Leveraging Relational Technology for Data-Centric Dynamic Systems

Diego Calvanese, Marco Montali, Fabio Patrizi, Andrey Rivkin

Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
{calvanese,montali,patrizi,rivkin}@inf.unibz.it

Abstract. We base our work on a model called data-centric dynamic system (DCDS), which can be seen as a framework for modeling and verification of systems where both the process controlling the dynamics and the manipulation of data are equally central. More specifically, a DCDS consists of a data layer and a process layer, interacting as follows: the data layer stores all the data of interest in a relational database, and the process layer modifies and evolves such data by executing actions under the control of a process, and possibly injecting into the system external data retrieved through service calls. In this work, we propose an implementation of DCDSs in which all aspects concerning not only the data layer but also the process layer, are realized by means of functionalities provided by a relational DBMS. We present the architecture of our prototype system, describe its functionality, and discuss the next steps we intend to take towards realizing a full-fledged DCDS-based system that supports verification of rich temporal properties.

1 Introduction

Modeling and analyzing the correctness of today's complex business processes is a very challenging task, that touches on the one side the management of static (data-related) aspects, and on the other side dynamic (process-related) concerns. Traditional approaches deal with these two pillars separately, and this *divide et impera* approach has led to the development of successful theories and technologies, such as:

- databases, ontologies and information integration to account for static aspects;
- business process management, service-oriented computing, formal verification and model checking for dynamic ones.

However, it has been extensively argued that this separation prevents business experts and analysts from understanding the organization as a whole, and of taking corresponding strategic decisions [13,18]. Therefore, more recently these two aspects have been addressed together, and this has lead to a flourishing literature dealing with the formal foundations [4,5] of *data-aware (business) processes*, as well as languages [15,12,16], and integrated software platforms [14] for modeling and running them.

In this spectrum, an important dimension regards whether the process works over a data component that is assumed to completely or only partially capture the domain knowledge. In this work, we focus on complete information, and consider in particular the framework of *data-centric dynamic systems* (DCDSs) [2], which tackles modeling and verification of data-aware processes running over a full-fledged relational database

with integrity constraints. On top of this relational database, a process modifies and evolves the data by executing (update) actions, possibly injecting external data retrieved through service calls. As pointed out in [2] DCDSs are expressively equivalent to the artifact-centric model. Furthermore, they can embed virtually all approaches proposed in the literature for formally capturing data-aware dynamic systems [12,5], including the well-known relational transducers [1].

In the last decade, verification of data-aware processes has been mainly studied from the foundational point of view [12,5], with the main goal of understanding the conditions that guarantee its decidability. Much less attention has been devoted to the actual implementation of corresponding verification methods, with only few exceptions, notably [10].

In this work, we discuss our ongoing effort towards the implementation of a system for running and verifying DCDSs. In particular, our objective has been to fully implement DCDSs by means of functionalities provided by a relational DBMS, in such a way that all aspects concerning both the data layer and the process layer are directly realized. Notably, the same implementation is used both for execution and verification, without incurring in the typical dychotomy between the running system and its verification model.

The paper is organized as follows. In Section 2, we provide an overview of DCDSs. In Section 3, we discuss a realistic example of DCDS, in which we highlight all the distinctive features of our model. In Section 4, we describe the architecture of our prototype system and its functionality. We then discuss, in Section 5, the next steps we intend to take towards realizing a full-fledged DCDS-based system that supports verification of rich temporal properties. Conclusions follow.

2 Data-Centric Dynamic Systems

A DCDS is a tuple $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D}$ is the *data layer* and $\mathcal{P}$ is the *process layer*. The data layer defines the data model of $\mathcal{S}$; it is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, where: $\mathcal{C}$ is a countably infinite set of constants providing data values, $\mathcal{R}$ is a a relational schema equipped with equality and full denial constraints $\mathcal{E}$, and $\mathcal{I}_0$ is an initial database instance over $\mathcal{C}$ that conforms to the schema $\mathcal{R}$ and satisfies the constraints $\mathcal{E}$. The *process layer* defines the progression mechanism for the DCDS; it is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \rho \rangle$, where: $\mathcal{F}$ is a finite set of functions representing calls to external services, $\mathcal{A}$ is a finite set of (update) actions, and ρ is a process specification.

Intuitively, the process layer captures the dynamics of the domain of interest, while the data layer captures its static properties. More specifically, the process layer describes the actions ($\mathcal{A}$) that can be executed to query and/or update the current state of the system (whose structure conforms to the data layer), and how/when such actions can be executed (process ρ). Some actions need to take values from the external environment; these can be obtained by performing service calls (from $\mathcal{F}$). Every action execution must guarantee that all the constraints in $\mathcal{E}$ are satisfied by the data layer, so as to prevent the exeuction of actions leading to states that are inconsistent with respect to the constraints.

An *action* $\alpha \in \mathcal{A}$ is an expression $\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$ where $\alpha(p_1, \ldots, p_n)$ is the action *signature*, with α the *action name* and $p_1, \ldots, p_n$ the *action parameters*,

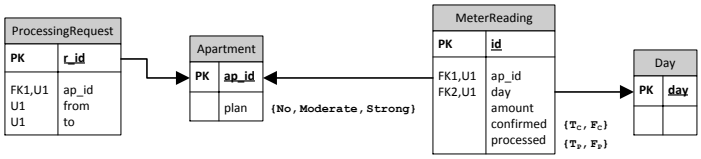

Fig. 1. The electrical readings processing data component

and $\{e_1, \ldots, e_m\}$ is a set of *(simultaneous) effect specifications*. Each e_i has the form $q_i \rightsquigarrow$ **del** D_i, **add** A_i, where: q_i is a query over $\mathcal{R}$ whose terms are variables, action parameters, and constants from $\mathtt{ADOM}(\mathcal{I}_0)$[1]; and D_i and A_i are sets of facts from $\mathcal{R}$, whose terms include free variables of q_i (which, in turn, include action parameters) and terms from $\mathtt{ADOM}(\mathcal{I}_0)$. Each A_i may include Skolem terms obtained by applying a function $f \in \mathcal{F}$ to any of the terms above. Skolem terms represent external service calls and model the values returned by the external environment when the action is executed.

The process specification ρ, is a finite set of *condition-action (CA) rules*. Each CA rule has the form $Q \mapsto \alpha$, where $\alpha \in \mathcal{A}$ and Q is a query over $\mathcal{R}$, constituting the precondition of the CA rule, whose free variables are the parameters of α (other terms can be quantified variables or constants in $\mathtt{ADOM}(\mathcal{I}_0)$). W.l.o.g., we can assume that there is a single CA rule for each action.

As regards the process execution, it essentially amounts to iterating over the following steps. First, an action α is chosen by the user and the corresponding CA rule in ρ is evaluated over the current database instance $\mathcal{I}$ (initially $\mathcal{I}_0$). This produces a (finite) set of complete *bindings* for α's parameters. Then, the user is asked to pick one of such bindings, say $\vec{p}$, so as to obtain a *ground action* $\alpha\vec{p}$. The next step is the action execution, which consists of applying all the action effects simultaneously. This requires *(i)* evaluating all queries $q_i\vec{p}$ (with partial assignment to their variables) associated with all effect specifications, *(ii)* binding the values occurring in the answers with the terms in all **del** D_i and all **add** A_i, and *(iii)* first deleting all the facts obtained in all D_i's, and subsequently adding those obtained in all A_i's, from and to the current database instance $\mathcal{I}$. In case some term t in some A_i involves a service call, the corresponding service is called with the appropriate inputs, to obtain the value to be assigned to t. We stress that all deletions take place at the same time, followed by all additions. Importantly, the final update (deletions and additions) is actually performed only if the resulting instance satisfies the constraints in $\mathcal{E}$ (otherwise a new iteration starts again on the current database instance, but the current binding is no longer provided as an option to the user). When the update is performed, a new instance $\mathcal{I}'$ is obtained, over which the process can iterate again. For a detailed description of the execution semantics, we refer the reader to [2].

3 The Electricity Consumption Process

In this work we rely on the example describing the processing of electricity consumption readouts in a house composed of three-room apartments. Each lodger has to choose an annual electricity saving plan for her apartment that is used as a threshold for the processing of the consumption data collected and stored in a database on a daily basis. These data can be processed given a selected period, and can later on be archived. This eventually leads to their elimination from the database.

The schema and constraints are listed in Figure 1. The schema is constituted by the following relations:

- $Apartment(id, plan)$ states that there is an apartment with a number id and a chosen saving plan $plan$. We reserve a special constant `null` to model the case when the apartment is not having a saving plan assigned.
- $Readings(id, ap\_id, day, amount, confirmed, processed)$ represents an $amount$ of electricity consumed by apartment ap_{id} on a certain day. Control state flags $confirmed$ and $processed$ are ranging over sets of predefined values $\{F_C, T_C\}$ and $\{F_P, T_P\}$, respectively. The first flag is used by an agent deciding on the correctness of the readings, while the second one is used by an agent checking the readings compliance with a saving plan assigned.
- $PRequest(p\_id, ap\_id, from, to)$ serves as a list of processing requests checking whether confirmed readings of an apartment with number ap_{id} conform to its saving plan within the selected $[from, to]$ period in days.
- $Day(day)$ is an auxiliary unary relation modeling a data type storing finite range of natural numbers from 1 to 365. With abuse of notation we define a $\leq$ relation such that for any two days d_1 and d_2: $d_1 \leq d_2 = \{d \in Day(d) \mid d_1 \leq_{\mathbb{N}} d \leq_{\mathbb{N}} d_2\}$, where $\leq_{\mathbb{N}}$ is the built-in comparison over $\mathbb{N}$.

The schema on Figure 1 is equipped with the following constraints:

- Every $Apartment$ is having one of the three predefined saving plans assigned:

$$\forall id, p.Apartment(id, p) \rightarrow (p = \texttt{No} \vee p = \texttt{Moderate} \vee p = \texttt{Strong})$$

  Similarly, for control state flags of $Readings$ we have:

$$\forall id, ap, d, a, c, p.Readings(id, ap, d, a, c, p) \rightarrow c = \texttt{F}_\texttt{C} \vee c = \texttt{T}_\texttt{C}$$
$$\forall id, ap, d, a, c, p.Readings(id, ap, d, a, c, p) \rightarrow p = \texttt{F}_\texttt{P} \vee p = \texttt{T}_\texttt{P}$$

- The first columns of $Apartment$, $Readings$ and $PRequest$ are the (primary) keys of corresponding relations:

$$\forall id, p, p'.Apartment(id, p) \wedge Apartment(id, p') \rightarrow p = p'$$
$$\forall id, ap, ap', d, d', a, a', c, c', p, p'.Readings(id, ap, d, a, c, p) \wedge$$
$$\quad Readings(id, ap', d', a', c', p') \rightarrow ap = ap' \wedge d = d' \wedge a = a' \wedge c = c' \wedge p = p'$$
$$\forall id, ap, ap', f, f', t, t'.PRequest(id, ap, f, t) \wedge PRequest(id, ap', f', t')$$
$$\quad \rightarrow ap = ap' \wedge f = f' \wedge t = t'$$

[1] The active domain $\texttt{ADOM}(\mathcal{I})$ of a DB instance $\mathcal{I}$ is the subset of elements of $\mathcal{C}$ occurring in $\mathcal{I}$.

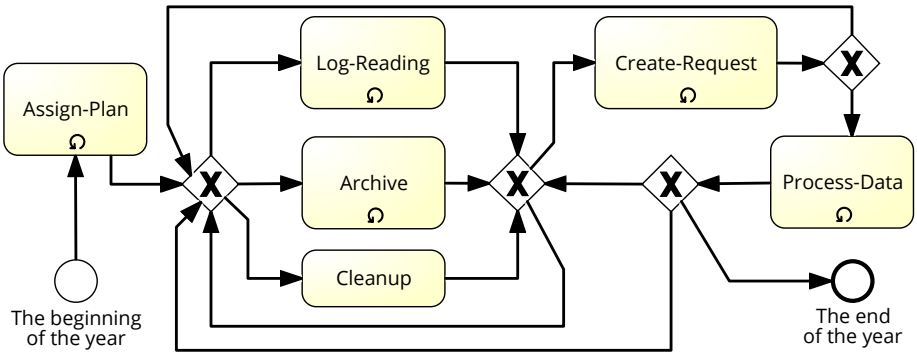

Fig. 2. BPMN diagram sketching how a house flows through the process

By analogy, one can formulate unique index constraints for $Readings$ and $PRequest$:

$$\forall ap, d, id, id', a, a', c, c', p, p'.Readings(id, ap, d, a, c, p) \wedge \\ Readings(id', ap, d, a', c', p') \rightarrow id = id' \wedge a = a' \wedge c = c' \wedge p = p'$$
$$\forall ap, f, t, id, id'.PRequest(id, ap, f, t) \wedge PRequest(id', ap, f, t) \rightarrow id = id'$$

– The foreign key constraints are formalized as:

$$\forall id, ap, d, a, c, p.Readings(id, ap, d, a, c, p) \rightarrow \exists pl.Apartment(id, pl)$$
$$\forall id, ap, f, t.PRequest(id, ap, f, t) \rightarrow \exists pl.Apartment(id, pl)$$

– The domain constraint demands that, if a chosen plan is either `Moderate` or `Strong`, then the relevant electricity consumption threshold should be respected:

$$\forall id, ap, d, a.Readings(id, ap, d, a, \mathtt{T_C}, \mathtt{T_P}) \wedge Apartment(ap, \mathtt{Moderate}) \rightarrow a \leq 11$$
$$\forall id, ap, d, a.Readings(id, ap, d, a, \mathtt{T_C}, \mathtt{T_P}) \wedge Apartment(ap, \mathtt{Strong}) \rightarrow a \leq 8$$

Finally, we assume that at the beginning of the year the list of all the apartments without defined plans (i.e., `null`) is known.

Figure 2 shows a BPMN diagram sketching the electrical readings processing mechanism, where each task corresponds to a certain DCDS action manipulating the data layer of Figure 2. Given our initial assumption on the apartment plans, one should first define a plan type by calling an action ASSIGN-PLAN for each $Apartment$ with `null` in the corresponding attribute. The plan type is assigned to the apartment by calling the $newPlan$ service.

$$Apartment(id, \mathtt{null}) \mapsto \textsc{Assign-Plan}(id)$$

$$\textsc{Assign-Plan}(\mathsf{id}) : \\ \{\mathsf{true} \rightsquigarrow \mathbf{del}\{Apartment(\mathsf{id}, \mathtt{null})\}\ \mathbf{add}\{Apartment(\mathsf{id}, newPlan(\mathsf{id}))\}\}$$

One electrical reading from the apartment is registered in the system for a given day by calling the $readout$ service. The correctness of the reading is obtained from service

confirm, which models a user form asking an external agent to provide a confirmation degree value. The reading entry identifier is generated using the $newID$ service.

$$\neg Readings(id, ap, d, a, c, p) \wedge \\ Day(d) \wedge Apartment(ap, pl) \wedge pl \neq \texttt{null} \mapsto \textsc{Log-Reading}(ap, d)$$

$$\textsc{Log-Reading}(\mathsf{ap}, \mathsf{d}) : \\ \{\mathsf{true} \rightsquigarrow \mathbf{add}\{Reading(newId(), \mathsf{ap}, \mathsf{d}, readout(\mathsf{ap}, \mathsf{d}), confirm(), \mathtt{F_P})\}\}$$

At any stage of the system execution, all the readings which have not been confirmed by the agent can be removed from the system using the CLEANUP action.

$$Apartment(id, p) \wedge p \neq \texttt{null} \mapsto \textsc{Cleanup}()$$

$$\textsc{Cleanup}() : \\ \{\forall id, c \exists ap, d, p.Reading(id, ap, d, a, c, p) \rightsquigarrow \mathbf{del}\{Reading(id, ap, d, a, c, p)\}\}$$

Each apartment's readings can be processed so that to check their compliance with a saving plan assigned. However, in order to do so the processing request has to be created. An external agent responsible for the creation of the request is asked to specify a processing period using two service calls: $getFromDate$ and $getToDate$.

$$Reading(id, ap, d, a, \mathtt{T_C}, p) \mapsto \textsc{Create-Request}(ap)$$

$$\textsc{Create-Request}(\mathsf{ap}) : \\ \{\mathsf{true} \rightsquigarrow \mathbf{add}\{PRequest(newID(), \mathsf{ap}, getFromDate(\mathsf{ap}), getToDate(\mathsf{ap}))\}\}$$

As soon as the processing request has been created, the readings during the specified period can be processed. Note that every processed entry should conform to the consumption threshold imposed by the chosen plan.

$$PReqest(id, ap, from, to) \wedge \\ Day(from) \wedge Day(to) \wedge Apartments(ap, p) \mapsto \textsc{Process-Data}(id, ap, from, to)$$

$$\textsc{Process-Data}(\mathsf{id}, \mathsf{ap}, \mathsf{from}, \mathsf{to}) : \\ \left\{ \begin{array}{l} \forall d \exists r\_id, a.Reading(r\_id, \mathsf{ap}, d, a, \mathtt{T_C}, \mathtt{F_P}) \wedge \mathsf{d} \geq \mathsf{from} \wedge \mathsf{d} \leq \mathsf{to} \rightsquigarrow \\ \quad \mathbf{del}\{PReqest(\mathsf{id}, \mathsf{ap}, \mathsf{from}, \mathsf{to}), Reading(r\_id, \mathsf{ap}, d, a, \mathtt{T_C}, \mathtt{F_P}\} \\ \quad \mathbf{add}\{Reading(r\_id, \mathsf{ap}, d, a, \mathtt{T_C}, \mathtt{T_P}\} \end{array} \right\}$$

One can try to archive all the processed readings which are not going to be exploited anymore in the system by calling the ARCHIVE action.

$$Apartment(ap, p) \wedge Day(from) \wedge Day(to) \wedge p \neq \texttt{null} \mapsto \textsc{Archive}(ap, from, to)$$

$$\textsc{Archive}(\mathsf{ap}, \mathsf{from}, \mathsf{to}) : \\ \left\{ \begin{array}{l} \forall id, d \exists a.Reading(id, \mathsf{ap}, d, a, \mathtt{T_C}, \mathtt{T_P}) \wedge d \geq \mathsf{from} \wedge d \leq \mathsf{to} \rightsquigarrow \\ \quad \mathbf{del}\{Reading(id, \mathsf{ap}, d, a, \mathtt{T_C}, \mathtt{T_P})\} \end{array} \right\}$$

In the next section, we describe an actual implementation of the system based on RDBMS technology.

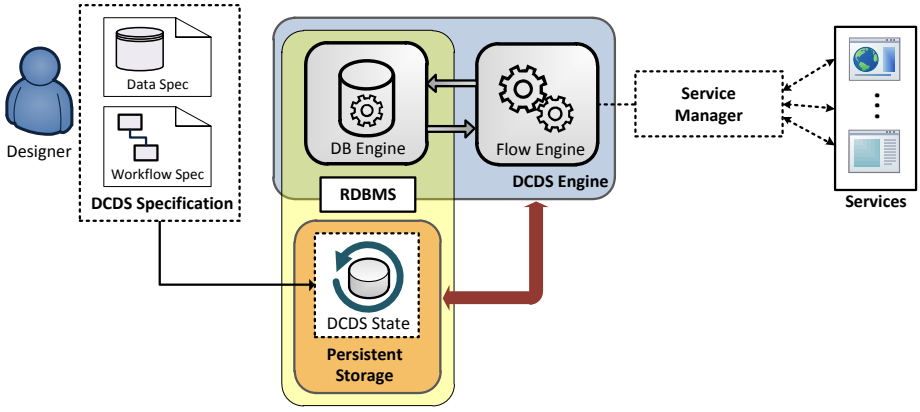

Fig. 3. Architecture of the DCDS implementation

4 Specifying and Implementing DCDSs in a RDBMS

A DCDS specification is maintained by the RDBMS, which interacts with the *Flow Engine* implemented in Java. The Flow Engine executes calls to the RDBMS, and handles the interaction with external services through a *Service Manager*.

Specifically, the RDBMSs maintains: *(i)* a *data layer specification* consisting of relational tables, equipped with functional dependencies and additional domain-dependent constraints, and *(ii)* a *process layer specification*, consisting of action metadata in the form of a relational table containing the action names together with their parameters. Moreover, the DBMS stores sets of prepared statements and of (parameterized) stored procedures. Each prepared statement is an SQL query that corresponds to *(i)* a query in the precondition of a CA rule, or *(ii)* a query in an action effect. Each stored procedure is associated to an action, and takes care of deleting and adding the facts in all its effects. The parameters of the procedure represent those action parameters that are used in the delete and add lists of the action effects.

At each moment in time, the DBMS stores the DCDS snapshot, which is subject to the data layer specification. The snapshot is initialized to the initial state of the DCDS, and then manipulated by the Flow Engine. We illustrate now the operation of the Flow Engine, which initializes the DCDS execution by querying the DBMS about the available actions, and then repeatedly calls an action by coordinating the activation of prepared statements and stored procedures according to the action execution cycle, while interacting with external services to acquire the service call results. Specifically, with reference to Figure 4, an *action execution cycle* is carried out as follows: *(1)* The cycle starts with the user choosing one of the available actions presented by the Flow Engine. *(2)* The Flow Engine evaluates the CA-rule associated to the chosen action α by calling the corresponding prepared statement over the current DCDS snapshot, and stores the returned possible parameter assignments for the action in a temporary table T. All parameters assignments in T are initially unmarked, meaning that they are available for the user to choose. *(3)* If unmarked parameters are present in T, the user is asked to

choose one of those, which is marked as unavailable, and the Flow Engine proceeds with evaluating α instantiated with the chosen parameters. *(4)* To do so, first the queries in all the effects of α are executed, which provides values to instantiate both the arguments of the service calls (stored in *service calls tables*, one for each service), and the facts to delete and add from the current snapshot. *(5)* The service calls are executed by calling the *service manager*, and the returned results provide the missing instantiations for the facts to add. *(6)* A transaction is started to perform the delete and add operations. *(7)* If the DCDS constraints are satisfied, the transaction is committed, and the next iteration of the action execution cycle is started. Otherwise, if the constraints are not satisfied, the transaction is aborted so that the DCDS snapshot stays unmodified, and the user is asked to choose a different parameter from the ones still available in T. *(8)* If no unmarked parameters are available, the user is asked to choose another action.

Figure 5 shows a simple run of the DCDS representing electrical readings handling process. The run starts with a table containing data of two apartments. The user chooses the only executable action (ASSIGN-PLAN), the Flow Engine evaluates its CA-rule query and asks the user to select one of the apartment id's. The latter one chooses $id = 2$ as the ASSIGN-PLAN call parameter and executes the action. Due to the fact that the query in ASSIGN-PLAN is trivial, the Flow Engine only needs to generate a service call table consisting of a single entry $newPlan(2)$ instantiated with `Moderate` - a value that is returned after $newPlan(2)$ has been executed by the *service manager*. The delete and add lists of the considered action are performing a simple update over the $Apartment$ table, yielding with a new database instance where the saving plan for apartment 2 has been updated to `Moderate`. By analogy with the ASSIGN-PLAN execution step, one can thoroughly follow up all the executions up to CREATE-REQUEST (inclusive) as it is shown in Figure 5. In the case of the PROCESS-DATA action, as soon as the corresponding CA-rule query has been evaluated and the action parameters have been instantiated, the Flow Engine proceeds with a non-trivial effect query which result T should consist of all the readings with a day lying in the interval between from and to. The derived result is used in the delete and add operations which hold true (i.e., the $processed$ value updates to $\mathtt{T_P}$) if and only if the $amount$ in every tuple of T does not exceed the threshold defined in the DCDS constraints.

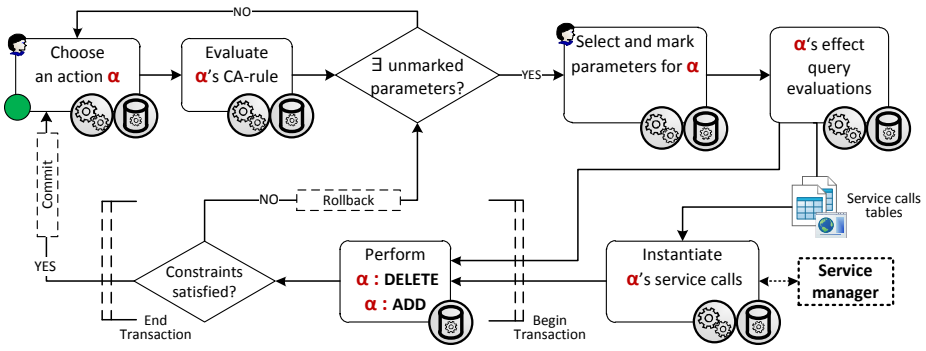

Fig. 4. The action execution cycle

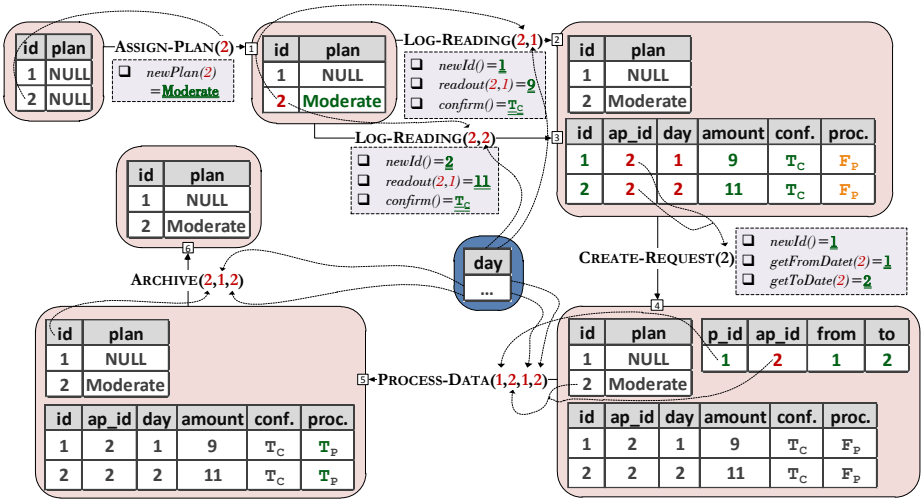

Fig. 5. A simple run of the DCDS example in Section 2

5 Towards a Verification System for DCDSs

Our ultimate goal is to implement a model checker for DCDSs that allows us to verify a DCDS against some property of interest, leveraging as much as possible the benefits and maturity of the relational technology. The formal verification setting, in particular the logic, has been devised in [2], to which we refer the reader for additional details. Here, we summarize the central notions of the framework and briefly discuss how we intend to proceed to move from an abstract setting to an actual implemented system.

The logic used to specify the properties of interest is a first-order extension of the μ-calculus [11], with an active-domain semantics and guarded quantification. The logic is called $\mu\mathcal{L}_p$ and its formulas Φ are inductively defined as follows:

$$\Phi ::= \varphi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.L(x) \wedge \Phi \mid L(\vec{x}) \wedge \langle\!-\!\rangle\Phi \mid L(\vec{x}) \wedge [\!-\!]\Phi \mid Z \mid \mu Z.\Phi,$$

where:

- φ is a (possibly open) FO formula over the relational vocabulary of the DCDS;
- $L(x)$ holds true at a DCDS state if x occurs in the corresponding database;
- $\langle\!-\!\rangle$ and $[\!-\!]$ are two modal operators, respectively standing for "there exists a successor such that" and "all successors are such that";
- Z and μ are the standard 0-ary second-order variable and the least-fixpoint operator of μ-calculus.

We do not get into further details of the logic but provide some intuitive examples below. Notice that quantification is guarded by the predicate L, which essentially implies that only the values that persist in the active domain along transitions are of interest.

Example 1. We consider two properties of interest in the electricity consumption process of Section 3. A property of interest is that whenever a processing request is created for a

given selected period in days, it will be eventually used by an agent in order to check and update the logged readings within that period. This can be expressed as follows:

$$\nu X.\big(\forall d_1, d_2, id, ap.PRequest(id, ap, d_1, d_2) \rightarrow \\ (\mu Y.(\forall id', d, a.Reading(id', ap, d, a, \mathtt{T_C}, \mathtt{T_P}) \wedge d_1 \leq d \wedge d \leq d_2) \\ \vee [\!-\!](PRequest(id, ap, d_1, d_2) \wedge Y)\big) \wedge [\!-\!]X$$

Another property of interest is

$$\nu X.\neg\big(\exists id, ap, d, a, c, p.Reading(id, ap, d, a, c, p) \wedge Apartment(ap, \mathtt{null})\big) \wedge [\!-\!]X$$

It is a safety property ensuring that it is not possible to register readings for an apartment that is not associated to any saving plan. ■

The semantics of $\mu\mathcal{L}_p$ formulas is defined in terms of a transition system (TS) $\Upsilon_{\mathcal{S}}$ capturing the execution semantics of a DCDS [2]. Intuitively, such a TS models all the executions of the process layer on the data layer, starting from the initial state. The states of $\Upsilon_{\mathcal{S}}$ represent the possible snapshots of the DCDS, i.e., database instances, while the transitions capture the execution of actions. Since not needed to evaluate the logic, transitions carry no information about the corresponding action. Notice that, in general, $\Upsilon_{\mathcal{S}}$ is infinite-state. This is a consequence of the fact that a DCDS can have infinitely many snapshots, as interpreted over infinite domains.

The state-infiniteness of $\Upsilon_{\mathcal{S}}$ precludes the possibility of resorting to (adaptations of) standard model checking techniques. In fact, it can be shown that, in the general case, verification of $\mu\mathcal{L}_p$ formulas is undecidable [2]. Nonetheless, a notable class of DCDSs for which verification is decidable has been isolated, namely that of *state-bounded* systems [2,3]. A DCDS is said to be state-bounded if all of its snapshots contain only a bounded number of distinct individuals. Notice that even under this assumption, a DCDS can be infinite-state, as the snapshots contain values taken from an infinite interpretation domain. Interestingly, for such systems one can derive a *faithful abstraction* in the form of a *finite-state* TS that can thus be used to perform the verification. More precisely, *for every $\mu\mathcal{L}_p$ formula Φ, the formula is satisfied by the original system iff it is satisfied by its finite abstraction*.

There exists an easy way to build the abstraction, which essentially amounts to apply the procedure sketched above for $\Upsilon_{\mathcal{S}}$, but using a *finite* interpretation domain of appropriate cardinality [2]. Interestingly, once it is known that the DCDS is state-bounded, this cardinality needs not to be given as input, but can be implicitly discovered by the RCYCL algorithm in [2]. The algorithm is based on two observations:

1. When a service is issued, it is not really important to consider the returned value, but only how the result relates to the other returned values, and the values currently present in the database.
2. The $\mu\mathcal{L}_p$ logic is not able to distinguish whether the result returned by a service call is *globally fresh* (i.e., it has never been seen in the past), or only *locally fresh* (i.e., it is different from all values currently present in the database).

The algorithm exploits observation (1) by considering, for each service call to be issued, only a bounded number of representative results. It then exploits observation (2) by *recycling*, along a run, previously encountered values that are locally fresh, instead

of considering globally fresh ones. The transition system $\Lambda_{\mathcal{S}}$ so obtained enjoys the following two key properties:

- $\Lambda_{\mathcal{S}}$ is a faithful abstraction of $\Upsilon_{\mathcal{S}}$, for all $\mu\mathcal{L}_p$ properties;
- when $\mathcal{S}$ is state-bounded, $\Lambda_{\mathcal{S}}$ is finite-state.

This algorithm is apt to be directly implemented on top of the architecture discussed in Section 4, by just changing the behavior of the service manager (cf. Figure 3) according to the description above. In particular, recycling can be efficiently realized by employing an ordered domain for freshly introduced values, taking the minimum value that is locally fresh whenever a fresh service call result must be considered.

By considering that $\Lambda_{\mathcal{S}}$ is defined on a finite interpretation domain, instead of evaluating the original formula Φ, one can equivalently use a "propositionalized" version of it, simply obtained by quantifier elimination. The resulting problem ends up being a standard model checking problem for which existing techniques and tools can be applied.

Our next step will consist in extending our prototype system with the capability to build the finite abstraction $\Lambda_{\mathcal{S}}$, maintaining it in the DBMS, and exploiting state-of-the-art model checking techniques to perform verification.

6 Conclusion and Future Work

In this paper we have presented an implementation of Data-Centric Dynamic Systems based on the use of a Relational Database Management System, exploited to handle *all* the aspects of a DCDS, including specification and execution. We have shown the implementation at work on a running example and have provided some details about our next steps. In particular, we have discussed how we intend to extend our system with verification functionalities. In this respect, we add that while the verification framework can deal with very general specifications expressible in a FO variant of the μ-calculus, we plan to first deal with more specific classes of formulas, obtained as FO variants of CTL and LTL. On the one hand, this calls for a careful analysis on how to compactly represent, inside the RDBMS, the relational state (or set of states) needed by the model checking algorithm at each step. On the other hand, a plethora of model checking tools are available for the verification of propositional CTL/LTL, such as NuXMV [9]. We plan to explore an alternative way of model checking DCDSs, by plugging them in on top of the architecture presented in this paper. In this respect, our ultimate goal is to integrate, with a minimal effort, state-of-the-art techniques developed in formal verification towards the model checking of DCDSs.

From the modeling point of view, the effort described in this paper constitutes the specification and execution core towards more sophisticated approaches dealing on the one hand with distributed control [17,6], and on the other hand with incomplete information and inconsistency management [7,8].

Acknowledgments. This research has been partially supported by: the EU IP project Optique (*Scalable End-user Access to Big Data*), grant agreement n. FP7-318338; the UNIBZ internal project KENDO (*Knowledge-driven ENterprise Distributed cOmputing*); and Ripartizione Diritto allo Studio, Università e Ricerca Scientifica of Provincia Autonoma di Bolzano–Alto Adige, under project VeriSynCoPateD (*Verification and Synthesis from Components of Processes that Manipulate Data*).

References

1. Abiteboul, S., Vianu, V., Fordham, B., Yesha, Y.: Relational transducers for electronic commerce. J. of Computer and System Sciences 61(2), 236–269 (2000)
2. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proc. of the 32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2013) (2013)
3. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of agent-based artifact systems. J. of Artificial Intelligence Research 51, 333–376 (2014)
4. Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Proc. of the 5th Int. Conference on Business Process Management (BPM 2007) (2007)
5. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: A database theory perspective. In: Proc. of the 32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2013) (2013)
6. Calvanese, D., Delzanno, G., Montali, M.: Verification of relational multiagent systems with data types. In: Proc. of the 29th AAAI Conf. on Artificial Intelligence (AAAI 2015). AAAI Press (2015)
7. Calvanese, D., Kharlamov, E., Montali, M., Santoso, A., Zheleznyakov, D.: Verification of inconsistency-aware knowledge and action bases. In: Proc. of the 23rd Int. Joint Conference on Artificial Intelligence (IJCAI 2013). AAAI Press (2013)
8. Calvanese, D., Montali, M., Santoso, A.: Verification of generalized inconsistency-aware knowledge and action bases. In: Proc. of the 24th Int. Joint Conference on Artificial Intelligence (IJCAI 2015). AAAI Press (2015)
9. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Proc. of the 26th Int. Conf. on Computer Aided Verification (CAV 2014). LNCS, vol. 8559, pp. 334–342. Springer (2014)
10. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web applications. J. Comput. Syst. Sci. 73(3), 442–474 (2007)
11. Emerson, E.A.: Model checking and the Mu-calculus. In: Immerman, N., Kolaitis, P. (eds.) DIMACS Symposium on Descriptive Complexity and Finite Model. American Mathematical Society Press (1997)
12. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: Proc. of the 7th Int. Conf. on Ontologies, DataBases, and Applications of Semantics (ODBASE 2008) (2008)
13. Karel, R., Richardson, C., Moore, C.: Warning: Don't assume your business processes use master data – Synchronize your business process and master data strategies. Report, Forrester (Sep 2009)
14. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: Fundamental requirements and their support in existing approaches. Int. J. of Information System Modeling and Design 2(2), 19–46 (2011)
15. Martin, D.L., Burstein, M.H., McDermott, D.V., McIlraith, S.A., Paolucci, M., Sycara, K.P., McGuinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with OWL-S. In: Proc. of the 16th Int. World Wide Web Conf. (WWW 2007) (2007)
16. Meyer, A., Smirnov, S., Weske, M.: Data in business processes. Tech. Rep. 50, Hasso-Plattner-Institut for IT Systems Engineering, Universität Potsdam (2011)
17. Montali, M., Calvanese, D., De Giacomo, G.: Verification of data-aware commitment-based multiagent system. In: Proc. of the Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS '14). pp. 157–164. IFAAMAS (2014)
18. Reichert, M.: Process and data: Two sides of the same coin? In: Proc. of OTM (2012)