Modeling and In-Database Management of Relational, Data-Aware Processes

Diego Calvanese[1], Marco Montali[1], Fabio Patrizi[2], and Andrey Rivkin[1](✉)

[1] Free University of Bozen-Bolzano, Bolzano, Italy
{calvanese,montali,rivkin}@inf.unibz.it
[2] Sapienza Università di Roma, Roma, Italy
patrizi@dis.uniroma1.it

Abstract. It is known that the engineering of information systems usually requires a huge effort in integrating master data and business processes. Existing approaches, both from academia and the industry, typically come with ad-hoc abstractions to represent and interact with the data component. This has two disadvantages: *(i)* an existing database (DB) cannot be effortlessly enriched with dynamics; *(ii)* such approaches generally do not allow for integrated modelling, verification, and enactment. We attack these two challenges by proposing a declarative approach, fully grounded in SQL, that supports the agile modelling of relational data-aware processes directly on top of relational DBs. We show how this approach can be automatically translated into a concrete procedural SQL dialect, executable directly inside any relational DB engine. The translation exploits an in-database representation of process states that, in turn, is used to handle, at once, process enactment with or without logging of the executed instances, as well as process verification. The approach has been implemented in a working prototype.

Keywords: Data-aware processes · Process engines · Relational databases

1 Introduction

During the last two decades, increasing attention has been given to the challenging problem of resolving the dichotomy between business process and master data management [3,10,18]. Devising integrated models and corresponding enactment platforms for processes and data is now acknowledged as a key milestone, which cannot be reduced to implementation-level solutions at the level of enterprise IT infrastructures [8].

This triggered a flourishing line of research on concrete languages for data-aware processes, and on the development of tools to model and enact such processes. The main unifying theme for such approaches is a shift from standard activity-centric business process meta-models, to a paradigm that focuses

© Springer Nature Switzerland AG 2019
P. Giorgini and B. Weber (Eds.): CAiSE 2019, LNCS 11483, pp. 328–345, 2019.
https://doi.org/10.1007/978-3-030-21290-2_21

first on the elicitation of business entities (and data), and then on their behavioral aspects. Notable approaches in this line are artifact-centric [10], object-centric [12] and data-centric models [19]. In parallel to these new modeling paradigms, also BPMS based on standard, activity-centric approaches a là BPMN, have increased the tool support of data-related aspects. Many modern BPM platforms provide (typically proprietary) data models, ad-hoc user interfaces to indicate how process tasks induce data updates, and query languages to express decisions based on data. While this approach has the main advantage of hiding the complexity of the underlying relational database (DB) from the modeler, it comes with two critical shortcomings. First, it makes it difficult to conceptually understand the overall process in terms of general, tool-agnostic principles, and to redeploy the same process in a different BPMS. This is witnessed by a number of ongoing proposals that explicitly bring forward complex mappings for model-to-model transformation (see, e.g., [11,23]). Second, this approach cannot be readily applied in the recurrent case where the process needs to be integrated with existing DB tables. In fact, updating an underlying DB through a more abstract data model is challenging and relates to the long-standing, well-known *view update problem* in the database literature [9].

We approach these issues by addressing three main research questions in a setting in which *the database can be directly accessed and updated by running processes*:

RQ1 Is it possible to ground data-centric approaches based on condition-action rules in the SQL standard, also accounting for the interaction with external systems?

RQ2 Is it possible to develop a corresponding enactment engine running directly on top of legacy DBMSs, and providing logging functionalities?

RQ3 Is it possible to implement the foundational techniques for the (abstract) state-space construction of data-centric approaches based on condition-action rules?

To answer **RQ1**, we propose a declarative language, called $\mathbb{dapSL}$, that introduces minimal changes to the SQL standard, allowing one to: *(i)* encapsulate process tasks into SQL-based, parameterized actions that update the DB, possibly injecting values obtained from external inputs (e.g., ERP systems, user forms, web services, external applications), and *(ii)* define rules determining which actions are executable and with which parameter bindings, based on the answers obtained by querying the DB. Notably, the last feature is intrinsic to many artifact- and data-centric approaches [1,2,6,13] since the data manipulation in them is handled via rules. In practice, $\mathbb{dapSL}$ can be used either in a bottom-up manner, as a scripting language that enriches DBs with processes, or in a top-down manner, as a way to complement standard, control flow-oriented process modelling languages with an unambiguous, runnable specification of conditions and tasks. From the formal point of view, $\mathbb{dapSL}$ represents the concrete counterpart of one of the most sophisticated formal models for data-aware processes [1], which comes with a series of (theoretical) results on the conditions

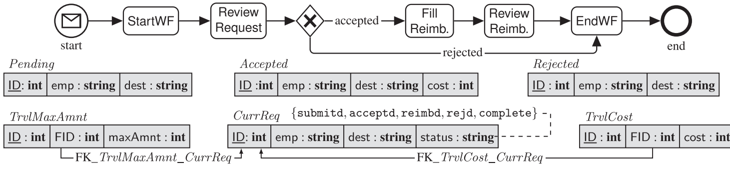

Fig. 1. The travel management process and a corresponding data model

under which verification can be carried out. In fact, a wide range of foundational results tackling the formalization of data-ware processes, and the identification of boundaries for their verifiability has been obtained [3], but the resulting approaches never resulted in actual modeling and enactment tools. In this sense, dapSL constitutes the first attempt to bridge the gap between such formal approaches and concrete modeling and execution.

To address **RQ2**, we propose a framework, called DAPHNE, where data-aware processes are directly specified on top of standard relational DBs. To support such specifications, an automatic translation of dapSL into a concrete procedural SQL dialect is presented, in turn providing direct in-database process execution support. The framework has been implemented within the DAPHNE engine, whose back-end consists of a relational storage with corresponding stored procedures to manage the action-induced updates, and whose JAVA front-end provides APIs and functionalities to inspect the current state of the process and its underlying data, as well as to interact with different concrete systems for acquiring external data.

Thanks to a sophisticated encoding of the dapSL process state and related data into SQL, DAPHNE seamlessly accounts for three key usage modalities: enactment with and without logging of historical data, and state space construction for formal analysis. The third modality is fully addressed within the scope of **RQ3**. Relying on DAPHNE, we propose a solution that constructs the state space invoking the same mechanism used for enactment. This ensures that the analysis is carried out on the exact same model that is enacted, differently from usual approaches in formal verification, where an abstract version of a concrete model is constructed in ad-hoc way for verification purposes.

DAPHNE is available at https://bit.ly/2KIbMvN.

2 Data-Aware Process Specification Language

We approach **RQ1** by introducing a declarative, SQL-based *data-aware process specification language* (dapSL) to describe processes operating over relational data. dapSL can be seen as a SQL-based front-end language for specifying *data-centric dynamic systems* (DCDSs) [1]. A dapSL specification consists of two main components: *(i)* a *data layer*, representing the structural aspects of the domain, and storing corresponding extensional data; *(ii)* a *control layer*, which queries and evolves the data of the data layer.

Example 1. As a running example, we consider a travel reimbursement process inspired by [7], whose control flow is depicted in Fig. 1. The process starts (StartWF) by checking pending employee travel requests in the DB. Then, after selecting a request, the system examines it (ReviewRequest), and decides whether to approve it or not. If approved, the process continues by calculating the maximum refundable amount, and the employee can go on her business trip. On arrival, she is asked to compile and submit a form with all the business trip expenses (FillReimb). The system analyzes the submitted form (ReviewReimb) and, if the estimated maximum has not been exceeded, approves the refunding. Otherwise the reimbursement is rejected. ◁

Data Layer. The *data layer* is a standard relational DB, consisting of an intensional (schema) part, and an extensional (instance) part. The intensional part is a *DB schema*, that is, a pair $\langle \mathcal{R}, \mathcal{E} \rangle$, where $\mathcal{R}$ is a finite set of *relation schemas*, and $\mathcal{E}$ is a finite set of *integrity constraints* over $\mathcal{R}$. To capture a DB schema, $\mathbb{dapSL}$ employs the standard SQL data definition language (DDL). For presentation reasons, in the remainder of the papers we refer to the components of a DB schema in an abstract way, following standard definitions. $\mathbb{dapSL}$ tackles three fundamental types of integrity constraints: a *primary key* for a relation R is denoted $\textsc{pk}(R)$; a *foreign key* from attributes B of relation S to attributes A of R is denoted $S[\mathsf{B}] \rightarrow R[\mathsf{A}]$; a *domain constraint* enumerates the values assigned to a relation attribute. Note that such constraints are expressed in $\mathbb{dapSL}$ by using the standard SQL DDL. The extensional part of the data layer is a *DB instance* (DB for short). $\mathbb{dapSL}$ delegates the representation of this part to the relational storage of choice. We always assume that a DB is consistent, that is, satisfies all constraints in $\mathcal{E}$. While the intensional part is fixed in a $\mathbb{dapSL}$ model, the extensional part starts from an initial DB that is then iteratively updated through the control layer, as dictated below.

Example 2. The $\mathbb{dapSL}$ DB schema for the process informally described in Example 1 is shown in Fig. 1. We recall of the relation schemas: *(i)* requests under process are stored in the relation *CurrReq*, whose components are the request UID, which is the primary key, the employee requesting a reimbursement, the trip destination, and the status of the request, which ranges over a set of predefined values (captured with a domain constraint); *(ii)* maximum allowed trip budgets are stored in *TrvlMaxAmnt*, whose components are the id (the primary key), the request reference number (a foreign key), and the maximum amount assigned for the trip; *(iii) TrvlCost* stores the total amount spent, with the same attributes as in *TrvlMaxAmnt*. ◁

Control Layer. The *control layer* defines how the data layer can be evolved through the execution of actions (concretely accounting for the different process tasks). Technically, the control layer is a triple $\langle \mathcal{F}, \mathcal{A}, \rho \rangle$, where $\mathcal{F}$ is a finite set of *external services*, $\mathcal{A}$ is a finite set of *atomic tasks* (or actions), and ρ is a *process specification.*

Each service is defined as a function signature that indicates how externally generated data can be brought into the process, abstractly accounting for a variety of concrete data injection mechanisms, e.g., user forms, web services, external ERP systems, internal generation of primary keys etc. Each service comes with a signature indicating the *service name*, its *formal input parameters* and their *types*, as well as the *output type*.

Actions are the basic building blocks of the control layer, and represent transactional operations over the data layer. Each action comes with a distinguished name and a set of formal parameters, and consists of a set of *parameterized SQL statements* that inspect and update the current state of the $\mathbb{dapSL}$ model (i.e., the current DB), using standard insert-delete SQL operations. operations are parameterized so as to allow referring with the statements to the action parameters, as well as to the results obtained by invoking a service call. Both kind of parameters are substituted with actual values when the action is concretely executed. Hence, whenever a SQL statement allows for using a constant value, $\mathbb{dapSL}$ allows for using either a constant, an action parameter, or a placeholder representing the invocation of a service call. To distinguish service invocations from action parameters, $\mathbb{dapSL}$ prefixes the service call name with @.

Formally, a $\mathbb{dapSL}$ *action* is an expression $\alpha(p_1, \ldots, p_n) : \{e_1; \ldots; e_m\}$, where: *(i)* $\alpha(p_1, \ldots, p_n)$ is the action *signature*, constituted by *action name* α and the set $\{p_1, \ldots, p_n\}$ of *action formal parameters*; *(ii)* $\{e_1, \ldots, e_m\}$ is a set of parameterized effect specifications, which are SQL insertions and deletions performed on the current DB. We assume that no two actions in $\mathcal{A}$ share the same name, and then use the action name to refer to its corresponding specification.

A *parameterized SQL insertion* has the form **INSERT INTO** $R(\mathsf{A}_1, \ldots, \mathsf{A}_k)$ **VALUES** $(t_1, \ldots, t_k)$, where $R \in \mathcal{R}$, and each t_j is either a value, an action formal parameter or a service call invocation (which syntactically corresponds to a scalar function call in SQL). Given a service call @F with parameters p, an invocation for F is of the form @F$(x_1, \ldots, x_p)$, where each x_j is either a value or an action formal parameter. **VALUES** can be substituted by a complex SQL inner selection query, which in turn supports *bulk insertions* into R by using all answers obtained by evaluating the inner query.

A *parameterized SQL deletion* has the form **DELETE FROM** R **WHERE** $\langle condition \rangle$, where $R \in \mathcal{R}$, and the **WHERE** clause may internally refer to the action formal parameters. This specification captures the simultaneous deletion of all tuples returned by the evaluation of *condition* on the current DB. Following classical conceptual modeling approaches to domain changes [17], we allow for overlapping deletions and insertions in such a way that first all deletions, and then all insertions are applied. This allows to unambiguously capture update effects (by deleting certain tuples, and inserting back variants of those tuples). Introducing explicit SQL update statements would create ambiguities on how to prioritize updates w.r.t. potentially overlapping deletions and insertions.

The executability of an action, including how its formal parameters may be bound to corresponding values, is dictated by the process specification ρ – a set of condition-action (CA) rules, again grounded in SQL, and used to declaratively

capture the control-flow of the $\mathbb{dapSL}$ model. For each action α in $\mathcal{A}$ with k parameters, ρ contains a single CA rule determining the executability of α. The CA rule is an expression of the form:

$$\texttt{SELECT } \mathsf{A}_1, \ldots, \mathsf{A}_s \texttt{ FROM } R_1, \ldots, R_m \texttt{ WHERE } \langle condition \rangle \texttt{ ENABLES } \alpha(\mathsf{A}_{n_1}, \ldots, \mathsf{A}_{n_k}),$$

where each A_i is an attribute, each $R_i \in \mathcal{R}$, $\alpha \in \mathcal{A}$, and $\{\mathsf{A}_{n_1}, \ldots, \mathsf{A}_{n_k}\} \subseteq \{\mathsf{A}_1, \ldots, \mathsf{A}_s\}$. Here, the SQL **SELECT** query represents the rule condition, and the results of the query provide alternative actual parameters that instantiate the formal parameters of α. This grounding mechanism is applied on a per-answer basis, that is, to execute α one has to choose how to instantiate the formal parameters of α with one of the query answers returned by the **SELECT** query. Multiple answers consequently provide alternative instantiation choices. Notice that requiring each action to have only one CA rule is w.l.o.g, as multiple CA rules for the same action can be compacted into a unique rule whose condition is the **UNION** of the condition queries in the original rules.

Example 3. We focus on three tasks of the process in Example 1, showing their encoding in $\mathbb{dapSL}$. StartWF creates a new travel reimbursement request by picking a pending requests from the current DB. We represent this in $\mathbb{dapSL}$ as an action with three formal parameters, denoting a pending request id, its responsible employee, and her intended destination:

```
SELECT id, emp, dest FROM Pending ENABLES StartWF(id, emp, dest);
StartWF(id, emp, dest):{DELETE FROM Pending WHERE Pending.id = id;
 INSERT INTO CurrReq(id, emp, dest, status) VALUES(@genpk(), emp, dest, submitd)}
```

Here, a new request is generated by removing from *Pending* the entry that matches the given id, then inserting a new tuple into *CurrReq* with the emp and dest values of the deleted tuple, and the status set to 'submitd'. To get a unique id for such a tuple, we invoke the nullary service call @genpk, returning a fresh primary key value. ReviewRequest examines an employee trip request and, if accepted, assigns its maximum reimbursable amount. The action can be executed only if a request in *CurrReq* actually exists:

```
SELECT id, emp, dest FROM CurrReq WHERE CurrReq.status = 'submitd'
  ENABLES RvwRequest(id, emp, dest);
RvwRequest(id, emp, dest):{DELETE FROM CurrReq WHERE CurrReq.id = id;
                           INSERT INTO CurrReq(id, emp, dest, status)
                             VALUES(id, emp, dest, @status(emp, dest));
                           INSERT INTO TrvlMaxAmnt(tid, tfid, tmaxAmnt)
                             VALUES(@genpk(), id, @maxAmnt(emp, dest))}
```

The request status of *CurrReq* is updated by calling service @status, that takes as input an employee name and a trip destination, and returns a new status value. Also, a new tuple containing the maximum reimbursable amount is added to *TrvlMaxAmnt*. To get the maximum refundable amount for *TrvlMaxAmnt*, we employ service @maxAmnt with the same arguments as @status.

Task FillReimb updates the current request by adding a compiled form with all the trip expenses. This can be done only when the request has been accepted:

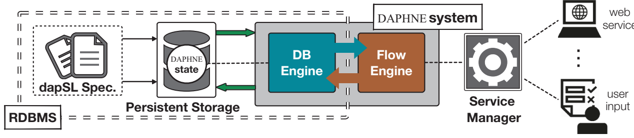

Fig. 2. Conceptual architecture of DAPHNE

$\texttt{\textbf{SELECT}}$ id, emp, dest $\texttt{\textbf{FROM}}$ $\mathit{CurrReq}$ $\texttt{\textbf{WHERE}}$ $\mathit{CurrReq}$.status = 'acceptd'
  $\texttt{\textbf{ENABLES}}$ $\texttt{FillReimb}$(id, emp, dest) ;
$\texttt{FillReimb}$(id, emp, dest) : { $\texttt{\textbf{INSERT INTO}}$ $\mathit{TrvlCost}$(id, fid, cost)
  $\texttt{\textbf{VALUES}}$(@genpk(), id, @cost(emp, dest)) }

Again, @genpk and @cost are used to obtain values externally upon insertion. ◁

Execution Semantics. The semantics mimics that of DCDSs [1]. Let $\mathcal{I}$ be the current DB for the data layer of the dapSL model of interest. An action α is *enabled* in $\mathcal{I}$ if the evaluation of the SQL query constituting the condition in the CA rule of α returns a nonempty result set. This result set is then used to instantiate α, by non-deterministically picking an answer tuple, and use it to bind the formal parameters of α to actual values. This produces a so-called *ground action* for α. The execution of a ground action amounts to simultaneous application of all its effect specifications, which requires to first manage the service call invocations, and then apply the deletions and insertions. This is done as follows. First, invocations in the ground action are instantiated by resolving the subqueries present in all those insertion effects whose values contain invocation placeholders. Each invocation then becomes a fully specified call to the corresponding service, passing the ground values as input. The result obtained from the call is used to replace the invocation itself, getting a fully instantiated **VALUES** clause for the insertion effect specification. A transactional update is consequently issued on $\mathcal{I}$, first pushing all deletions, and then all insertions. If the resulting DB satisfies the constraints of the data layer, then the update is committed, otherwise it is rolled back.

3 The DAPHNE System

We discuss how dapSL has been implemented in a concrete system, called DAPHNE, that provides in-database process enactment and state-space construction for formal analysis. In particular, the first feature is required to address **RQ2** by realizing three main functionalities: *(1)* indicate the executable actions and their parameters; *(2)* manage the invocation of an executable action and the corresponding update; *(3)* handle normal execution vs. execution with logging of historical snapshots transparently to the user.

The core architecture of DAPHNE is depicted in Fig. 2. The system takes as input a representation of a dapSL specification (or model) and uses a standard

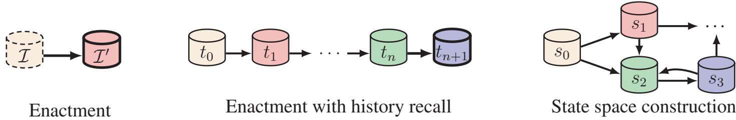

Fig. 3. The three main usage modalities for DAPHNE, and sketch of the corresponding data structures stored within the DBMS

database management system (DBMS) to support its execution. The DBMS takes care of storing the data relevant to the input dapSL model and supports, through the ***DB Engine*** of the underlying DBMS, the application of a set of operations that jointly realize the given dapSL actions. The ***Flow Engine*** constitutes the application layer of the system; it facilitates the execution of a dapSL model by coordinating the activities that involve the user, the DBMS, and the services. Specifically, the **Flow Engine** issues queries to the DBMS, calls stored procedures, and handles the communication with external services through a further module called ***Service Manager***.

Next we give a detailed representation of DAPHNE's architecture by describing the stages of each execution step. For the moment we do not consider the concrete encoding of dapSL inside the DBMS. At each point in time, the DBMS stores the current state of the dapSL model. We assume that, before the execution starts, the DBMS contains an initial database instance for the data layer of dapSL model. To start the execution, the **Flow Engine** queries the DBMS about the actions that are enabled in the current state; if one is found, the engine retrieves all possible parameter assignments that can be selected to ground the action, and returns them to the user (or the software module responsible for the process enactment). The user is then asked to choose one of such parameter assignments. At this point, the actual application of the ground action is triggered. The **Flow Engine** invokes a set of stored procedures from the DBMS that take care of evaluating and applying action effects. If needed by the action specification, the **Flow Engine** interacts with external services, through the **Service Manager**, to acquire new data via service calls. The tuples to be deleted and inserted in the various relations of the dapSL model are then computed, and the consequent changes are pushed to the DBMS within a transaction, so that the underlying database instance is updated only if all constraints are satisfied. After the update is committed or rolled back, the action execution cycle can be repeated by selecting either a new parameter assignment or another action available in the newly generated state.

3.1 Encoding a dapSL in DAPHNE

We now detail how DAPHNE encodes a dapSL model $\mathcal{S} = \langle \mathcal{L}, \mathcal{P} \rangle$ with data layer $\mathcal{L} = \langle \mathcal{R}, \mathcal{E} \rangle$ and control layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \rho \rangle$ into a DBMS. Intuitively DAPHNE represents $\mathcal{L}$ as a set of tables, and $\mathcal{P}$ as a set of stored procedures working over those and auxiliary tables. Such data structures and stored procedures are

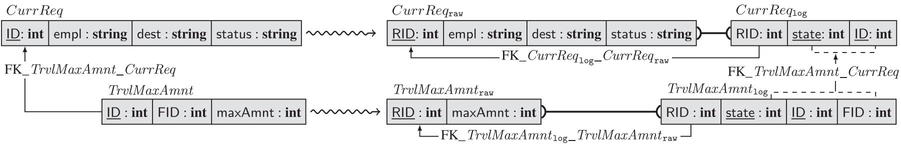

Fig. 4. Relational schemas of *CurrReq* and *TrvlMaxAmnt*, and their historical representation in DAPHNE via two pairs of corresponding tables: $CurrReq_{\texttt{raw}}$ and $CurrReq_{\texttt{log}}$, $TrvlMaxAmnt_{\texttt{raw}}$ and $TrvlMaxAmnt_{\texttt{log}}$.

defined in terms of the native language of the chosen DBMS. These can be either created manually, or automatically instrumented by DAPHNE itself if the user uses the DAPHNE APIs to communicate the definition of $\mathcal{S}$. We employ jOOQ (https://www.jooq.org/) as the basis for the concrete input syntax of dapSL models within DAPHNE. An overview about how jOOQ and the APIs actually work is given in a companion report [5].

Before entering into the encoding details, it is important to stress that DAPHNE provides three main usage modalities. The first modality is *enactment*. Here DAPHNE supports users in the process execution, storing the *current DB*, and suitably updating it in response to the execution of actions. The second modality is *enactment with historical recall*. This corresponds to enactment, but storing all the historical DBs together with information about the applied actions (name, parameters, service call invocations and results, and timestamps). This provides full traceability about how the process execution evolved the initial state into the current one. The last modality is *state space construction for formal analysis*, where DAPHNE generates all possible "relevant" possible executions of the system, using abstract state identifiers instead from timestamps, and using representative DBs to compactly represent (infinitely many) DBs from which $\mathcal{S}$ wold induce the same evolution. This allows DAPHNE to connect back to representative states in case the execution of an action leads to a configuration of the DB that has been already encountered before. Technically, traces of $\mathcal{S}$ are folded into an abstract *relational transition system* (RTS) [4], which faithfully represents not only all possible runs of $\mathcal{S}$, but also its branching structure.

Data Layer. DAPHNE does not internally store the data layer as it is specified in $\mathcal{L}$, but adopts a more sophisticated schema. This is done to have a unique homogeneous approach that supports the three usage modalities mentioned before. In fact, instructing the DBMS to directly store the schema expressed in $\mathcal{L}$ would suffice only in the enactment case, but not to store historical data about previous states, nor the state space with its branching nature. To accommodate all three usages at once, DAPHNE proceeds as follows. Each relation schema R of $\mathcal{L}$ becomes relativized to a *state identifier*, and decomposed into two interconnected relation schemas: *(i)* $R_{\texttt{raw}}$ (*raw data storage*), an inflationary table that incrementally stores all the tuples that have been ever inserted in R; *(ii)* $R_{\texttt{log}}$ (*state log*), which is responsible at once for maintaining the referential integrity of the data in a state, as well as for fully reconstructing the exact content of R

in a state. In details, $R_{\texttt{raw}}$ contains all the attributes A of R that are *not* part of primary keys nor sources of a foreign key, plus an additional surrogate identifier RID, so that $\text{PK}(R_{\texttt{raw}}) = \langle \mathsf{RID} \rangle$. Each possible combination of values over A is stored only once in $R_{\texttt{raw}}$ (i.e., $R_{\texttt{raw}}[A]$ is a key), thus maximizing compactness. At the same time, $R_{\texttt{log}}$ contains the following attributes: *(i)* an attribute state representing the state identifier; *(ii)* the primary key of (the original relation) R; *(iii)* a reference to $R_{\texttt{raw}}$, i.e., an attribute RID with $R_{\texttt{log}}[\mathsf{RID}] \rightarrow R_{\texttt{raw}}[\mathsf{RID}]$; *(iv)* all attributes of R that are sources of a foreign key in $\mathcal{L}$. To guarantee referential integrity, $R_{\texttt{log}}$ must ensure that (primary) keys and foreign keys are now relativized to a state. This is essential, as the same tuple of R may evolve across states, consequently requiring to historically store its different versions, and suitably keep track of which version refers to which state. Also foreign keys have to be understood within the same state: if a reference tuple changes from one state to the other, all the other tuples referencing it need to update their references accordingly. To realize this, we set $\text{PK}(R_{\texttt{log}}) = \langle \text{PK}(R), \mathsf{state} \rangle$. Similarly, for each foreign key $S[B] \rightarrow R[A]$ originally associated to relations R and S in $\mathcal{L}$, we insert in the DBMS the foreign key $S_{\texttt{log}}[B, \mathsf{state}] \rightarrow R_{\texttt{log}}[A, \mathsf{state}]$ over their corresponding state log relations.

With this strategy, the "referential" part of R is suitably relativized w.r.t. a state, while at the same time all the other attributes are compactly stored in $R_{\texttt{raw}}$, and referenced possibly multiple times from $R_{\texttt{log}}$. In addition, notice that, given a state identified by $\texttt{s}$, the full extension of relation R in $\texttt{s}$ can be fully reconstructed by *(i)* selecting the tuples of $R_{\texttt{log}}$ where $\mathsf{state} = \texttt{s}$; *(ii)* joining the obtained selection with $R_{\texttt{raw}}$ on RID; *(iii)* finally projecting the result on the original attributes of R. In general, this technique shows how an arbitrary SQL query over $\mathcal{L}$ can be directly reformulated as a state-relativized query over the corresponding DAPHNE schema.

Example 4. Consider relation schemas *CurrReq* and *TrvlMaxAmnt* in Fig. 1. Figure 4 shows their representation in DAPHNE, suitably pairing $\mathit{CurrReq}_{\texttt{raw}}$ with $\mathit{CurrReq}_{\texttt{log}}$, and $\mathit{TrvlMaxAmnt}_{\texttt{raw}}$ with $\mathit{TrvlMaxAmnt}_{\texttt{log}}$. Each state log table directly references a corresponding raw data storage table (e.g., $\mathit{CurrReq}_{\texttt{log}}[\mathsf{RID}] \rightarrow \mathit{CurrReq}_{\texttt{raw}}[\mathsf{RID}]$), and *TrvlMaxAmnt*'s state log table, due to the FK in the original DAP, will reference a suitable key of $\mathit{CurrReq}_{\texttt{log}}$ (i.e., $\mathit{TrvlMaxAmnt}_{\texttt{log}}[\mathsf{state}, \mathsf{FID}] \rightarrow \mathit{CurrReq}_{\texttt{log}}[\mathsf{state}, \mathsf{ID}]$). Figures 5, 6 and 7 show the evolution of the DBMS in response to the application of three ground actions, with full history recall. ◁

We now discuss updates over R. As already pointed out, $R_{\texttt{raw}}$ stores any tuple that occurs in some state, that is, tuples are never deleted from $R_{\texttt{raw}}$. Deletions are simply obtained by *not* referencing the deleted tuple in the new state. For instance, in Fig. 5, it can be seen that the first tuple of $\mathit{Pending}_{\texttt{raw}}$ (properly extended with its ID, through RID) has been deleted from *Pending* in state $\mathsf{2}$: while being present in state $\texttt{1}$ (cf. first tuple of $\mathit{Pending}_{\texttt{log}}$), the tuple is not anymore in state $\mathsf{2}$ (cf. third tuple of $\mathit{Pending}_{\texttt{log}}$).

As for additions, we proceed as follows. Before inserting a new tuple, we check whether it is already present in $R_{\texttt{raw}}$. If so, we update only $R_{\texttt{log}}$ by copying the

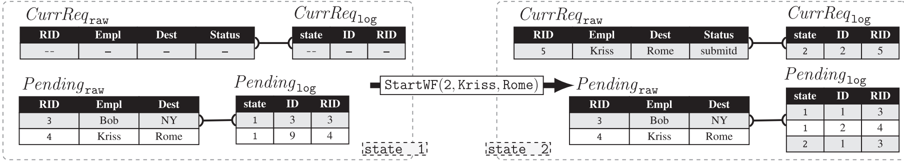

Fig. 5. Action application with two partial DB snapshots mentioning *Pending* and *CurrReq*. Here, `StartWF` is applied in state 1 with $\{$id $= 2,$ empl $=$ `Kriss`, dest $=$ `Rome`$\}$ as binding and, in turn, generates a new state 2.

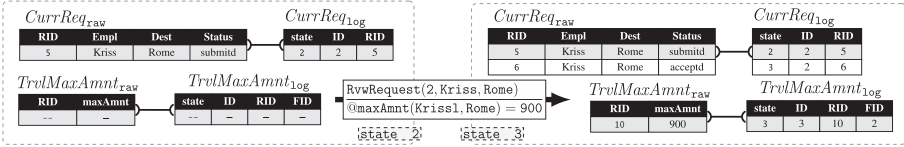

Fig. 6. Action application with two partial DB snapshots mentioning *CurrReq* and *TrvlMaxAmnt*. Here, `RvwRequest` is applied in state 2 with $\{$id $= 2,$ empl $=$ `Kriss`, dest $=$ `Rome`$\}$ as binding and 900 resulting from the invocation of service `@maxAmnt` that, in turn, generates a new state 3.

$R_{\texttt{log}}$ tuple referencing the corresponding RID in $R_{\texttt{raw}}$. In the copied tuple, the value of attribute state is going to be the one of the newly generated state, while the values of ID and all foreign key attributes remain unchanged. If the tuple is not present in $R_{\texttt{raw}}$, it is also added to $R_{\texttt{raw}}$ together with a fresh RID. Notice that in that case its ID and FK attributes are provided as input, and thus they are simply added, together with the value of state, to $R_{\texttt{log}}$. In the actual implementation, $R_{\texttt{raw}}$ features also a hash attribute, with the value of a hash function computed based on original R attributes (extracted from both $R_{\texttt{raw}}$ and $R_{\texttt{log}}$). This speeds up the search for identical tuples in $R_{\texttt{raw}}$.

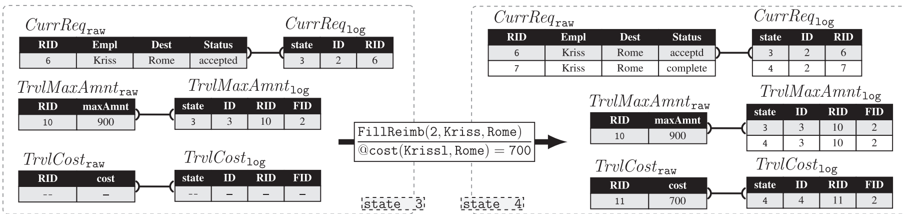

Fig. 7. Action application with three partial DB snapshots mentioning *CurrReq*, *TrvlMaxAmnt* and *TrvlCost*. Here, `FillReimb` is applied in state 3 with $\{$id $= 2,$ empl $=$ `Kriss`, dest $=$ `Rome`$\}$ as binding and 700 resulting from the invocation of service `@cost` that, in turn, generates a new state 4.

Finally, we consider the case of relation schemas whose content is not changed when updating a state to a new state. Assume that relation schema S stays unaltered. After updating R, it is enough to update $S_{\texttt{log}}$ by copying previous state entries and updating the value of their state id to the actual one. If a FK, whose left-hand side is $S[B]$, belongs to $\mathcal{L}$, the pair $\langle B, \texttt{state}\rangle$ will reference the most recent versions of the previously referenced tuples. Consider, e.g., Fig. 7. While in his current request `Kriss` is changing the request status when moving from state `3` to state `4`, the maximum traveling budget assigned to this request (a tuple in *TrvlMaxAmnt*) should reference the latest version of the corresponding tuple in *CurrReq*. Indeed, in state `4`, a new tuple in $\mathit{TrvlMaxAmnt}_{\texttt{log}}$ is referencing a new tuple in $\mathit{CurrReq}_{\texttt{log}}$ that, in turn, corresponds to the one with the updated request status.

Control Layer. Each action α of $\mathcal{P}$, together with its dedicated CA rule, is encoded by DAPHNE into three stored procedures. The encoding is quite direct, thanks to the fact that both action conditions and action effect specifications are specified using SQL.

The first stored procedure, `α_ca_eval(s)`, evaluates the CA rule of α in state `s` over the respective DB (obtained by inspecting the state log relations whose state column matches with `s`), and stores the all returned parameter assignments for α in a dedicated table `α_params`. All parameter assignments in `α_params` are initially unmarked, meaning that they are available for the user to choose. The second stored procedure, `α_eff_eval(s,b)`, executes queries corresponding to all the effects of α over the DB of state `s`, possibly using action parameters from `α_params` extracted via a binding identifier `b`. Query results provide values to instantiate service calls, as well as those facts that must be deleted from or added to the current DB. The third stored procedure, `α_eff_exec(s,b)`, transactionally performs the actual delete and insert operations for a given state `s` and a binding identifier `b`, using the results of service calls. It is worth noting that, in our running example, stored procedures that represent an action, all together, in average contain around 40 complex queries. We now detail the DAPHNE action execution cycle in a given state `s`. *(1)* The cycle starts with the user choosing one of the available actions presented by the **Flow Engine**. The available actions are acquired by calling `α_ca_eval(s)`, for each action α in $\mathcal{P}$. *(2)* If any unmarked parameter is present in `α_params`, the user is asked to choose one of those (by selecting a binding identifier `b`); once chosen, the parameter is marked, and the **Flow Engine** proceeds to the evaluation of α by calling `α_eff_eval(s,b)`. If there are no such parameters, the user is asked to choose another available action, and the present step is repeated. *(3)* If `α_eff_eval(s,b)` involves service calls, these are passed to the **Service Manager** component, which fetches the corresponding results. *(4)* `α_eff_exec(s,b)` is executed. If all constraints in $\mathcal{L}$ are satisfied, the transaction is committed and a new iteration starts from step 1; otherwise, the transaction is aborted and the execution history is kept unaltered, and the execution continues from step 2.

3.2 Realization of the Three Usage Modalities

Let us now discuss how the three usage modalities are realized in DAPHNE. The simple enactment modality is realized by only recalling the current information about log relations. Enactment with history recall is instead handled as follows. First, the generation of a new state always comes with an additional update over an accessory 1-tuple relation schema indicating the timestamp of the actual update operation. The fact that timestamps always increase along the execution guarantees that each new state is genuinely different from previously encountered ones. Finally, an additional binary state transition table is employed, so as to keep track of the resulting total order over state identifiers. By considering our running example, in state 4 shown in Fig. 7, the content of the transition table would consist of the three pairs $\langle 1, 2\rangle$, $\langle 2, 3\rangle$, and $\langle 3, 4\rangle$.

We now discuss state space construction, which is the cornerstone of **RQ3** and that should be ideally realized in a way that adheres to how the enactment engine works. In this way, we ensure that the state space is carried out on the exact same model that is enacted. Due to the presence of external services that may inject fresh input data, there are in general infinitely many different executions of the process, possibly visiting infinitely many different DBs (differing in at least one tuple). In other words, the resulting transition system has infinitely many different states. However, thanks to the correspondence between $\mathbb{dapSL}$ and DCDSs, we can realize in DAPHNE the abstraction techniques from [1,4] to attack the verification of such infinite-state transition systems. The main idea behind such techniques is the following. When carrying out verification, it is not important to observe all possible DBs that can be produced by executing the available actions with all possible service call results, but it suffices to only consider a *meaningful* combination of values, representing all possible ways to relate tuples with other tuples in the DB, in terms of (in)equality of their different components. This is done by carefully selecting the representative values. In [1,4], it has been shown that this technique produces a *faithful* representation of the original RTS, and that this representation is also *finite* if the original system is *state bounded*, that is, has a pre-defined (possibly unknown) bound on the number of tuples that can be stored therein.[1] Constructing the state space is therefore instrumental towards verification of properties such as reachability and soundness, and temporal model checking (in the style of [4]).

State space construction is smoothly handled in DAPHNE as follows. When executed in this mode, DAPHNE replaces the service call manager with a mock-up manager that, whenever a service call is invoked, returns all and only *meaningful* results, in the technical sense described above. E.g., if the current DB only contains string a, invoking a service call that returns a string may only give two interesting results: a itself, or a string different than a. To cover the latter case, the mock-up manager picks a representative value, say b in this example, implementing the representative selection strategy defined in [1,4]. With this mock-up manager in place, DAPHNE constructs the state space by executing

[1] Even in the presence of this bound, infinitely many different DBs can be encountered, by changing the values stored therein.

the following iteration. A state s is picked (at the beginning, only the initial state exists). For each enabled ground action in s, and for all *relevant* possible results returned by the mock-up manager, the DB instance corresponding to the update is generated. If such a DB instance has been already encountered (i.e., is associated to an already existing state), then the corresponding state id s' is fetched. Otherwise, a new id s' is created, inserting its content into the DBMS. Recall that s' is *not* a timestamp, but just a symbolic, unique state id. The state transition table is then updated, by inserting $\langle \mathsf{s}, \mathsf{s}' \rangle$, which indeed witnesses that s' is one of the successors of s. The cycle is then repeated until all states and all enabled ground actions therein are processed. Notice that, differently from the enactment with history recall, in this case the state transition table is graph-structured, and in fact reconstructs the abstract representation of the (RTS) system capturing the execution semantics of $\mathcal{S}$. In [5] we give a few examples of the state spaces constructed for the travel reimbursement process using a special visualizer API implemented in DAPHNE mainly for debugging purposes.

We report here some initial experimental results on state space construction in DAPHNE, demonstrating the complexity in constructing a transition system whose states are full-fledged relational DBs.

Table 1. Experimental results

#(F)	**#(States)**	**#(Edges)**	**avg(Time)**
1	10	10	0.93
2	128	231	4.36
3	1949	5456	114.39
4	32925	128155	6575.95

All experiments were performed on a MacOS machine with a 2.4 GHz Intel Core i5 and 8 GB RAM, encoding and storing the travel management process in PostgreSQL 9.4. Table 1 shows results for the construction of abstract RTSs on randomly generated initial DBs with a number of facts specified in **#(F)**. Each experiments come with a time limit of 7200 s, which suffices to generate relatively big abstract RTSs and recognize growth patterns in the conducted experiments. We provide full statistics on the generated abstract RTS: **#(S)** represents the number of states, **#(E)** the number of edges, and **avg(Time)** the average RCYCL execution time. The most critical measures are **Time** and **#(E)**, with the latter showing the number of successful action executions, each of which consists in the application of the corresponding SQL code. Such a code consists, for the travel management process, of ∼40 complex queries per action. This gives an indication about the feasibility of our approach, but also points out that optimizations have to be studied to handle very large state spaces.

4 Discussion and Related Work

Our approach belongs to the line of research focused on modeling, enactment and verification of data-centric processes. Specifically, DAPHNE directly relates to: *(i)* the declarative rule-based *Guard-Stage-Milestone* (GSM) language [6] and its BizArtifact execution platform; *(ii)* the OMG *CMMN* standard for case handling (https://www.omg.org/spec/CMMN/); *(iii)* the object-aware business process management approach implemented by *PHILharmonic Flows* [12]; *(iv)*

the extension of GSM called EZ-Flow [23], with SeGA [22] as an execution platform; *(v)* the declarative data-centric process language RESEDA based on term rewriting systems [19]. These approaches emphasize the evolution of data objects through different states, but often miss a clear representation of the control-flow dimension. For example, GSM provides means for specifying business artifact lifecycles in a declarative rule-based manner, and heavily relies on ECA rules over the data to implicitly define the allowed execution flows. Other examples in *(ii)*–*(iv)* are rooted in similar abstractions, but provide more sophisticated interaction mechanisms between artifacts and their lifecycle components. dapSL shares with these approaches the idea of "data-centricity", but departs from them since it provides a minimalistic, programming-oriented solution that only retains the notions of persistent data, actions, and CA rules. In this respect, the closest approach to ours is RESEDA. Similarly to dapSL, a RESEDA process consists of reactive rules and behavioral constraints operating over data using the paradigm of term rewriting. RESEDA manipulates only semi-structured data (such as XML or JSON), which have to be specified directly in the tool. dapSL focuses instead on the standard relational model to represent data.

Differently from all such approaches, state-of-the-art business process management systems (BPMSs), such as Bizagi, Bonita BPM, Camunda, Activiti, and YAWL, support an explicit, conceptual representation of the process control flow and the lifecycle of activities, following conventional process modeling notations such as the de-facto standard BPMN. However, such notations tackle only the process logic, and do not provide any conceptual means to address the decision and task logic. Consequently, no conceptual guidelines are given to such BPMSs when it comes to the interplay between the process and the underlying persistent data. The result is that they see the data logic as a "procedural attachment", i.e., a piece of code whose functioning is not captured at the conceptual level, making it difficult to properly understand and govern the so-obtained integrated models [3,8,18]. On the positive side, contemporary BPMSs typically adopt a high-level representation of persistent data, e.g., in the form of object-oriented or conceptual data models. We comment on the relationship and possible synergies between such approaches and dapSL, considering how dapSL could be enhanced with an explicit representation of control-flow, and/or with a conceptual layer for representing the data.

dapSL with Explicit Control-Flow. At the modeling level, it is natural to envision an integrated approach where the process logic (including control-flow, event-driven behavior, hierarchical decomposition and activity lifecycle) is specified using conventional notations such as BPMN, the decision logic is specified in standard SQL, and the task logic using dapSL actions. The main question then becomes how the resulting approach can be enacted/analyzed. An option is to apply a translation of the process and decision logic into dapSL rules, and then rely on DAPHNE for enactment and state-space construction. Thanks to the correspondence between dapSL and DCDSs, this can be done by directly implementing the existing translation procedures from process modeling notations to DCDSs, where each control-flow pattern and step in the lifecycle of activi-

ties/artifacts becomes a dedicated CA rule. Translations have been defined for: *(i)* data-centric process models supporting the explicit notion of process instance (i.e., case) [14]; *(ii)* Petri nets equipped with resources and tokens with data [15]; *(iii)* recent variants of (colored) Petri nets equipped with a DB storage [7,16]; *(iv)* GSM-based business artifacts [20].

dapSL with Conceptual Data Models. It is often desirable to specify processes at a higher level of abstraction than the database level (e.g., in the form of an object model or ontology). However, if storage stays at the relational level, introducing an intermediate conceptual layer poses key theoretical challenges, not specifically related to our approach, but in general to the alignment between the conceptual and the storage layers. In this respect, the most difficult problem, reminiscent of the long-standing view update problem in database theory [9], is to suitably propagate updates expressed over the conceptual layer on the actual data storage. Some recent works attacked this problem by introducing intermediate data structures [22], or by establishing complex mappings to disambiguate how updates should be propagated [21]. This is not needed if the conceptual layer is a "lossless view" of the data layer, ensuring that high-level updates can always be rewritten into relational updates. This is the approach adopted in Object-Relational Mapping (ORM), such as Hibernate. A conceptual data layer realized as an ORM specification can be directly tackled in DAPHNE. In fact, dapSL can easily be reformulated by using ORM query languages (such as Hibernate HQL) instead of SQL. Then, the back-end of DAPHNE can be simply modified so as to invoke, when needed, the translation of high-level queries into SQL, a functionality natively offered by ORM technologies.

5 Conclusions

We have introduced a declarative, purely relational framework for data-aware processes, in which SQL is used as the core data inspection and update language. We have reported on its implementation in the DAPHNE tool, which at once accounts for modeling, enactment, and state space construction for verification. Since our approach is having a minimalistic, SQL-centric flavor, it is crucial to empirically validate its usability among database and process experts. We also intend to interface DAPHNE with different concrete end user-oriented languages for the integrated modeling of processes and data. As for formal analysis, we plan to augment the state space construction with native temporal model checking capabilities. Finally, given that DAPHNE can generate a log including all performed actions and data changes, we aim at investigating its possible applications to (multi-perspective) process mining.

Acknowledgments. This research has been partially supported by the UNIBZ projects *REKAP* and *DACOMAN*.

References

1. Hariri, B.B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proceedings of PODS (2013)
2. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of agent-based artifact systems. J. Artif. Intell. Res. **51**, 333–376 (2014). https://doi.org/10.1613/jair.4424
3. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: a database theory perspective. In: Proceedings of PODS (2013)
4. Calvanese, D., De Giacomo, G., Montali, M., Patrizi, F.: FO μ-calculus over generic transition systems and applications to the situation calculus. Inf. Comp. **259**(3), 328–347 (2018)
5. Calvanese, D., Montali, M., Patrizi, F., Rivkin, A.: Modelling and enactment of data-aware processes. CoRR abs/1810.08062 (2018). http://arxiv.org/abs/1810.08062
6. Damaggio, E., Hull, R., Vaculín, R.: On the Equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 396–412. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2_29
7. De Masellis, R., Di Francescomarino, C., Ghidini, C., Montali, M., Tessaris, S.: Add data into business process verification: bridging the gap between theory and practice. In: Proceedings of AAAI. AAAI Press (2017)
8. Dumas, M.: On the convergence of data and process engineering. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 19–26. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23737-9_2
9. Furtado, A.L., Casanova, M.A.: Updating relational views. In: Kim, W., Reiner, D.S., Batory, D.S. (eds.) Query Processing in Database Systems. Topics in Information Systems, pp. 127–142. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82375-6_7
10. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88873-4_17
11. Köpke, J., Su, J.: Towards quality-aware translations of activity-centric processes to guard stage milestone. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 308–325. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_18
12. 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), 19–46 (2011)
13. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: a practical verifier for artifact systems. PVLDB **11**(3), 283–296 (2017)
14. Montali, M., Calvanese, D.: Soundness of data-aware, case-centric processes. Int. J. Softw. Tools Technol. Transf. **18**(5), 535–558 (2016)
15. Montali, M., Rivkin, A.: Model checking Petri nets with names using data-centric dynamic systems. Formal Aspects Comput. **28**, 615–641 (2016)
16. Montali, M., Rivkin, A.: DB-Nets: on the marriage of colored Petri Nets and relational databases. In: Koutny, M., Kleijn, J., Penczek, W. (eds.) Transactions on Petri Nets and Other Models of Concurrency XII. LNCS, vol. 10470, pp. 91–118. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-55862-1_5

17. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-39390-0
18. Reichert, M.: Process and data: two sides of the same coin? In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 2–19. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_2
19. Seco, J.C., Debois, S., Hildebrandt, T.T., Slaats, T.: RESEDA: declaring live event-driven computations as reactive semi-structured data. In: Proceedings of EDOC, pp. 75–84 (2018)
20. Solomakhin, D., Montali, M., Tessaris, S., De Masellis, R.: Verification of artifact-centric systems: decidability and modeling issues. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 252–266. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_18
21. Sun, Y., Su, J., Wu, B., Yang, J.: Modeling data for business processes. In: Proceedings of ICDE, pp. 1048–1059. IEEE Computer Society (2014)
22. Sun, Y., Su, J., Yang, J.: Universal artifacts: a new approach to business process management (BPM) systems. ACM TMIS **7**(1), 3 (2016)
23. Xu, W., Su, J., Yan, Z., Yang, J., Zhang, L.: An artifact-centric approach to dynamic modification of workflow execution. In: Meersman, R., et al. (eds.) OTM 2011. LNCS, vol. 7044, pp. 256–273. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25109-2_17