# Delta-BPMN: a Concrete Language and Verifier for Data-aware BPMN

Silvio Ghilardi[1], Alessandro Gianola[2], Marco Montali[2], and Andrey Rivkin[2]

[1] Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy
silvio.ghilardi@unimi.it
[2] Free University of Bozen-Bolzano, Bolzano, Italy
{gianola,montali,rivkin}@inf.unibz.it

**Abstract.** The increasing recognition of the need for integrating data and processes, both at conceptual and system levels, raises a new demand in standard-friendly, verifiable data-aware process modelling languages. So far, a few proposals in the area have been largely focusing on either uncharted approaches or conceptual proposals that would lack in tool support. In this work, we propose delta-BPMN – a verifiable operational framework for data-aware processes that employs (block-structured) BPMN to capture the process backbone, and a SQL-based language for representing and manipulating volatile and persistent data. We also propose a proof-of-concept implementation of delta-BPMN by realising the front-end part in Camunda and the back-end in a framework that translates language specifications into the executable code of a state-of-the-art SMT-based model checker.

**Keywords:** Data-aware processes · BPMN · Model checking

## 1 Introduction

The integration between data and processes is a long-standing challenge in information systems engineering [14,19,21]. This comes with a number of difficulties. On the one hand, the model should be expressive enough to represent complex processes where data influence how the process control-flow routes cases, while the process tasks inspect and manipulate data. On the other hand, such expressiveness has to be suitably controlled towards enabling verification, execution, monitoring, and mining of such multi-perspective models. A third, orthogonal dimension concerns the choice of modeling constructs, which often depart from those offered by process and data modeling standards such as BPMN and SQL, in turn hampering the adoption of the resulting frameworks.

These three dimensions can be recognized in their full complexity when it comes to the *verification* of the resulting integrated models [5,10]. Verification is of particular importance in this spectrum, as even data and process models that appear correct when analyzed in isolation may lead to errors once integrated [18]. "Verifiability" of models is thus typically obtained by using abstract languages that do not adhere to well-established standards when it comes to the data and/or process component: either the control-flow backbone of the process is captured using Petri nets or other mathematical formalisms for dynamic systems that cannot be directly understood using front-end notations such as BPMN, or the data manipulation part relies on abstract, logical operations that cannot be straightforwardly represented in concrete data manipulation languages such as SQL. At the same time, the repertoire of constructs used to model data-aware processes cannot cover these languages in their full generality, as verification becomes immediately undecidable if they are not suitably controlled [5]. A last crucial issue is that the vast majority of the literature in this spectrum mainly provides foundational results that do not directly translate into effective verification tools.

In this work, we tackle these limitations and propose delta-BPMN, an operational framework at once supporting modeling and verification of BPMN enriched with data management capabilities. delta-BPMN comes with a threefold contribution. First, we introduce the front-end data modeling and manipulation language PDMML, supported by delta-BPMN, which instantiates the data-related aspects of the abstract modeling language studied in [1] by using a SQL-based dialect to represent as well as manipulate volatile and persistent data, and show how it can be embedded into a (block-structured) fragment of BPMN that captures the process backbone. The features of PDMML are based on requirements for concrete, verifiable data-aware process modeling languages distilled from the literature. Second, we show how the delta-BPMN front-end can be realized in actual business process management systems, considering in particular Camunda[3], one of the most popular BPMN environments. Third, we report on the implementation of a translator that, building on the encoding rules abstractly defined in [1], takes a delta-BPMN model created in Camunda and transforms it into the syntax of MCMT[4], a state-of-the-art SMT-based model checker for infinite-state systems that can then be used for verification.

## 2 Requirement Analysis and Related Work

The integration of data and processes is a long-standing line of research at the intersection of BPM, data management, process mining, and formal methods. Since our focus is on verification, we circumscribe the relevant works to those dealing with the formal analysis of data-aware processes. As pointed out in the introduction, this is also crucial because the choice of language constructs is affected by the task one needs to solve - in particular, verifying such sophisticated models requires to suitably control the data and control-flow components as well as their interaction [5,10].

A second important point is that the vast majority of the contributions in this line of research provide foundational results, but do not come with corresponding operational tools for verification. Hence, all in all, *we consider in this research only those approaches for the integration of data and processes that come with verification tool support*: VERIFAS [16], BAUML [11], ISML [18], dapSL [4], and the delta-BPMN approach considered here, which relies on the DAB formal model [1] as its foundational basis.

We use these approaches to distill a series of important requirements on languages for verifiable data-aware processes, indicating which provide full ($+$), partial ($+/-$), or no support ($-$) for that requirement. The first two requirements concern verifiability, respectively capturing foundational and practical aspects.

[3] `https://camunda.com`

[4] `http://users.mat.unimi.it/users/ghilardi/mcmt/`

Table 1: Requirements coverage (covered $+$, partially $(+/-)$, not $-$)

| Framework | **RQ 1** | **RQ 2** | **RQ 3** | **RQ 4** | **RQ 5** | **RQ6** | **verification logic** |
|---|---|---|---|---|---|---|---|
| VERIFAS [16] | $+$ | $+$ | $-$ | $+$ | $+$ | y | fragment of LTL-FO |
| BAUML [11] | $+$ | $+/-$ | $+$ | $+$ | $+$ | n | fixed test cases |
| ISML [18] | $+/-$ | $-$ | $+/-$ | $+$ | $+/-$ | n | state-space exploration |
| dapSL [4] | $+/-$ | $-$ | $+/-$ | $+$ | $+/-$ | n | state-space exploration |
| **delta-BPMN** | $+$ | $+$ | $+/-$ | $+$ | $+$ | y | data-aware safety |

**RQ 1.** The language should be operationally verifiable with a tool. ◁

While the approaches above all come with an operational counterpart for verification, there are huge differences in how this support is provided. VERIFAS comes with an embedded, ad-hoc verification tool $(+)$ that supports the model checking of properties expressed in a *fragment of first-order LTL*. BAUML encodes verification into a form of first-order satisfiability checking over the flow of time $(+)$, defining a fixed set of *test cases* expressing properties to be checked as derived predicates. ISML relies on state-space construction techniques for Colored Petri nets, but in doing so it assumes that the data domains are all bounded $(+/-)$; no specific verification language is defined, leaving to the user the decision on how to explore the state space. dapSL relies instead on an ad-hoc state-space construction that, under suitable restrictions, is guaranteed to faithfully represent in a finite-state way the infinite state space induced by the data-aware process; however, no additional techniques are defined to explore the state space or check temporal properties of interest $(+/-)$. Finally, delta-BPMN encodes verification of *(data-aware) safety properties* (expressed in the language defined in [1]) into the state-of-the-art MCMT model checker $(+)$.

The second requirement concerns the analysis of key properties (such as soundness, completeness, and termination) of the algorithmic techniques used for verification. This is crucial since, in general, verifying data-aware processes is highly undecidable [5,10].

**RQ 2.** The verification techniques come with an analysis of key properties such as soundness, completeness, termination. ◁

Since ISML and dapSL do not come with specific algorithmic techniques for verification, no such analysis is provided there $(-)$. BAUML relies on first-order satisfiability techniques that come with semi-decidability guarantees. In [11], it is claimed that for a certain class of state-bounded artifact systems, verification terminates; however, this is not guaranteed, as for that class only decidability of verification is known, not that the specifically employed satisfiability algorithm terminates $(+/-)$. VERIFAS comes with a deep, foundational study on the boundaries of decidability of verification [9]; the study identifies classes of data-aware processes for which finite-state abstractions can be constructed, guaranteeing termination of the verifier when analyzing such classes $(+)$. Finally, delta-BPMN relies on the foundational DAB framework [1], where soundness, completeness, termination of the algorithmic technique implemented in MCMT are extensively studied $(+)$.

The third crucial requirement is about the type of language adopted, and whether it adheres to accepted standards or is instead rather ad-hoc.

**RQ 3.** The language relies on well-assessed standards for processes and data. $\triangleleft$

Recall that, to carry out verification, the features supported by the language need to be carefully controlled. So we do not assess approaches based on their coverage of constructs, but rather focus on which notations they employ. On the one hand, approaches like VERIFAS adopt a language inspired by artifact-centric models but defined in an abstract, mathematical syntax ($-$). At the other end of the spectrum, BAUML comes with a combination of UML/OCL-based models to specify the various process components ($+$). In between we find the other proposals ($+/-$): ISML relies on Petri nets and employs data definition and manipulation languages defined in an ad-hoc way; dapSL instead defines the control-flow implicitly via condition-action rules, and uses a language grounded in the SQL standard for querying and updating the data. delta-BPMN relies on a combination of (block-structured) BPMN and SQL for data manipulation; while standard SQL is employed for data queries and updates, the language has to be extended with some ad-hoc constructs when it comes to actions and (user) inputs ($+/-$).

In data-aware processes, it is essential to capture the fact that while the process is executed, new data can be acquired.

**RQ 4.** The language supports the injection of data into the process by the external environment. $\triangleleft$

All of the listed approaches agree on the need of equipping the language with mechanisms to inject data from the external environment. VERIFAS and BAUML allow one to nondeterministically assign values from value domains to (special) variables, ISML extends this functionality with an ability to guarantee that assigned values are globally fresh (but then it works by assuming a fixed finite domain for such fresh input), whereas dapSL supports all such functionalities using a language of service calls. In delta-BPMN we adopt a data injection approach similar to the one used in VERIFAS.

When executing process cases, one typically distinguishes at least two types of data: volatile data attached to the case itself, and persistent data that may be accessed and updated by different cases at once. This leads to our last requirement.

**RQ 5.** The language distinguishes volatile and persistent data elements. $\triangleleft$

While BAUML, VERIFAS, and DAB natively provide distinct notions for case variables and underlying persistent data ($+$), ISML models conceptually account for token data and separate facts, but such facts are not stored in a persistent storage ($+/-$), while dapSL models all data as tuples of a relational database ($-$).

At last, a very important aspect that puts the approaches into two distinct groups, is whether persistent data are managed under a unique access policy, or instead there is a fine-grained distinction based on how the process can access them. This impacts the type of verification conducted, as discussed below. Since supporting or not read-only data simply separates the different approaches, but does not correspond to a qualitative difference, we simply put *'yes'* (y) when it is supported and *'no'* (n) when it is not.

**RQ 6.** The language separates read-only persistent data from persistent data that are updated during the execution. $\triangleleft$

This is an important distinction because it heavily affects the type of verification that must be considered [5,10]. On the one hand, approaches like BAUML, dapSL, and ISML that do not distinguish read-only from updatable persistent data (n) require to fully fix their initial configuration, and provide verification verdicts by considering all possible evolutions of the process starting from this initial configuration. Contrariwise, approaches like VERIFAS and delta-BPMN that do this distinction (y) in turn focus on forms of parameterized verification where the properties of interest are studied for every possible configuration of the read-only data, certifying whether the process correctly works regardless of which specific read-only data are picked.

Table 1 summarizes the different requirements and support provided by the analyzed literature. We take this as a basis to compare the delta-BPMN language and verification infrastructure with the other existing approaches. For completeness, we also indicate in the table which verification properties are considered in each approach.

It is also worth noting that there is a plethora of other approaches falling into the artifact-/data-/object-centric spectrum. For example, Guard-Stage-Milestone (GSM) language [8], the object-aware business process management framework of PHILharmonic Flows [15], the declarative data-centric process language RESEDA based on term rewriting systems [20]. In a nutshell, these approaches combine data and processes dimensions, but largely focus on modeling, with few exceptions offering runtime verification of specific properties (e.g., RESEDA allows for a specific form of liveness checking) supported by a tool.

Other relevant works investigate the integration of data and processes with a system engineering approach [17,12,7] tailored to modeling and enactment. Of particular relevance is ADEPT [7], which is similar in spirit to delta-BPMN, as it allows to combine a block-structured process modeling language with SQL statements to interact with an underlying relational storage, with the goal of providing execution and analytic services. The main difference with delta-BPMN is that our PDMML language focuses on conservative extensions of (block-structured) BPMN and SQL to obtain a verifiable, integrated model.

## 3 The PDMML Language

To realize the modeling requirements introduced in Section 2, we start from the approach in [1]. The main issue there is that while the process backbone relies on (block-structured) BPMN, the definition and manipulation of data is done with an abstract, mathematical language that does not come with a concrete, user-oriented syntax.

To define a delta-BPMN model, we then revisit the data component of the process, introducing a Process Data Modeling and Manipulation Language (PDMML) . We do so in two steps: first, we start from BPMN and isolate the main data abstractions that must be represented in our framework, introducing suitable *data definition* operations in PDMML; second, we start from the abstract, logical language studied in [1], and introduce a concrete counterpart for *data manipulation* statements in PDMML, using SQL as main inspiration. In this way, we achieve compliance with **RQ 3**. We then integrate PDMML language for data inspection and manipulation within BPMN blocks, so as to comply with **RQ 3** for both the data and the control-flow aspects.

Notice that, deliberately, PDMML does not come with explicit mechanisms to refer to other process instances from a given instance. This is due to technical reasons related to verification, which will be highlighted in Section 4.2.

## 3.1 Sources of Data and their Definition

While BPMN does not introduce any specific language to manipulate and query data, it introduces two main abstractions to account for them: *data objects*, representing *volatile data* manipulated by each case in isolation; and *persistent stores*, representing persistent units of information that are accessed and updated possibly by multiple cases.

**Persistent data.** To account for **RQ 6**, PDMML allows to define two types of persistent storages with different access policies. More specifically, we use a so-called *repository* $\mathcal{R}$ to store data that can be both queried and updated, and a *catalog store* $\mathcal{C}$ with a read access only. The declaration of these two stores is done with a set of statements, each accounting for a relation schema (or *table*) therein. Each table comes with *typed attributes* defining the *names* of the table columns with the respective *(value) types*.

An attribute is declared in PDMML as $\mathsf{A}:\mathbf{T}$, where $\mathsf{A}$ is an attribute name and $\mathbf{T}$ is its type. Each type is of one of the following three different forms: *(i)* a *primitive*, system-reserved *type* (such as strings and integers); *(ii)* a dedicated *id type* $\mathbf{T}_R$ accounting for the identifiers of table $R$ (like ISBNs for the $Book$ table - if they are used as primary key to identify books); *(iii)* a *data type* accounting for a semantic domain (like person names or addresses). For every catalog table, say, with name $R$, PDMML also requires to define an attribute with name id and a distinguished id type $\mathbf{T}_R$, so as to account for the primary key of that table in an unambiguous way.

Based on these notions, a *catalog* is a set of *catalog tables*, each defined with a statement of the form $R(\mathsf{id}:\mathbf{T}_R, \mathsf{A_1}:\mathbf{T_1}, \ldots, \mathsf{A_n}:\mathbf{T_n})$, where: *(i)* $R$ is the *table name*; *(ii)* $\mathsf{id}:\mathbf{T}_R$ is the explicit table identifier of $R$ with a dedicated (identifier) type $\mathbf{T}_R$; *(iii)* $n+1$ is the *table arity*; *(iv)* for every $i \in \{1, \ldots, n\}$, $\mathbf{T}_i$ is a primitive type, an identifier type of some relation in the catalog or a data type. Each catalog table is equipped with a *table id attribute* of the form $\mathsf{id}:\mathbf{T}_R$, always assumed to appear in the first position. According to the definition, the other attributes may have, as type, the identifier type of another catalog table. This mechanism is used to define, in a compact way, the presence of a *foreign key* dependency relating two catalog tables.

Similarly to the case of a catalog, a *repository* is a set of *repository tables*, each defined with a similar statement to that of catalog tables, with the only difference that now there is no explicit table identifier. This means that, while repository tables can reference catalog tables, they cannot reference other repository tables, and thus behave like *free relations*. Conceptually, this is not a limitation, since the idea behind the use of the repository is not to support a full-fledged database (as it is done for the catalog), but to provide a working memory where data taken from the catalog, case variables and external sources are accumulated and manipulated. This approach to model the repository is in line with foundational frameworks studied before [16,1,3]. In addition, it enjoys the key properties of the most sophisticated scenarios known in the literature to guarantee verifiability [9,3,1] – hence we have to stick with it in the light of **RQ 1**.

As customary, when defining tables, PDMML requires that each table name is used only once overall (at the catalog *and* repository level). Hence we can use the table name to unambiguously refer to the table as a whole. To disambiguate attributes from different tables, we sometimes use a dot notation, where $R.A$ indicates attribute $A$ within table $R$. In addition, table aliases can be used within queries towards expressing self-joins.[5]

**Volatile data.** For modeling volatile, case data in a way that makes them compatible with persistent data, we use typed variables whose declaration signature is similar to the one of attributes. Specifically, a *case variable* with name $v$ and type $T$ is then simply defined in PDMML as a statement $\#v:\mathbf{T}$. The definition of the volatile data of a process then just consists of a set of case variable statements.

The collective set of declarations for case variables, catalog relations, and repository relations is called *data model*.

**Example 1.** Consider a mortgage approval process followed by the Customer Service Representatives (CSR) department of a bank.[6] To manage information about available mortgage types, customers' bank accounts, submitted applications, status of their records and possible mortgage approval results, the process relies on multiple sources of data.

Each mortgage application is created by a CSR employee and can be managed throughout the process execution by using process variables. At the same time, certain data values have to be moved from volatile case variables to a persistent repository, and vice-versa. In this process, for example, we use variables $\#cid : \mathbf{CID}$, $\#bid : \mathbf{BaID}$, $\#bankAmount : \mathbf{Num}$ to store information about a customer as well as their eligible bank account, and variables $\#tid : \mathbf{MTID}$, $\#duration : \mathbf{Num}$, $\#amount : \mathbf{Num}$ to collect data for the mortgage contract.

The information static to the process (i.e., it shall never be updated) is stored in the CSR's read-only database. For example, table $BankAccount(\mathsf{BAid} : \mathbf{BankAccountID}, \mathsf{CBA} : \mathbf{CID}, \mathsf{Deposit} : \mathbf{int}, \mathsf{StatusBank} : \mathbf{String})$ contains information about possibly multiple bank accounts owned by the customers together with the account status information retained in $\mathsf{StatusBank} : \mathbf{String}$), whereas $MortgageType(\mathsf{Mid} : \mathbf{MTID}, \mathsf{Name} : \mathbf{String}, \mathsf{Amount} : \mathbf{Num}, \mathsf{Duration} : \mathbf{Num}, \mathsf{Interest} : \mathbf{Num})$ contains details regarding various mortgage offers, including information on mortgage duration and the amount of interests to be paid. $\triangleleft$

## 3.2 The Process Component of delta-BPMN

The control-flow backbone of a delta-BPMN process relies on the recursive composition of block-structured BPMN patterns that adhere to the standard BPMN 2.0 syntax. We focus on block-structured BPMN since this allows us to define a direct execution semantics also for advanced constructs like interrupting exceptions and cancelations, and to exploit this upon verifying the resulting models (see [1] for the technical details). However, our approach would seamlessly carry over to the case where the control-flow backbone of the process is captured using a Petri net, as in [13].

[5] This latter feature is currently not supported by the implementation, but it will be supported soon. The page `https://tinyurl.com/y6npo4kz` provides a continuously updated list of the most recent, newly added features.

[6] The example builds on a model from Business Process Incubator (see `https://tinyurl.com/8au7xfmw`) enriched with data by analogy with a similar model from the benchmark in [16].

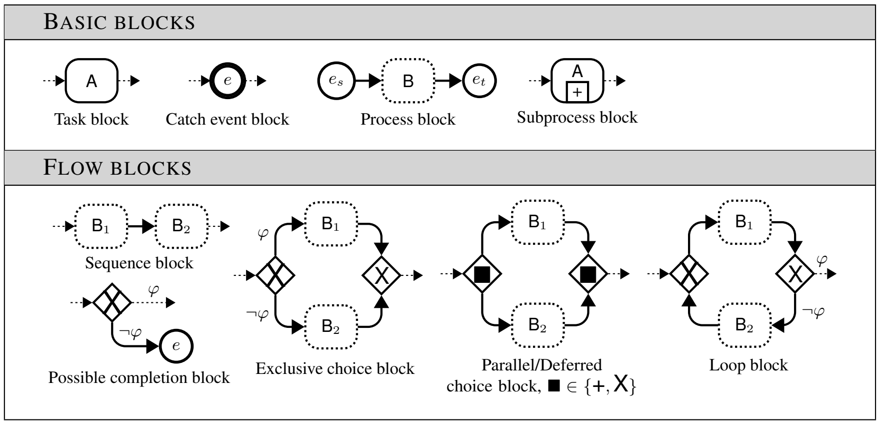


Fig. 1: Supported BPMN blocks

Although, conceptually, delta-BPMN supports the same set of blocks as DAB [1], its current implementation covers the fundamental blocks shown in Figure 1. As usual, blocks are classified into leaf blocks (in our case, tasks and events) and non-leaf blocks that combine sub-blocks in a specific control-flow structure.

Implicitly, each block has a lifecycle. Initially, the block is inactive and its state is `idle`. When a process instance, throughout its execution, reaches an `idle` block, it becomes `enabled`. This means that the `enabled` element may be then nondeterministically executed depending on the choice of the process executor(s). When the process instance has completed traversing the block, the block lifecycle state changes from `enabled` to `compl`. The `compl` element then advances the progression of the process instance following what is dictated by the parent block. In the exact same moment, the block changes its state back to `idle`. The execution rules used for regulating the evolution of each block depending on its type faithfully reconstruct what prescribed by the BPMN standard. Consider, for example, a deferred choice block S with two sub-blocks B$_1$ and B$_2$. Its lifecycle starts in state `enabled`, that can be nondeterministically progressed to state `active`. This progression simultaneously forces the change of state of B$_1$ and B$_2$ from `idle` to `enabled`. As soon as one of the two sub-blocks, say, B$_1$ is selected, it moves to `active` whereas its sibling block B$_2$ goes back to `idle`. As soon as B$_1$ finishes by reaching state `compl`, it switches to `idle` and triggers a simultaneous transition of the parent block S from `active` to `compl`. Following this logic, one can analogously and exhaustively define the lifecycle model for each type of block.

**Example 2.** Figure 2 shows the control-flow backbone of the mortgage approval process (Example 1), represented in delta-BPMN by following the same block decomposition.$\lhd$

The main, open question is how data enter into the definition of blocks. Following the BPMN standard, this is handled in two distinct points: leaf blocks (capturing tasks and events), and (data-driven) choices. Such blocks are annotated with suitable PDMML statements to capture data inspection and manipulation. This is handled next.

### 3.3 Inspecting and Manipulating Data with PDMML

To express how a task/event inspects and manipulates data, we decorate it with three distinct PDMML expressions, respectively defining: *(i) newly declared variables*, to account for external data input; *(ii)* a *precondition*, providing possible bindings for the input variables of the task/event considering the catalog, the repository, as well as the case and newly defined variables; *(iii)* an *effect* that, once a binding for the newly declared variables and for the input variables is picked, determines how the task/event manipulates the case variables and the repository.

An obvious choice to inspect relational data as those present in our catalog and repository is to resort to relational query languages such as SQL. This choice would be in line with **RQ 3**. However, our setting requires to consider two crucial aspects. On the one hand, it is important to coherently employ a single query language to account for different querying needs, such as expressing the precondition of a task or the conditions determining which route to take in a choice. On the other hand, differently from pure SQL, our queries have to consider the presence of case variables, addressing the possibility of simultaneously working over persistent and volatile data, as well as the possibility of injecting data from the external environment. For example, think of a job category that has been chosen by an applicant during the application process (and thus suitably stored in a dedicated case variable) and for which the process should provide all open positions. In this case one would need to use the job category value in the `WHERE` clause of a dedicated `SELECT` query accessing the catalog that already contains information about all the positions for the previously selected category. At the same time, one might also want to query only the current state of the case variables, or to ask the user to provide their credit card number when paying a fee.

**Newly declared variables.** The ability of injecting a data object of type $T$ form the external environment (cf. **RQ 4**) is handled through a *newly declared variable* with the following PDMML statement $decl ::= (\texttt{var}\ v : \mathbf{T})^*$, where $v$ is the name of the newly declared variable. Upon execution, $v$ is bound to *any* value from $T$. When attached to a task, newly declared variables can be seen as an abstract representation of a user form or a web service result. When attached to an event, they represent the event payload.

**Preconditions.** Preconditions indicate under which circumstances a task can be executed or an event triggered. They also retrieve data from the catalog, repository, case variables and newly defined variables attached to the same leaf block. To account for these different aspects, PDMML incorporates a hybrid SQL-based query language that can retrieve volatile and persistent data at once. Consistently with the execution semantics given in [1] that is, in turn, in line with the customary "variable binding" abstraction employed in formalisms such as Colored Petri nets, the typical usage of queries in our framework is to return a set of answers from which one is (nondeterministically) picked to induce a progression step within the process. Notice that this way of managing query results is customary in the artifact-centric literature [5,9,16,3].

To define preconditions, we first need to introduce PDMML conditions, defined as:

$$cond ::= x_1 \odot x_2 \ \mid\ cond_1\ \texttt{AND}\ cond_2 \ \mid\ cond_1\ \texttt{OR}\ cond_2$$

Essentially, a PDMML condition is a boolean expression (with negation pushed inwards) over atomic conditions of the form $x_1 \odot x_2$, where $x_1$ and $x_2$ are expression terms (whose specific shape is determined by the context in which the condition is used), and $\odot \in \{=, \neq, >, <, \leq, \geq\}$) is a comparison operator. In atomic conditions, we assume component-wise type compatibility of terms (e.g., the two operands in $x_1 \odot x_2$ must have the same type). Notice that, as customary, the atomic condition `TRUE` (capturing the condition that always succeeds) can be defined as an abbreviation (similarly for `FALSE`).

Using conditions as atomic building blocks, a PDMML precondition is defined as:

$$\begin{aligned}
pre &::= cond \mid query \\
query &::= \texttt{SELECT}\ \mathsf{A}_1, \ldots, \mathsf{A}_s\ \texttt{FROM}\ R_1, \ldots, R_m\ \texttt{WHERE}\ filter \\
filter &::= cond \mid \texttt{TUPLE}\ (\vec{x})\ \texttt{IN}\ R \mid \texttt{TUPLE}\ (\vec{x})\ \texttt{NOT IN}\ R \\
&\quad \mid filter_1\ \texttt{AND}\ filter_2 \mid filter_1\ \texttt{OR}\ filter_2
\end{aligned}$$

Here, each $R_i$ from the SQL-like *query* can be a repository or a catalog relation, whereas $R$ from *filter* can only be a catalog relation. This is in line with theoretical results reported in [3,1]. Terms in *cond* of *pre* can be case variables, constants, or newly defined variables declared in the same leaf block. Instead, terms used in *cond* of $filter$ coincide with those from above, but can also use attributes that appear in the **FROM** statement of the contingent *query* expression (i.e., $\mathsf{A}_1, \ldots, \mathsf{A}_s$). When writing queries, notation $R.A$ can be used to more explicitly refer to attribute $A$ of table $R$.

**Example 3.** In the mortgage approval process scenario touched in Examples 1 and 4, the following query can be used to list bank accounts of the customers who have completed the mortgage application procedure:

$$\begin{aligned}
&\texttt{SELECT}\ \mathsf{BAid},\ \mathsf{CBA},\ \mathsf{StatusBank}\ \texttt{FROM}\ BankAccount \\
&\texttt{WHERE}\ \mathsf{CBA} = \#cid\ \texttt{AND}\ \#status = \texttt{CompletedApplication}
\end{aligned}$$

Here, $\#status : \mathbf{String}$ indicates the current status of the process.

**Effects.** Task/event effects consist of data manipulation PDMML statements operating over case variables and repository tables. In the following, we use term *input variable* to refer to newly defined variables or attributes of the precondition attached to the same leaf block of the effect under scrutiny.

Each case variable $\#v$ can be *updated* using a trivial assignment statement $\#v = u$, where $u$ is either a constant or an input variable. It is assumed that, for each case variable, at most one case variable assignment statement can be written within one update.

One can also model insertion and deletion of tuples into the persistent storage. Since the catalog is read-only, these updates can be performed only on the repository relations.

An *insertion* (statement) on some repository relation $R$ is defined as **INSERT** $v_1, \ldots, v_n$ **INTO** $R$, where each $v_i$ is either a constant, a case variable or an input variable. This `INSERT` statement is similar to the corresponding classical DML (data manipulation language) statement in SQL. However, we deliberately avoid using the `VALUES` clause since we insert one tuple at a time, and so we can rely on the more compact notation where the elements to be inserted are directly indicated close to $R$.

A *deletion* (statement) is defined as **DELETE** $v_1, \ldots, v_n$ **FROM** $R$. Here, similarly to the insertion, each $v_i$ is either a constant, a (case) variable, or an input variable, whose type coincides with the type of the $i$-th attribute in $R$.

We also allow to perform *conditional updates*. For that, we employ a modified SQL `CASE` statement directly embedded into the update logic. This statement logically resembles an *if-then-else* expression with multiple *else-if* branches, and in which each condition in the *if*-part is a query. To ensure verifiability [1,3] (cf. **RQ 1**), it is necessary for the statement to obey to one limitation: it cannot access any other repository table beyond the one that is being updated. The conditional update statement has the form:

```
UPDATE $R$ SET $R$.$a_1$=@$v_1$, ..., $R$.$a_m$=@$v_m$ WHERE
CASE WHEN $F_1$ THEN @$v_1$=$u_1^1$, ..., @$v_m$=$u_m^1$
       ...
     WHEN $F_k$ THEN @$v_1$=$u_1^k$, ..., @$v_m$=$u_m^k$
     ELSE @$v_1$=$u_1^e$, ..., @$v_m$=$u_m^e$
```

This statement is the most sophisticated one in the offered language as it requires the modeler to take care of the following two aspects. First, similarly to the SQL's `UPDATE` statement, which can modify multiple tuples in a table, ours performs a (conditional) *bulk edit* of elements in *each* tuple of $R$, and the `SET` clause specifies (using names of the attributes of $R$ with the $R$'s name in the prefix)[7] what are exactly those elements. The `SET` clause also uses placeholder variables @$v_j$ that support the conditional update logic: whenever a tuple in $R$ satisfies one of the $F_i$ filters, the corresponding `THEN` clause will assign concrete values $u_j^i$ to all the placeholder variables mentioned in `SET`. Second, the modeler has to carefully control the variables and attributes used both in the `WHEN` and `THEN` clauses. As we have already mentioned above, each $F_i$ cannot access repository relations but $R$ itself. At the same time, it can reuse elements from the precondition query such as variables and attributes. This, in turn, allows to use $F_i$ for filtering results returned by the precondition query, and thus allowing to carefully select the data that are going to be used in the final update of every single tuple of $R$. As for the elements appearing in `THEN` clauses, their values can be constants as well as elements taken from results returned by the precondition query. In the following we provide a few examples demonstrating correct and illegal update statements.

**Example 4.** Continuing with our running example, we now give the example of a legal conditional update handling the assessment of the eligibility of a mortgage application. To manage key information about the applications submitted for the mortgage approval, the bank employs a repository that consists of one relation schema:

$$\mathit{Info}(\mathsf{Bank}:\mathbf{BaID}, \mathsf{StatusB}:\mathbf{String}, \mathsf{Reliability}:\mathbf{String}) \quad \triangleleft$$

Here, for each application, CSR performs an assessment procedure, during which all customer's bank accounts are checked for reliability. All the accounts with histories that did not include any fraudulent charges, are then marked accordingly in relation *Info*. Technically, we formalize this situation with a conditional update of the form:

```
UPDATE Info SET Info.Reliability=@v WHERE
CASE WHEN Info.StatusB!=fraud THEN @v=Yes
     ELSE @v= No
```

[7] This disambiguates the situation where the same relation $R$ is used in the update precondition with some of its attributes both appearing in the `SELECT` and some of the `WHEN` clauses.

Note that the *when-then-else* clause allows us to perform a bulk update over the repository relation $Info$ by changing the reliability status of its entries.

Consider the repository relation $Rejected(\mathsf{Bank}:\mathbf{BaID})$, storing bank accounts that have been already rejected before in the process by another department. The following update statement, that additionally checks if the bank account has already been rejected, is illegal, since the condition of the first case involves the repo-relation $Rejected$:

```
UPDATE Info SET Info.Reliability=@v WHERE
CASE WHEN Info.StatusB!=fraud AND TUPLE (Info.Bank) NOT IN Rejected
THEN @v=Yes ELSE @v=No
```

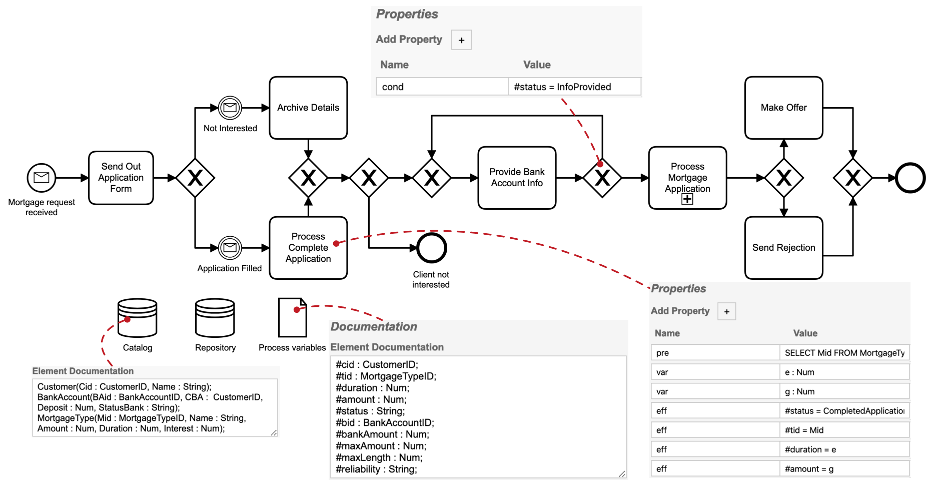
The overall execution semantics of leaf blocks is defined as follows. Once the leaf block is `enabled`, a binding for its newly defined attributes can be provided. If, under this binding, the precondition of the leaf block evaluates to true for at least one binding of its attributes, then the leaf block may nondeterministically fire, depending on the choice of the process executors. Upon firing, the binding of precondition attributes and of newly defined variables provide a grounding the for effect attached to the leaf block. Once the effect has been performed, the block completes its execution, and the state of its lifecycle becomes `compl`, as described above. The only additional requirement is that, in the case of a task having both a precondition and an effect, we assume that the task is atomic at the level of data updates. This is not for a technical reason, but for a conceptual one: it is essential to ensure that insertions/deletions/updates are applied on the same data snapshot that was used for checking the task precondition, in accordance with the standard *transactional semantics* of relational updates. Breaking simultaneity would lead to race conditions with other update specifications potentially operating over the same case variables or repository tables. Notice that race conditions can still occur at the level of the process, when parallel blocks and sequences of tasks/events are employed. Consequently, requiring atomicity for leaf blocks with preconditions and effects does not lead to a loss of generality.

### 3.4 Guards for Conditional Flows

The last place where PDMML statements are needed is in the context of blocks employing choice splits as a way to conditionally route process instances. Specifically, each conditional flow is linked to a PDMML condition whose terms are case variables or constants. Notice that using only case variables is not a limitation, since, as we have seen before, case variables can be filled with data extracted from the catalog/repository, or injected from the external environment.

As shown in Figure 1, we assume that each choice split foresees two outputs with complementary guards. This means that the user has to specify only one guard $\varphi$, while the other guard (indicated as $\neg\varphi$ in the figure) is automatically constructed via syntactic manipulation of $\varphi$ as follows: De Morgan laws are applied until negation appears just in front of atomic conditions, and then the negated atomic conditions are replaced by their corresponding, complementary conditions (e.g., $\leq$ is substituted by $>$).

We have now completed the definition of PDMML. In the next section we show how PDMML is practically realised in delta-BPMN.



| Name | Value |
|---|---|
| cond | #status = InfoProvided |

Element Documentation

```
Customer(Cid : CustomerID, Name : String);
BankAccount(BAid : BankAccountID, CBA :  CustomerID,
Deposit : Num, StatusBank : String);
MortgageType(Mid : MortgageTypeID, Name : String,
Amount : Num, Duration : Num, Interest : Num);
```

Element Documentation

```
#cid : CustomerID;
#tid : MortgageTypeID;
#duration : Num;
#amount : Num;
#status : String;
#bid : BankAccountID;
#bankAmount : Num;
#maxAmount : Num;
#maxLength : Num;
#reliability : String;
```

| Name | Value |
|---|---|
| pre | SELECT Mid FROM MortgageTy |
| var | e : Num |
| var | g : Num |
| eff | #status = CompletedApplicatio |
| eff | #tid = Mid |
| eff | #duration = e |
| eff | #amount = g |

Fig. 2: A delta-BPMN model with a few examples of Camunda-based annotations (taken as screenshots from the tool)

## 4 delta-BPMN in Action

We now put delta-BPMN in action, considering both modeling and verification.

### 4.1 Modeling delta-BPMN Processes with Camunda

We discuss how Camunda, one of the most widely employed (open-source) platforms for process modeling and automation, can be directly adapted to model delta-BPMN processes. We in particular employ the Camunda Modeler environment (`camunda.com`) to create the process control-flow, and its extension part to incorporate PDMML statements. At this stage, it is not essential to recognize the process blocks (and check whether the process control-flow is block-structured): we just annotate the overall process model with the data definitions, the tasks/events with the corresponding PDMML preconditions and effects, and the choice branches with PDMML boolean queries.

An alternative possibility would have been to require the modeler to explicitly insert data object and data store icons in the process model, and annotate those. However, this would clutter the visual representation of the process, creating unreadable diagrams.

More specifically, to declare repository (resp., catalog) relations we use a dedicated persistent store symbol called `Repository` (resp., `Catalog`). The declarations themselves, separated by the semicolon from one another, are put into the documentation box of the element's documentation. For example, Figure 2 demonstrates a snapshot of a catalog declaration containing definitions of two relations $Customer$ and $MortgageType$ from Example 1. We deal similarly with case variables: a single data object called `Process variables` is used, whose documentation box contains all case variable declarations with the semicolon being used as a separator (cf. Figure 2).

Modelling queries as well as other data manipulation expressions in traditional BPMN 2.0 could be done using annotations. This could be considered as a more traditional approach that, however, as we have already discussed above, can lead to difficulties in managing the processes diagram. Instead, we propose to handle such expressions declaratively within the Camunda extension elements. Given that properties in Camunda are represented as key-value pairs, adding a declaration is rather easy: one needs to use a special data manipulation expression identifier as the key and the actual expression as the value. Consistently with Section 3, we use the following reserved identifiers: *(i)* `cond` – a gateway/flow condition identifier; *(ii)* `pre` – a precondition identifier; *(iii)* `var` – a new typed variable declaration identifier; *(iv)* `eff` – an update statement identifier.

Each key is meant to be used only with values of a particular type. Like that, `cond` and `pre` identify queries, whereas `var` and `eff` respectively denote new variable declarations and update statements. All the BPMN elements that admit the aforementioned extensions can have multiple `var` and `eff` identifiers. This is useful as there can be more than one new typed variable declaration as well as multiple case variable assignment statements.

**Example 5.** Task Process Complete Application in Figure 2 selects a mortgage type in case a customer has agreed to apply for it. This is done by adding a `pre`-identified property to extension elements of the task with the following query that nondeterministically selects one mortgage type from the $MortgageType$ relation:

**SELECT** Mid **FROM** $MortgageType$ **WHERE** $\#status =$ FillApp **AND** $e > 0$ **AND** $g > 0$

As an effect, this task is supposed to move a chosen mortgage type ID to a dedicated case variable, and decide on the amount of money asked as well as the interest to be paid in case the mortgage offer gets accepted. The latter is done with two newly declared variables $e$ and $g$, and three `eff`-identified properties with the following case variable assignments: $\#tid =$ Mid, $\#duration = e$ and $\#amount = g$. Note that the last two essentially model a user input and thus realize the data injection mentioned in **RQ 4**. $\lhd$

All the queries identified with `cond` can be used only in blocks containing choice splits (i.e., blocks from Figure 1 with $\varphi$ annotations on the arcs). In Figure 2, we show a screenshot of a simple condition assigned to one of the XOR gateways of the loop block.

## 4.2 Encoding delta-BPMN Camunda Processes in MCMT

To make delta-BPMN processes modeled in Camunda verifiable (cf. **RQ 1**) we have implemented a translator that takes as input a `.bpmn` file produced by Camunda following the modeling guidelines of the previous section, and transforms it into the syntax of a state-of-the-art model checker that can verify data-aware processes parametrically to the read-only relations, namely the latest version of MCMT [1,3,2].

The translation first checks whether the input model is block-structured, isolating the various blocks. This is done through traversal algorithm that is of independent interest. Each block is then separately converted into a corresponding set of MCMT instructions by implementing, rule by rule, the encoding mechanism proposed in [1]. This works since the concrete PDMML syntax introduced here for data definition and manipulation faithfully mirrors the abstract, logical language employed there.

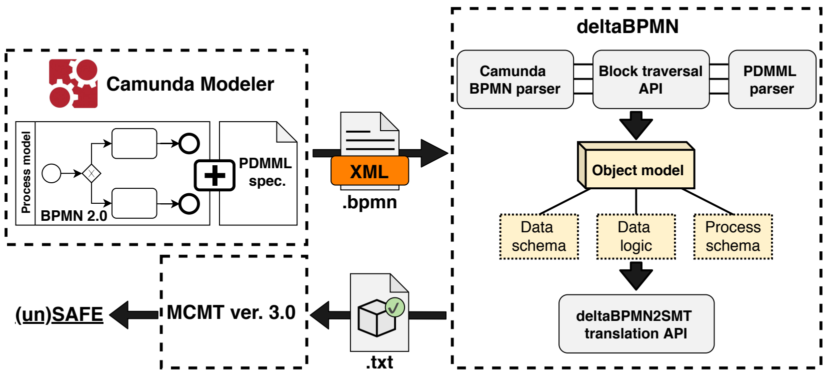


Fig. 3: Conceptual architecture of the delta-BPMN framework

For verification, we obviously need also to express which properties we want to check. Every property is defined as a condition that specifies a "bad", undesired state of the model. To add a property, we employ the same mechanism as above that uses Camunda extension elements. More specifically, we add another reserved identifier `verify` which can be used to add property key-value pairs directly to the process. For example, one can write the PDMML condition (*#status*=`Archived` **`AND`** *lifecycleMortgage*=`Completed`) to verify the safety of the model in Figure 2, in particular ascertaining whether the mortgage approval process has been finalized with the customer not being interested in the related offer (see the related End event Client not interested in Figure 2), thus resulting in her application being archived. Notice that here we use a special variable *lifecycleMortgage* to access the process lifecycle state. In general, one may query the process lifecyle by using a special internal variable *lifecycleModelName*, where *ModelName* is the actual process model name. Verification of lifecycle properties for single blocks can be tackled by introducing dedicated case variables, manipulating them in effects according to the lifecycle evolution of the block.

It is important to mention that, although this feature is not explicitly reflected in the PDMML language, delta-BPMN provides support for modeling and verification of multi-instance scenarios in which process instances can access and manipulate the same catalog and repository. Formal details are given in [1]. In summary, [1] indicates that unboundedly many simultaneously active process instances can be verified for safety if they do not explicitly refer to each other (i.e., they do not expose their own case identifiers to other instances). Explicit mutual references can instead be handled if the maximum number of simultaneously active process instances is known a-priori.

Figure 3 shows the overall toolchain employed for verification. First, a modeler has to produce a delta-BPMN process by enriching a regular block-structured BPMN 2.0 process with a PDMML specification via Camunda extensions using the technique from above. Camunda Modeler then allows to export the delta-BPMN process as an XML-formatted `.bpmn` file. This file can be then processed by our Java-based tool, called *deltaBPMN*, that employs the following APIs for generating the process specification that can be readily verified by MCMT (`http://users.mat.unimi.it/users/ghilardi/mcmt/`). In the nutshell, the tool takes two major steps to process the delta-BPMN model. First, it uses the Camunda's BPMN model API to access process components from the input `.bpmn` file and uses our block traversal API as well as PDMML parser to recognize blocks as well as PDMML statements/declarations and consecutively generate delta-BPMN objects. The latter are specified according to the object model that has been mainly distilled from the formalism studied in [1] and that consists of three major parts: a data schema storing all case variable and relation declarations (from both $\mathcal{R}$ and $\mathcal{C}$), a process schema storing nested supported process block definitions, and a data logic containing update declarations and conditions assigned to blocks. The block traversal API uses a newly developed algorithm for detecting nested blocks that comply to the object model structure. Via the *deltaBPMN2SMT* translation API that internally follows the translation in [1], the tool then processes the extracted object model and generates a text file containing the delta-BPMN process specification rewritten in the MCMT syntax.

Finally, the derived specification can be directly checked in the MCMT tool that, in turn, will detect whether the specification is safe or unsafe with respect to the *"bad"* property specified in the initial model. MCMT can be executed in the command line using the following command: **`[time] mcmt <filename>`**. Here, argument **`[time]`** is not mandatory, but can be used if one wants to display the MCMT execution time. More information on the model checker installation process, the language for specifying safety properties of delta-BPMN models, advanced execution options and additional details, together with the actual delta-BPMN implementation, can be found on the tool website here: `https://tinyurl.com/y6npo4kz`.

## 5 Conclusions

We have introduced a SQL-based language for modeling and manipulating volatile and persistent data, and demonstrated how it can be incorporated into the existing BPMN standard, resulting in a language for modeling data-aware BPMN that we called delta-BPMN. We showed how this delta-BPMN processes can be modeled with Camunda using its native extension capabilities. We also reported on an implementation of a prototype that takes delta-BPMN models produced in Camunda and automatically translates them into the syntax of MCMT that, in turn, allows for their immediate verification. Given that Camunda also allows to extend its user interface with additional third-party functionalities, we intend to develop a fully integrated environment for modelling and verification of delta-BPMN processes. We also plan to investigate in more detail usability aspects of our proposal and set up a concrete benchmark that could be then fully adopted (including process- and data-specific metrics) within the RePRoSitory platform [6].

**Acknowledgments.** This research has been partially supported by the UNIBZ projects *REKAP*, *VERBA* and *DUB*. We would like to thank Davide Cremonini for contributing to the initial phase of the delta-BPMN implementation.

## References

1. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Formal modeling and SMT-based parameterized verification of data-aware BPMN. In: Proc. of BPM. pp. 157–175.

LNCS, Springer (2019)

2. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: From model completeness to verification of data aware processes. In: Description Logic, Theory Combination, and All That. LNCS, vol. 11560, pp. 212–239. Springer (2019)
3. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: SMT-based verification of data-aware processes: a model-theoretic approach. Mathematical Structures in Computer Science **30**(3), 271–313 (2020)
4. Calvanese, D., Montali, M., Patrizi, F., Rivkin, A.: Modeling and in-database management of relational, data-aware processes. In: Proc. of CAiSE. pp. 328–345. LNCS, Springer (2019)
5. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: A database theory perspective. In: Proc. of PODS (2013)
6. Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F.: Reprository: a repository platform for sharing business process models. In: Proc. of BPM (PhD/Demos). CEUR Workshop Proceedings, vol. 2420, pp. 149–153. CEUR-WS.org (2019)
7. Dadam, P., Reichert, M., Rinderle-Ma, S., Lanz, A., Pryss, R., Predeschly, M., Kolb, J., Ly, L.T., Jurisch, M., Kreher, U., Göser, K.: From ADEPT to aristaflow BPM suite: A research vision has become reality. In: BPM Workshops. LNCS, vol. 43, pp. 529–531. Springer (2009)
8. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. In: Proc. of BPM (2011)
9. Deutsch, A., Li, Y., Vianu, V.: Verification of hierarchical artifact systems. In: Proc. of PODS. pp. 179–194. ACM (2016)
10. Deutsch, A., Hull, R., Li, Y., Vianu, V.: Automatic verification of database-centric systems. ACM SIGLOG News **5**(2), 37–56 (2018)
11. Estañol, M., Sancho, M., Teniente, E.: Ensuring the semantic correctness of a BAUML artifact-centric BPM. Inf. Softw. Technol. **93**, 147–162 (2018)
12. Fahland, D., Meyer, A., Pufahl, L., Batoulis, K., Weske, M.: Automating data exchange in process choreographies (extended abstract). In: Proc. of EMISA 2016. CEUR Workshop Proceedings, vol. 1701, pp. 13–16. CEUR-WS.org (2016)
13. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Petri nets with parameterised data - modelling and verification. In: Proc. of BPM. pp. 55–74. LNCS, Springer (2020)
14. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: Proc. of OTM. LNCS, vol. 5332. Springer (2008)
15. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: Fundamental requirements and their support in existing approaches. Int. J. Inf. Syst. Model. Des. **2**(2) (2011)
16. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: A practical verifier for artifact systems. PVLDB **11**(3) (2017)
17. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Proc. of BPM. pp. 171–186. LNCS, Springer (2013)
18. Polyvyanyy, A., van der Werf, J.M.E.M., Overbeek, S., Brouwers, R.: Information systems modeling: Language, verification, and tool support. In: Proc. of CAiSE. pp. 194–212. LNCS, Springer (2019)
19. Reichert, M.: Process and data: Two sides of the same coin? In: Proc. of OTM. LNCS, vol. 7565. Springer (2012)
20. Seco, J.C., Debois, S., Hildebrandt, T.T., Slaats, T.: RESEDA: declaring live event-driven computations as reactive semi-structured data. In: Proc. of EDOC. pp. 75–84 (2018)
21. Steinau, S., Marrella, A., Andrews, K., Leotta, F., Mecella, M., Reichert, M.: DALEC: a framework for the systematic evaluation of data-centric approaches to process management software. Softw. Syst. Model. **18**(4), 2679–2716 (2019)