are handled the same way the models written by the user are

[3] `https://camunda.com/products/bpmn-engine`, last accessed: 31.07.2020

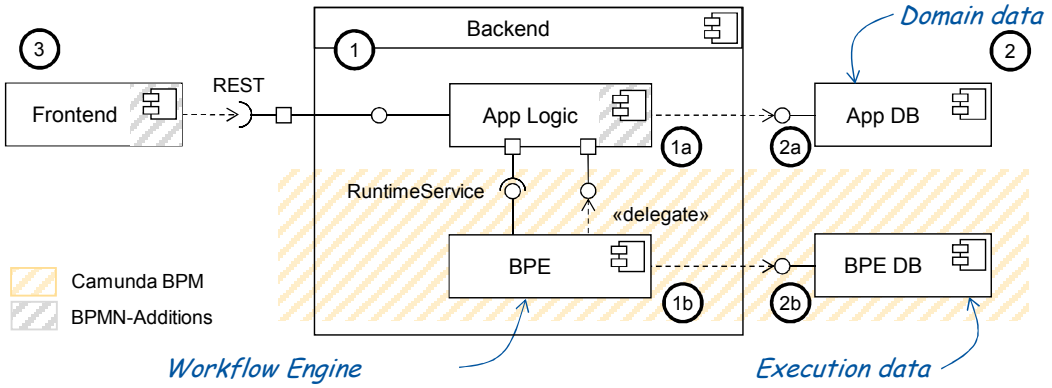

Fig. 6 High-level architecture of the generated PAIS. Separated in (1) back-end, (2) database, and (3) front-end.

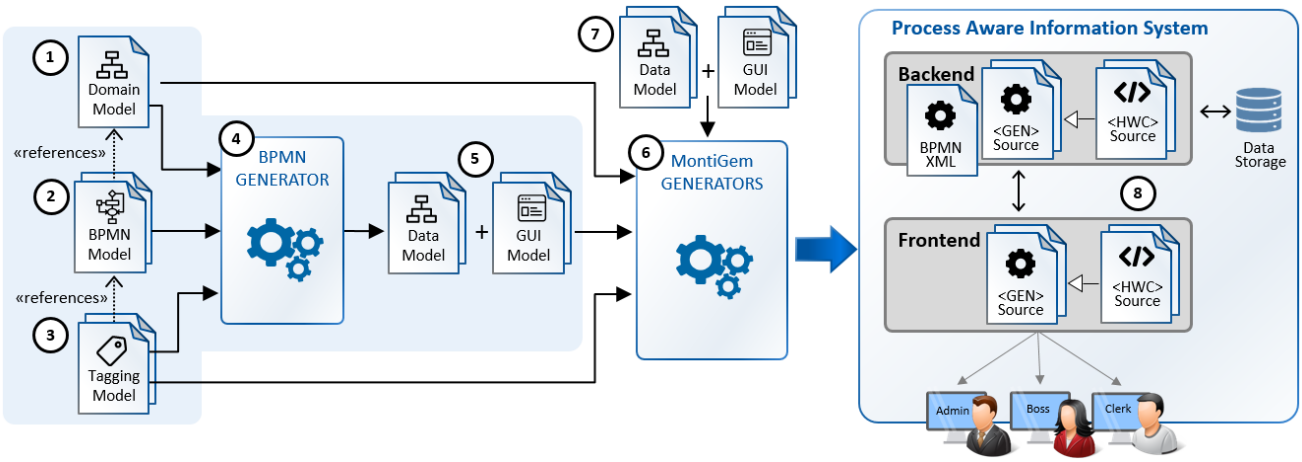

Fig. 7 Overview of the 2 step generation process and artifacts.

handled. At the technical level, it enables using multiple model sources at once and the reuse of existing code generators, resulting in higher productivity and more reliable software.

The BPMN generator (4) is responsible to process all necessary model files (1-3): It takes the CD domain model (1) defining the data structure, BPMN models (2) defining the processes and tagging models (3) defining roles as input and produces for each of the given BPMN models (2), one or more CD data models and GUI models (5). The GUI models and CDs are process-specific: For each user task, a GUI model is created, that describes the corresponding task page. To support the communication between the front-end and the back-end further data structures are generated as CD data models, e.g., DTOs for the inputs and outputs of user tasks.

In parallel to the generation process, an optional model check is possible (see Figure 3). The results of this check are shown to the developer.

In a second generation step, the MontiGem generator framework (6) is used to generate the PAIS based on the domain model (1), further domain specific models (7), and the generated models from the first step (5). The PAIS consists of a back-end, front-end, and databases. The back-end includes the generated classes and the BPMN 2.0 XML process descriptions derived from the BPMN models which are executed by the process engine at run-time. The domain CD is used to generate the data structure for the particular domain. GUI models describe the contents and layout of the pages for the generated front-end. Additionally, business logic, e.g., service task implementations, and connections between the process data and the stored domain data, has to be provided by handwritten code (8). Any generated code can be extended by handwritten code in the respective language using the TOP-mechanism [37] (see subsection 3.1). For further details on this second generation step, we refer the reader to [2,29].

Only little intervention from the application developer is required to get the generated PAIS up and running, i.e., developers need not provide handwritten lines of HTML or CSS code. The strategy is consistent with MDSE principles. The intermediate models gener-

ated in the first step are platform-independent. Hence, when targeting different platforms, the model transformations in the first generation step can be reused as-is and only transformations in the second step have to be adapted. Moreover, the developer can overwrite a generated model by placing a handwritten model with the same name which remains untouched during following re-generation processes. This allows for agile and iterative application advancement.

To further automate the generation process, we use Tagging [31] in addition to the textual BPMN models and CDs for (1) the automatic assignment of user tasks to system users or roles, which restricts the group of people who are allowed to perform a task, and (2) the customization of the generated GUI forms for user tasks.

The interpretation of the process models during runtime provides full control over the available tasks a user can work on. The process models complement the application's logic and provide the means to model it.

Adding additional BPMN models during run-time is possible if they meet certain requirements, such as using only existing data types and input GUI. Otherwise the data type for a resource could not be handled by the application. Such models have to be provided during compile-time, so that the type and GUI can be generated (see section 8).

6 Validation by Example: Manufacturing

For validation, we apply our approach to the manufacturing process introduced in subsection 3.4 and additional examples from organizational processes, e.g., the approval and canceling of holiday requests. Such a variety of processes are typical for overarching systems that handle different aspects of an application. We have demonstrated the feasibility of our approach by implementing an additional generator (see 4 in Figure 7) which works together with the generator framework MontiGem. As MontiGem is already used in real-world full-size projects [28], its practicability is already demonstrated.

Domain Model. One central element for the generator is the domain model, as it defines the domain of the application, e.g., the types used for the `ProduceGearShaft` process. A goal of the BPMN generator is to extend the input domain model with additional types defined in the BPMN model (Figure 7 step 5). Our textual BPMN notation therefore allows not only the use, but also the definition of new data types.

Listing 2 Running example: Excerpt of the class diagram.

```
1  classdiagram Domain {
2    class Gearbox;
3
4    class ShaftParameters {
5      int numberOfTeeth;
6      String material;
7      ...
8    }
9
10   class BearingMeasurements {
11     Double deltaX;
12     Double deltaY;
13     Double deltaZ;
14     Double result;
15   }
16
17   composition [1] Gearbox ->
18     (gearShaft) ShaftParameter [1];
19   composition [1] Gearbox ->
20     (bearing) BearingMeasurements [1];
21 }
```

An excerpt of the UML/P CD [64] of our running example is shown in Listing 2. The CD shows some of the classes that are necessary for the `ProduceGearShaft` process. The syntax is oriented on Java. Three classes are defined (ll. 2, 4, and 10) as well as their attributes. Additional associations (ll. 17-20) are defined to create a connection between the classes. The language allows for underspecification to simplify the usability.

BPMN Model(s). For each supported process a BPMN model has to be created. From the BPMN models (see Listing 1 for one of the processes), our generation process including the two generators creates a PAIS that supports both the engineer and the plant workers in executing the process. This role information can be specified by additional tagging models like in Listing 3.

Tagging Model. In BPMN, lanes are purely informative; their meaning is up to the modeler [57]. At runtime, however, it must be clear which system user or system role is responsible for executing the given task instance. Tagging allows to add extra information to a given model. This is used to define additional information without changing the original model and can be used by a generator. The separation of the information leads to simpler models and different additional tagging models can be used in different contexts. We provide this environment-specific information through a resource tagging model. Resource tags can be applied to tasks and lanes. When applied to a lane, the resource assignment applies to all tasks within the lane.

Listing 3 Resource tagging model

```
1 tags ResourcesTags for ProduceGearShaft {
2     tag Engineer with Initiator;
3     tag   Worker with Role = "admin";
```

```
4 }
```

In Listing 3, the lane `Engineer` is tagged with the resource tag `Initiator`. This ensures that instances of user tasks contained in the lane `Engineer` are automatically assigned to the user (the engineer) who started the corresponding process instance. Furthermore, the lane `Worker` is tagged with the resource tag `Role` and a value of `"admin"`. Thereby, each member of the system role `"admin"` is able to claim and complete instances of user tasks contained in the lane `Worker`. By specifying resource assignments through a separate tagging model, the BPMN model remains clean and reusable in different environments (by providing different resource and form tagging models). By specifying resource assignments through a separate tagging model, we avoid mixing environment-specific and environment-agnostic information within the BPMN model.

Generated GUIs and functionality. From the task definition in the BPMN models and the domain class diagram (Listing 2 ll. 10-15), GUI models for user tasks are automatically generated and then used by the MontiGem generator to generate GUIs in the resulting PAIS. The generated form in Figure 8 shows an example where the user is asked to fill out a form for the particular user task, namely entering the measurement results when checking the bearings. This streamlines the generation of user tasks, allows for easy interaction and leads to the user entering all the required information.

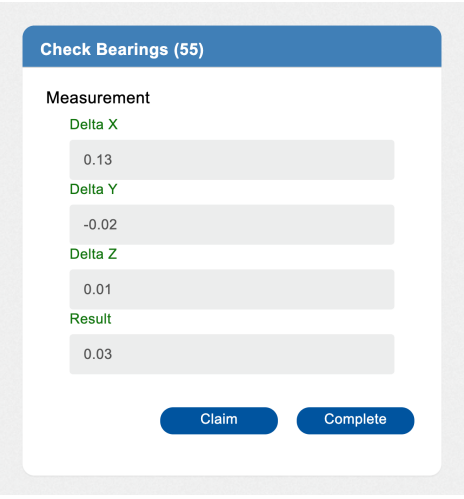

Fig. 8 Example: Task form for entering bearing measurements.

Additionally, a *process list* is generated which shows all processes the current system-user is allowed to start. There, it is also possible to start a new case (instance) of a process, e.g., to produce a new gear shaft.

A *task list* (see Figure 9) shows pending task instances (of any case) assigned to the user directly or a group of which the user is a member (*My Tasks* and *My Group Tasks*) and completed task instances (*Completed*). The user can select pending task instances to complete them. For group tasks, a user may also claim and drop task instances.

Comparison of handwritten and generated lines of code. The generated PAIS requires only little developer intervention to be operational. Table 1 compares the numbers of handwritten, generated, and run-time artifacts and lines of code. Handwritten artifacts are provided by domain experts and developers. Generated artifacts are derived from the handwritten input models and the generated intermediate models, and run-time artifacts are shipped as part of every generated application. Consequently, the number of generated artifacts grows with the number of input models and their complexity, while the number of run-time artifacts remains constant. In Table 1, the generated PAIS includes three processes, the manufacturing process and two processes for the approval and canceling of holiday requests. Overall, we manage to generate $\frac{9000-170}{9000} \approx 98,1\%$ of the back-end code and $\frac{20800-43}{20800} \approx 99,8\%$ of the front-end code (excluding run-time artifacts).

To sum up, the example shows that the addition of BPMN models resulted in a high amount of code that can be generated. It reduces the need for handwritten GUI models, as GUI models for user interaction are additionally generated and used in the second generation phase (cf. Figure 7).

The BPMN generator allows for the adaption of existing applications generated with MontiGem and provide an easy to use approach to define business processes using BPMN models.

7 Related Work

To compare our approach to others, we have investigated other BPMN and behavior languages as well as model-driven approaches for workflow and process engineering. Moreover, we discuss the limitations and strengths of our approach.

7.1 Process and BPMN Languages

Modeling languages for business processes, exist in a broad variety. The standards for BPMN [57] and UML activity diagrams [58] with their graphical notations are probably most widely known. Application domains For

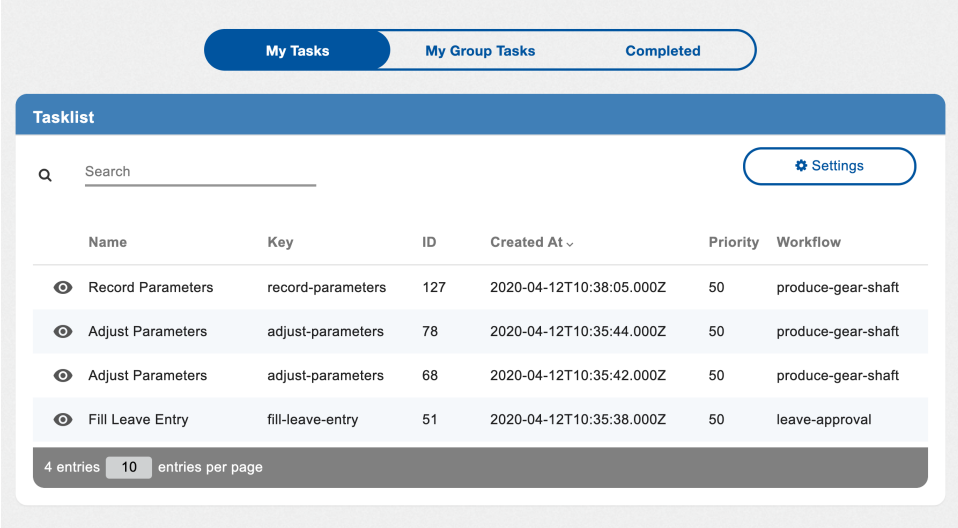

Fig. 9 Example: Task list with pending user tasks for the production of a new gear shaft.

Table 1 Numbers of handwritten, generated, and run-time lines of code (number of artifacts in brackets).

	Models					Back-end	Front-end	
	CDs	OCL	GUI	BPMN	Tags	Java	Type-Script	HTML CSS
handwritten	53 (2)	–	37 (1)	72 (3)	34 (3)	170 (14)	43 (1)	–
Generated	369 (11)	9 (1)	524 (9)	–	11 (2)	9000 (212)	20800 (419)	4600 (70)
Run-time	45 (6)	–	523 (7)	–	–	31200 (477)	14600 (426)	2700 (15)

BPMN there exists an Extensible Markup Language (XML)-based exchange format which enables to implement transformations from graphical to textual representations. Therefore, this section reviews only textual implementations of business process modeling languages and, in particular, of BPMN.

Process modeling languages with a textual syntax exist. These do not aim to implement the BPMN standard and are often applied for creating web services composition specifications [78,82], rather than for code generation: Examples are the Business Process Modeling Language (BPML) [5], the Business Process Execution Language for Web Services (BPEL4WS) [55], or the XML Process Definition Language (XPDL) [80]. TN4PM [51] covers common elements of BPMN, UML activity diagrams, and Role Activity Diagrams. The notation is inspired by the simulation language GPSS/H and uses the concept of entities but is rather technical due to programming constructs like *if-then-else* and *goto*, it only deals with the control flow aspect of process modeling and coverage of BPMN is limited.

Textual implementations of BPMN are rather rare. The ones that exist are not suited for a model-driven and generative approach to engineering PAIS that is integrative in the sense of section 2: The textual BPMN-representation of Urzica, Tnase and Florea [70] facilitates the mapping of BPMN processes to agent specifications. The notation supports only a few, further restricted BPMN elements. It lacks support for data and expressions, does not support graph-structured processes, and serves as an intermediate language that is not optimized for business stakeholders. Nalepa, Kluza and Ciaputa [53] propose a textual BPMN notation for collaborative process modeling in the context of a semantic wiki. The notation represents BPMN processes in an object-like syntax similar to JSON, with keys and values. It is considered easy to read, and coverage of BPMN is considered high but lacks support for data objects. S-BPM DSL [38] is a textual notation for Subject-Oriented BPM (S-BPM), which is based on the subject-predicate-object pattern of sentences in natural language. The set of language elements is much smaller than in BPMN: It lacks

modeling concepts such as events, data, and (formal) expressions. The structure of models in S-BPM DSL is comparable to models in our BPMN notation. S-BPM DSL is implemented as an embedded DSL using Scala. A textual notation of BPMN with the aim to reduce modeling efforts to allow for live modeling during meetings is proposed in [39]. The language is also a DSL in the sense that it covers those parts of the BPMN standard relevant for the application domain of the language. The target application of the language, however, is not code generation. The plantBPMN [27] is a textual BPMN notation created with Xtext[4] and similar to ours. The notation plantBPMN focuses on public processes [57], i.e., process models with multiple pools. In contrast, our notation focuses on private executable processes [57], i.e., process models with a single pool. While plantBPMN has high coverage in terms of graphical BPMN elements, non-visible properties were rarely included [27]. In contrast, our notation includes additional information essential for code generation, e.g., non-visual attributes, such as conditions and data types.

Besides languages related with BPMN, there exists a variety of other textual (software) process languages and with a broad range of application areas. PML [54] is an early process scripting language. PML is intended to model scripted processes comprising people and tools. It features basic control-flow constructs and embeds scripts, e.g., HTML markup for manual actions or Perl scripts for automated actions. PML has no control flow conditions and does not support graph-structured processes. WebWorkFlow [36] is a high-level textual language for describing workflows. A workflow consists of multiple procedures which can be composed. Possible compositions of procedures are sequential, parallel, iteration, or race condition. However, graph-structured processes are not supported and concepts in WebWorkFlow broadly differ from BPMN. The Workflow Definition Language (WDL) is another textual workflow language in the context of the workflow management system Panta Rhei [21]. WDL is comprehensive, but it does not support graph-structured processes and the concepts differ from BPMN. The Information Systems Modeling Language (ISML) [62] allows for conceptual modeling and verification of information systems. The language includes information models (set theory and first-order logic) as well as process models (Petri nets with identifiers) and uses an automated theorem prover. Code generation is not in focus of this language. An overview of textual process modeling languages or tooling that works with textual notations to extract process informations is given in [39] or [10].

[4] https://www.eclipse.org/Xtext/

7.2 Model-Driven Workflow and Process Engineering

Process models include manual and automated activities (tasks). Existing approaches can be divided into supporting either automated activities (automation-focused) or the process participants in carrying out manual activities (user-centric). Usually, the former fail to provide a suitable user interface, while the latter do not consider the interaction with external applications and business partners [69].

Other approaches which combine code and processes are, e.g., iTask [61, 49] and ExSpect [73]. iTask generates a workflow management system from declarative specifications. However, it lacks a number of key features to make it suited for programming GUI applications. ExSpect is a simulation and animation tool for hierarchical timed colored Petri nets with priorities. Simulation is not in focus of our work and we rely on a web technology for GUIs.

There exists a variety of approaches alongside *Service-Oriented Architecture (SOA)* research, which consider a mapping from BPMN to SoaML, a standardized UML profile for modeling services within SOA. Service-oriented frameworks for BPMN, e.g., MINERVA [16], generate platform-specific service implementations from a BPMN model. Nevertheless, it lacks a solution on how to handle BPMN manual activities in a web application. Fazziki et al. [25] aligns SOA and BPM with a MDSE approach. BPMN models and behavioral UML diagrams are mapped to a component model. However, the approach lacks information about the translation into code and on how the components interact to accomplish the process behavior. Chaâbane et al. [11] proposes the BPMN extension BPMN4SOA for specifying web service invocations and data object manipulations in a platform-independent way in the BPMN model and provides code generators to Java and BPEL but their approach does not allow to include arbitrary business logic or hand-written additions.

Other approaches try to *derive (web) applications from BPMN*. The WebRatio BPM platform [7] is a commercial tool-suite to create process-oriented web (and mobile) applications based on Java EE. It combines BPMN for process modeling, WebML for application modeling, and UML CDs for data modeling. In contrast to our approach, WebRatio BPM covers only a small subset of BPMN elements, uses an extended BPMN notation, and does not use a process engine for managing the execution of process instances. Loja et al. [44] discusses a generated PAIS using three purpose-built meta-models: a business domain, a user interface, and a business process meta-model and presents a prototypical process engine to enact

the modeled processes without using the BPMN standard. Torres and Pelechano [69] generates full web applications from BPMN models, which support both automated and human activities. The method does not allow for integration with hand-written code. Furthermore, there exists a variety of approaches that *derive GUIs* from BPMN models [4, 9, 20] or user iterface flow models [83]. However, these approaches either do not consider the application logic, persistence, or communication aspects of process-aware information systems.

8 Discussion: Limitations and Strengths

We have shown the practicability of our approach using some real-life examples in a demo-application. The practicability of the generator framework MontiGem without the BPMN additions was already shown in the full-size real-world project MaCoCo [28]. Thus, we discuss the the extension of the MontiGem framework in terms of its limitations, strengths and usability in an already existing application.

8.1 Limitations of the approach

The limitations result from using DSLs and a generative approach as well as from the requirements of the generated PAIS.

Technology stack. The use of many DSLs can lead to interoperability, language-version and language-migration problems (also called DSL-Babel challenge) [26]. A common technology stack for the different DSLs reduces this problem. Therefore, we use the MontiCore language workbench and the MontiGem generator framework for the definition of the DSLs.

Concepts in the grammar. Thus, a threat to validity is the current size of the grammar which includes 89,5% of the analytic and 86,8% of the executable BPMN elements. Our experiences have indicated, that a smaller set of concepts might already be sufficient for PAIS generation. However, further investigations are needed to find out if a domain-specific version of the language with a smaller scope or even a simple process modeling DSL will be sufficient.

Our system requirements do not include an automatic data flow check during run-time and we do not support message exchange between different organizations in our BPMN models (and thus the PAIS is not supporting this). The developed BPMN notation is limited to internal processes, i.e., processes with a single pool. It is possible to extend the textual notation to processes with multiple pools, so-called collaborations that show the exchange of messages between the different parties. This results from the main interest in generating PAIS for business processes within the confines of a single organization. However, modeling the message exchange would make it necessary to generate the associated communication infrastructure with external parties. Thus, a language for collaborations would have to be added.

Grammar structure. An improvable aspect of the current solution is the structure of the BPMN language itself, which is defined in one large-scale grammar. Clearly, a modular structure of the DSL with several component grammars would increase the reuseability and allow for extensions [22] and several domain-specific variants of the language [8]. The division in multiple smaller parts would facilitate the exchange of parts such as how data objects are defined or the use of another constraint language. Other (parts of) process languages such as activity diagrams could also be considered to extend the use of existing tooling.

Use of language concepts in the generation process. Until now, behavior models are used in MontiGem only to create PAISs. However, the textual BPMN DSL could be used for several other purposes, e.g., for automated regression testing [46] or in combination with an interpreter instead of a generator [45]. Clearly, also assistive systems which use human behavior information gathered from sensor data to support the users [48], would profit from the proposed approach. If BPMN is the optimal solution or other behavior modeling languages might have a better fit needs to be investigated.

Generative Approach vs. Interpretation. Changes in business processes require new generation and deployment of the system which makes our approach less flexible than systems that only interpret BPMN models at run-time. The data structure and GUI models resulting from the generation using the BPMN models could be written and generated separately, and used by the process engine during run-time [18]. This would avoid a need for regeneration, as the necessary environment would be already generated. This still requires BPMN models to be transformed to the general BPMN format before they can be interpreted directly. However, as our formerly existing application needs the generation step anyway for changes of the data structure, a generation need for changing BPMN models is not a deterioration of the current development process.

Textual vs. Graphical Notation. The generation process takes BPMN models in the presented textual notation as input. As BPMN provides an XML-based exchange format which most BPMN editors, like e.g., Camunda, provide, and since the tooling of our tex-

tual BPMN provides a transformation from this XML-format to the textual notation, it is also possible to provide the BPMN models in the graphical notation. Vice versa, e.g., Camunda, also allows to import BPMN models in the XML exchange format. Layouting these models remains a manual task. Since our language tooling also provides an automatic transformation from the textual BPMN notation to the XML exchange format, displaying and editing the textual models, e.g., in Camunda is possible.

Automation. As for most generative approaches, the business logic, such as algorithms to calculate combinations of input data in certain forms, is not repetitive and, thus, might always need additional hand-written code. This means that the application can be released in a fully automated way but cannot be fully used with the BPMN extension. However, since this is also the case for the other of the parts of the generated application, this is not an impediment.

8.2 Strengths

The strengths of our approach are its scalability, adaptability and the common language infrastructure.

Scalability. The technical scalability is given as several models can be used in parallel, both in the system generation process as well as during run-time, and we allow for recursive process calls. Using the BPMN standard, there is no limitation to a particular domain, which means domain scalability is given. As we transform our models to the standardized BPMN exchange format, the modular system allows for exchanging the process engine with any BPMN-compliant engine.

Adaptability of the application. Our approach explicitly allows for the integration with hand-written code and supports repeated generation and agile, iterative engineering processes [29]. Together with a high degree of test, build and release automation, also changes in the process model can be realized fast and delivered in a short period of time, which is crucial for real-world applications.

Common language and tooling infrastructure. Furthermore, the language workbench MontiCore enables the combined use of heterogeneous languages to describe orthogonal system aspects in the most appropriate language, e.g., BPMN for business processes and UML CDs and OCL for data. As these languages have the whole infrastructure in common (AST, symbol table and CoCos) and allows for imports and reuse of models in other languages (resolving) it is easy to use a combination of multiple languages. Translating BPMN models to a petri net representation enabled to reuse existing model checkers for implementing the CoCo checks. As petri nets may not be as intuitive to the developers, using a petri net language directly would lower the modeling efficiency. By using a model-to-model transformation, we do not have to sacrifice the intuitiveness of BPMN while still being able to reuse existing and well-probed tools for verifying well-formedness.

Process and form consistency. Within the MaCoCo project that handles more than 160 instances of the application, we already generate elements for the UI and input forms. This allows us to keep the look and feel the same for the user and avoids styling deviations between different forms. The same applies to the generated processes, as they are used for systematical generated code.

8.3 Challenges for the use in real-world projects

Within this paper, we have shown the application of our approach in green-field, which means that no prior application exists. Clearly, this is easier than in a brown field approach, where the generated application needs to be aligned with an already existing application and existing functionalities.

Adding new functionalities vs. replacing functionalities. Using our approach for the uplifting towards process-awareness of an already generated application, which was generated using the same generator base would not be a challenge for additional functionality, as this additional generation step does not effect already existing models or pages in the application. All BPMN, Tagging-, data- and GUI-models as well as the pages in the GUI are an addition to existing ones. Relevant changes may only be necessary for the main navigation. As this is generated as well, the replacement of the navigation is not a big issue.

More challenging would be the replacement of existing functionalities such as input forms where the user enters specific data. In these cases, already generated forms in the data-centric application will need to be replaced: In Figure 7, some GUI models (5), which were generated by the BPMN generator (4), will replace some of the hand-written GUI models (7) to generate the new forms. If hand-written additions (8) already exist, they might have to be adapted to extend the newly generated pages and fit to the structure.

Version changes during the development process. The development of this approach together with a new DSL took several months. Within this time, the MontiGem generator project evolved into a new project structure, e.g., separated projects for the runtime environment and specific application data. Additionally, also the language versions evolved. Internal changes in the generator can affect the generated code and

therefore a synchronization between the projects and the languages might be necessary.

Variations in processes for different database instances in MaCoCo. Considering different organization sizes (small/medium/large chairs with app. 5/30/150 staff members) require different GUIs and organizational hierarchies within a data-centric application. Regarding processes, it might be relevant to allow different kinds of processes, different process granularity or not to use specific processes at all. With our generative approach, it is possible to generate these different versions of forms and annotate the GUI models with specific details which settings lead to showing one or another version in the navigation of the GUI. The annotation has to be added by hand in the current version.

Architecture. The current architecture requires two databases (see Figure 6), one for the domain data (2a) and one for the execution data of the processes (2b). Currently, the synchronization between these two databases is handled by parts of the generated application back-end to ensure data consistency. This could be improved if the databases could reference each other.

9 Conclusion

To sum up, we have introduced an approach that allows for the generative development of enterprise information systems with an integrated process engine. For this, we have developed a novel textual BPMN notation that is suited for code generation and we have shown how to use textual BPMN models in the generation process of process-aware information systems.

Our approach includes two different engineering phases, namely (1) language engineering including further evolvement of DSLs and (2) application engineering including generator engineering, which are loosely coupled. Thus, the phases could easily be fulfilled by separate teams which indicates the suitability of the approach for larger projects. In our case, both phases were performed by one team.

Current business applications require both, a focus on storing and representing data as well as the ability to handle processes within the organization. This shift from data-centric to process-aware information systems leads to the requirement to include process modeling languages within MDSE approaches. The developed prototype and the strengths of our approach (see section 8) has provided us with the necessary information to consider an application of this approach in full-size real-world applications, e.g., the MaCoCo project[28]. Additional application areas include assistive services within generated information systems [48], to generate process-aware digital twin cockpits [6], or the addition of assistive services for the human-in-the-loop within digital twins [15].

This paper constitutes a promising step towards aligning business and IT. The textual BPMN notation enables business users and developers to jointly model and reason about business processes. Moreover, MDSE provides the technical backbone to generate running applications from the process models. Business users and developers can validate their assumptions in a real application and adapt the process models or the underlying business processes if necessary. The result is a collaborative and highly iterative development process.

Compliance with ethical standards

Conflict of Interest: The authors declare that they have no conflict of interest.

References

1. van der Aalst, W.M.P.: Process-Aware Information Systems: Lessons to Be Learned from Process Mining, pp. 1–26. Springer Berlin Heidelberg (2009)
2. Adam, K., Michael, J., Netz, L., Rumpe, B., Varga, S.: 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), *LNI*, vol. P-304, pp. 59–66. Gesellschaft für Informatik e.V. (2020)
3. Adam, K., Netz, L., Varga, S., Michael, J., Rumpe, B., Heuser, P., Letmathe, P.: Model-Based Generation of Enterprise Information Systems. In: M. Fellmann, K. Sandkuhl (eds.) Enterprise Modeling and Information Systems Architectures (EMISA'18), *CEUR Workshop Proceedings*, vol. 2097, pp. 75–79. CEUR-WS.org (2018)
4. Alfonso Hoyos, J.P., Restrepo-Calle, F.: Automatic Source Code Generation for Web-Based Process-Oriented Information Systems. In: Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE 17) (2017)
5. Arkin, A.: Business Process Modeling Language. http://xml.coverpages.org/BPML-2002.pdf (2002)
6. Bano, D., Michael, J., Rumpe, B., Varga, S., Weske, M.: Process-Aware Digital Twin Cockpit Synthesis from Event Logs. Journal of Computer Languages (COLA) **70** (2022). DOI 10.1016/j.cola.2022.101121
7. Brambilla, M., Butti, S., Fraternali, P.: WebRatio BPM: A tool for designing and deploying business processes on the web. In: Int. Conf. on Web Engineering (2010)
8. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Systematic Composition of Independent Language Features. Journal of Systems and Software **152**, 50–69 (2019). DOI 10.1016/j.jss.2019.02.026
9. Calegari, D., Delgado, A.: Model-Driven Generation of a BPMS Portal Based on Interaction Flow Modeling Language Models. In: Int. WS on Interplay of

Model-Driven and Component-Based Software Engineering (ModComp17), pp. 31–37 (2017)
10. Ceri, S., Brambilla, M., Fraternali, P.: The History of WebML Lessons Learned from 10 Years of Model-Driven Development of Web-Applications. Conceptual Modeling: Foundations and Applications **5600** (2009)
11. Chaâbane, A., Turki, S.H., Charfi, A., Bouaziz, R.: From Platform Independent Service Composition Model in BPMN4SOA to Executable Service Compositions. In: Conf. on Information Integration and Web-Based Applications & Services (2010)
12. Chinosi, M., Trombetta, A.: BPMN: An introduction to the standard. Comput. Stand. Interfaces **34**(1) (2012)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
14. Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., Varga, S., Wortmann, A.: Generating Customized Low-Code Development Platforms for Digital Twins. Journal of Computer Languages (COLA) **70** (2022). DOI 10.1016/j.cola.2022.101117
15. Dalibor, M., Michael, J., Rumpe, B., Varga, S., Wortmann, A.: Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In: G. Dobbie, U. Frank, G. Kappel, S.W. Liddle, H.C. Mayr (eds.) Conceptual Modeling, pp. 377–387. Springer International Publishing (2020). DOI 10.1007/978-3-030-62522-1_28
16. Delgado, A., Ruiz, F., de Guzmán, I.G.R., Piattini, M.: MINERVA: Model drIveN and sErvice oRiented Framework for the Continuous Business Process improVement and relAted Tools. In: Conf. on Service-Oriented Comp. (ICSOC'09) (2009)
17. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
18. Drave, I., Gerasimov, A., Michael, J., Netz, L., Rumpe, B., Varga, S.: A Methodology for Retrofitting Generative Aspects in Existing Applications. Journal of Object Technology **20**, 1–24 (2021). DOI 10.5381/jot.2021.20.2.a7
19. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Introduction. In: Process-Aware Information Systems, chap. 1, pp. 1–20. John Wiley & Sons, Ltd (2005)
20. Díaz, E., Panach, J.I., Rueda, S., Distante, D.: A family of experiments to generate graphical user interfaces from bpmn models with stereotypes. Journal of Systems and Software **173**, 110883 (2021). DOI 10.1016/j.jss.2020.110883
21. Eder, J., Groiss, H., Liebhart, W.: The Workflow Management System Panta Rhei, pp. 129–144. Springer Berlin Heidelberg (1998). DOI 10.1007/978-3-642-58908-9_7
22. Erdweg, S., Rieger, F.: A Framework for Extensible Languages. In: Proc. of the 12th International Conference on Generative Programming: Concepts and Experiences, GPCE '13, p. 3–12. ACM (2013). DOI 10.1145/2517208.2517210
23. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous Soundness Checking of Industrial Business Process Models. In: 7th International Conference on Business Process Management, BPM 2009, pp. 278–293. Springer-Verlag, Ulm, Germany (2009). DOI 10.1007/978-3-642-03848-8\%00819
24. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models. Data Knowl. Eng. **70**(5), 448–466 (2011). DOI 10.1016/j.datak.2011.01.004
25. Fazziki, A.E., Lakhrissi, H., Yetognon, K., Sadgal, M.: A Service Oriented Information System: A Model Driven Approach. In: Int. Conf. on Signal Image Technology and Internet Based Systems (SITIS '12) (2012)
26. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering (FOSE '07) pp. 37–54 (2007)
27. Freund, N.: Development of a Text-Based Representation of BPMN Models. Master's thesis, Leibniz Universität Hannover, Hannover, Germany (2018)
28. Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., Rumpe, B., Varga, S.: Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers, pp. 22–30. CEUR Workshop Proceedings (2020)
29. Gerasimov, A., Michael, J., Netz, L., Rumpe, B., Varga, S.: Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In: B. Anderson, J. Thatcher, R. Meservy (eds.) 25th Americas Conference on Information Systems (AMCIS 2020), AIS Electronic Library (AISeL), pp. 1–10. Association for Information Systems (AIS) (2020)
30. Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., Navarro Perez, A., Plotnikov, D., Reiß, D., Roth, A., Rumpe, B., Schindler, M., Wortmann, A.: Integration of Handwritten and Generated Object-Oriented Code. In: Model-Driven Engineering and Software Development, *Communications in Computer and Information Science*, vol. 580, pp. 112–132. Springer (2015)
31. Greifenberg, T., Look, M., Roidl, S., Rumpe, B.: Engineering Tagging Languages for DSLs. In: Conference on Model Driven Engineering Languages and Systems (MODELS'15), pp. 34–43. ACM/IEEE (2015)
32. Groefsema, H., Bucur, D.: A Survey of Formal Business Process Verification: From Soundness to Variability. In: 3rd International Symposium on Business Modeling and Software Design, BMSD 2013 (2013)
33. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Composition of Heterogeneous Modeling Languages. In: Model-Driven Engineering and Software Development, *Communications in Computer and Information Science*, vol. 580, pp. 45–66. Springer (2015)
34. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In: Model-Driven Engineering and Software Development Conference (MODELSWARD'15), pp. 19–31. SciTePress (2015)
35. Heim, R., Mir Seyed Nazari, P., Rumpe, B., Wortmann, A.: Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In: Conference on Modelling Foundations and Applications (ECMFA), LNCS 9764, pp. 67–82. Springer (2016)
36. Hemel, Z., Verhaaf, R., Visser, E.: WebWorkFlow: An object-oriented workflow modeling language for web applications. In: Model Driven Engineering Languages and Systems (Models '08), pp. 113–127. Springer, Toulouse, France (2008)
37. Hölldobler, K., Kautz, O., Rumpe, B.: MontiCore Language Workbench and Library Handbook: Edition

2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag (2021). URL http://www.monticore.de/handbook.pdf
38. Höver, K.M., Borgert, S., Mühlhäuser, M.: A domain specific language for describing S-BPM processes. In: S-BPM ONE - Running Processes. Springer (2013)
39. Ivanchikj, A., Serbout, S., Pautasso, C.: From text to visual bpmn process models: Design and evaluation. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (2020)
40. Kim, G.W., Lee, J.H., Son, J.H.: Classification and analyses of business process anomalies. In: Conf. on Comm. Software and Networks (ICCSN'09). IEEE (2009)
41. Koehler, J., Vanhatalo, J.: Process anti-patterns: How to avoid the common traps of business process modeling. IBM WebSphere Developer Technical Journal **10** (2007)
42. Kühne, S., Kern, H., Gruhn, V., Laue, R.: Business process modelling with continuous validation. In: Business Process Management Workshops (2009)
43. Liu, R., Kumar, A.: An analysis and taxonomy of unstructured workflows. In: Conf. on Business Process Management, *LNCS*, vol. 3649. Springer (2005)
44. Loja, L.F.B., Neto, V.V.G., da Costa, S.L., de Oliveira, J.L.: A business process metamodel for Enterprise Information Systems automatic generation. In: Anais Do I Congresso Brasileiro de Software: Teoria e Prática-i Workshop Brasileiro de Desenvolvimento de Software Dirigido Por Modelos, vol. 8 (2010)
45. López-Pintado, O., Dumas, M., García-Bañuelos, L., Weber, I.: Interpreted Execution of Business Process Models on Blockchain. In: IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC), pp. 206–215 (2019)
46. Makki, M., Van Landuyt, D., Joosen, W.: Automated Regression Testing of BPMN 2.0 Processes: A Capture and Replay Framework for Continuous Delivery. In: Proc. of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, p. 178–189. ACM (2016). DOI 10.1145/2993236.2993257
47. Michael, J., Netz, L., Rumpe, B., Varga, S.: Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In: N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer, A. Wortmann (eds.) Proceedings of MODELS 2019. Workshop MDE4IoT, pp. 595–614. CEUR Workshop Proceedings (2019)
48. Michael, J., Rumpe, B., Varga, S.: Human behavior, goals and model-driven software engineering for assistive systems. In: A. Koschmider, J. Michael, B. Thalheim (eds.) Enterprise Modeling and Information Systems Architectures (EMSIA 2020), vol. 2628, pp. 11–18. CEUR Workshop Proceedings (2020)
49. Michels, S., Plasmeijer, R., Achten, P.: iTask as a New Paradigm for Building GUI Applications. In: J. Hage, M.T. Morazán (eds.) Implementation and Application of Functional Languages, pp. 153–168. Springer Berlin Heidelberg (2011)
50. Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines. In: Modellierung 2022, LNI. GI (2022)
51. Mogos, A.H., Urzica, A.: TN4PM: A textual notation for process modelling. In: G.A. Papadopoulos, C. Badica (eds.) Intelligent Distributed Computing III (2009)
52. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77** (1989)
53. Nalepa, G.J., Kluza, K., Ciaputa, U.: Proposal of automation of the collaborative modeling and evaluation of business processes using a semantic wiki. In: 17th Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2012) (2012)
54. Noll, J., Scacchi, W.: Specifying process-oriented hypertext for organizational computing. J. Netw. Comput. Appl. **24**(1), 39–61 (2001)
55. OASIS: Web services business process execution language version 2.0. Specification (2017)
56. Object Management Group: Omg unified modeling language, v2.5.1 (2017)
57. OMG: Business Process Model and Notation (BPMN), Version 2.0.2. Tech. rep., Object Management Group (2013)
58. OMG: OMG Unified Modeling Language (OMG UML), Version 2.5.1. Specification, Object Management Group (2017)
59. Onoda, S., Ikkai, Y., Kobayashi, T., Komoda, N.: Definition of deadlock patterns for business processes workflow models. In: 32nd Hawaii Int. Conf. on System Sciences (HICSS-32) (1999)
60. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall (1981)
61. Plasmeijer, R., Achten, P., Koopman, P., Lijnse, B., van Noort, T.: An iTask Case Study: A Conference Management System, pp. 306–329. Springer Berlin Heidelberg (2009). DOI 10.1007/978-3-642-04652-0_7
62. Polyvyanyy, A., van der Werf, J.M.E.M., Overbeek, S., Brouwers, R.: Information Systems Modeling: Language, Verification, and Tool Support. In: P. Giorgini, B. Weber (eds.) Advanced Information Systems Engineering, pp. 194–212. Springer International Publishing (2019)
63. Reisig, W.: Petri Nets: An Introduction. Springer-Verlag Berlin Heidelberg (1985)
64. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International (2016). URL http://www.se-rwth.de/mbse/
65. Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International (2017). URL http://www.se-rwth.de/mbse/
66. Rychkova, I., Le Grand, B., Souveyet, C.: Towards Executable Specifications for Case Management Processes. Springer International Publishing (2017)
67. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
68. Stair, R., Reynolds, G.: Principles of Information Systems. Cengage Learning (2020)
69. Torres, V., Pelechano, V.: Building business process driven web applications. In: Int. Conf. on Business Process Management (2006)
70. Urzica, A., Tanase, C., Florea, A.M.: Bridging the gap between business experts and software agents: BPMN to AUML transformation. UPB Scientific Bulletin, Series C: Electrical Engineering **72** (2010)
71. van der Aalst, W.M.P.: Verification of workflow nets. In: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, ICATPN '97, pp. 407–426. Springer-Verlag, Berlin, Heidelberg (1997)
72. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers **8** (1998)
73. van der Aalst, W.M.P., de Crom, P.J.N., Goverde, R.R.H.M.J., van Hee, K.M., Hofman, W.J., Reijers, H.A., van der Toorn, R.A.: ExSpect 6.4 An Executable Specification Tool for Hierarchical Colored Petri Nets. In:

M. Nielsen, D. Simpson (eds.) Application and Theory of Petri Nets 2000, pp. 455–464. Springer Berlin Heidelberg (2000)
74. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business Process Management: A Survey. In: Business Process Management (2003)
75. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Formal Aspects of Computing Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing **23** (2011)
76. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: Classification, decidability, and analysis. Form. Asp. Comput. **23** (2011)
77. van Eck, M.L., Lu, X., Leemans, S.J.J., van der Aalst, W.M.P.: PM^2 : A process mining project methodology. In: Advanced Information Systems Engineering, pp. 297–313. Springer International Publishing (2015)
78. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of web services composition languages: The case of bpel4ws. In: I.Y. Song, S.W. Liddle, T.W. Ling, P. Scheuermann (eds.) Conceptual Modeling - ER 2003, pp. 200–215. Springer Berlin Heidelberg (2003)
79. Wolf, K.: Petri net model checking with LoLA 2. In: Int. Conf. on Application and Theory of Petri Nets and Concurrency (2018)
80. Workflow Management Coalition (WfMC): Process definition interface - XML process definition language. Specification WFMC-TC-1025, The Workflow Management Coalition (2005)
81. Wynn, M.T., Verbeek, H.M., van der Aalst, W.M., ter Hofstede, A.H., Edmond, D.: Business process verification - Finally a reality! Business Process Management Journal **15**(1) (2009)
82. Yang, H., Shi, W.: Research on modeling and transformation method of web service composition based on petri net. In: 2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP) (2021)
83. Yongchareon, S., Liu, C., Zhao, X., Yu, J., Ngamakeur, K., Xu, J.: Deriving user interface flow models for artifact-centric business processes. Computers in Industry **96**, 66–85 (2018). DOI 10.1016/j.compind.2017.11.001