Towards a Reference Implementation for Data Centric Dynamic Systems

Alessandro Russo[1](✉), Massimo Mecella[1], Marco Montali[2], and Fabio Patrizi[1]

[1] DIAG, Sapienza Università di Roma, Rome, Italy
{arusso,mecella,patrizi}@dis.uniroma1.it

[2] KRDB Research Centre, Free University of Bozen-Bolzano, Bolzano, Italy
montali@inf.unibz.it

Abstract. Data- and artifact-centric business processes are gaining momentum due to their ability of explicitly capturing the interplay between the process control-flow and the manipulated data. In this paper, we rely on the framework of Data-Centric Dynamic Systems (DCDSs), which has been recently introduced for the formal specification and verification of data-centric processes, showing how it can be lifted towards run-time execution support. In particular, we focus on the problem of database update as induced by the action execution, introducing a set of patterns that allow for an incremental management of the update. At the same time, we discuss the natural correspondence between DCDSs and state-of-the-art rule engines, e.g., JBoss Drools, which paves the way towards a reference implementation for data- and artifact-centric processes, where the model used for analysis and verification is fully aligned with the one adopted for the execution.

1 Introduction

Most of the current approaches to Business Process Management (BPM) adopt a procedural and imperative point-of-view, based on an explicit specification of the tasks to be performed and the execution relationships between them that define the overall flow of control. The modeling perspective is *activity-centric* and the main driver for run-time process progression is given by activity completions that enable subsequent tasks according to the control-flow. Languages such as BPMN and YAWL follow this imperative activity-centric paradigm and mainly focus on the control-flow perspective. Approaches aiming at producing *executable* process specifications should not only be limited to the control-flow perspective, but should also consider the *data perspective*, describing data elements consumed, produced and exchanged during process executions, and the *resource perspective*, describing the operational and organizational context for process execution in terms of resources (i.e., people, systems and services able to execute tasks) and their capabilities (i.e., any qualification, skill, equipment, property etc. relevant for task assignment and execution), along with the policies and rules used to assign tasks to resources for execution. Declarative *constraint-based* approaches,

N. Lohmann et al. (Eds.): BPM 2013 Workshops, LNBIP 171, pp. 141–154, 2014.
DOI: 10.1007/978-3-319-06257-0_12, © Springer International Publishing Switzerland 2014

such as Declare [9], for modeling, enacting and monitoring processes are an initial attempt to increase flexible modeling capabilities, through the specification of a (minimal) set of control-flow constraints to be satisfied (or not violated), defined as relationships among tasks that implicitly define possible execution alternatives by prohibiting undesired execution behaviors. Resulting models have no rigid control-flow structure, but they still focus on tasks/activities and provide limited support for data-oriented modeling and execution.

The root cause of many of the limitations of activity-centric approaches (based either on imperative procedural models or on declarative constraint-based specifications) is often identified in the lack of integration of processes and data [6]. In such models, the information perspective includes a set of data objects and the data flow between activities, along with the definition of which activities may read/write data elements as I/O parameters, but the information and data flows are hidden in the model [7]. To support the enactment of these models, activity-centric process-aware information systems basically distinguish between (i) application data, managed out of the scope of the process by application services invoked during activity executions; (ii) process-relevant data, represented as process variables that are read and updated by the activities and are used by the system to evaluate transitions and path choices (as routing conditions) within process instances; (iii) process control data, that define the current state of a process and its execution history. According to [4], this separation between process data/variables and external data sources leads to an "impedance mismatch" problem between the process layer and the data layer in a typical process-oriented information system. In addition, a recent work [8] has considered the role of data in twelve process modeling languages. The evaluation shows that the general level of data support is low: while in most of the cases the representation of data objects is supported, complex data relationships and their role in process modeling and execution are not considered.

To overcome the limitation of activity-centric approaches, *data-centric*, *object-aware* and *case management* approaches have recently emerged. The PHILharmonicFlows framework and prototype enables object-aware process management on the basis of a tight integration of processes, functions, data and users [6]. Process modeling and execution relies on two levels of granularity that cover object behavior (or life-cycle) and object interactions. The framework enables the definition of object types and object relations in a data model, while object behavior is expressed in terms of a process whose execution is driven by object attribute changes.

In data-centric methodologies, as the *business artifacts framework* [5], the data perspective is predominant and captures domain-relevant object types, their attributes, their possible states and life-cycles, and their interrelations, which together form a complex data structure or *information model*. This data model enables the identification and definition of the activities that rely on the object-related information and act on it, producing changes on attribute values, relations and object states. The general artifact-centric model does not restrict the way to specify artifact life-cycles, and constraints can be defined in terms

of: (i) abstract procedural process specifications, e.g., expressed as state machines or transition systems, as in SIENA [3]; (ii) logical/declarative formalisms (e.g., temporal or dynamic logics) or as a set of rules defined over the states of the artifacts, as in the Guard-Stage-Milestone (GSM) model [5] supported by the Barcelona GSM environment [13].

Such recent research efforts that focus on data-centric process management are often framed within the wider discussion that opposes BPM with *adaptive case* management (ACM) [12], a paradigm for supporting unstructured, unpredictable and unrepeatable business cases. released a first standard version of the Case Management Modeling Notation (CMMN)[1]; rather than an extension of BPMN, indeed CMMN relies on GSM constructs (guards, stages, milestones and sentries), with the additional possibility to unlink milestones from specific stages, define repetition strategies for stages and tasks, and enable late modeling/planning by introducing discretionary elements to be selected at run-time.

Several works have provided a theoretical foundation to the artifact-centric paradigm, with specific focus on the possibilities to perform verification tasks on the models. We refer the reader to [2] for a comprehensive discussion of the relevant literature. in [1], referred to as Relational Data-Centric Dynamic Systems (DCDSs), that considers both the case in which actions behave deterministically and the case in which they behave nondeterministically, so being still more realistic in modeling external inputs (either from human actors or services). Syntactic restrictions guaranteeing decidability of verification are shown for both cases. In [11] it is shown how to reduce a GSM schema to a DCDS schema. Thus DCDSs are capable of capturing concrete artifact-centric models (being GSM at the core of the CMMN standard) and it gives a procedure to analyze GSM schemas: verification of GSM schemas is, in general, undecidable, but once traduced in a DCDS it is possible to exploit the results in [1] for decidability of verification. A syntactic condition that is checkable directly on a GSM schema is presented and that, being subsumed by the conditions for DCDSs, guarantee decidability of verification.

From a practitioners' point of view, an important missing piece is the availability of process management systems enacting artifact-centric process models. In particular, a kind of reference/core implementation would be beneficial for rapid prototyping purposes, as well as for further research aiming at assessing their practical use, with the need of evaluating the related paradigms and methods in concrete settings. realization of such a reference implementation for DCDSs. Such an ambitious aim poses several challenges, which will be discussed in this paper as well, and has interesting outcomes, i.e., the seamless use of the specification model also as effective run-time of the process instances themselves. It is particularly interesting that the same model used for analysis and for verification is then used for the enactment, and this property is not guaranteed by other formalisms/approaches. Notably, the reduction from GSM to DCDSs [11] produces a DCDS model that resembles an execution engine based on forward rules, and requires to realize some "tricks" and supportive relationships that are

[1] http://www.omg.org/spec/CMMN/1.0/Beta1/

very similar to those ones that would serve precisely to manage the execution. We will discuss how to base a reference implementation on state-of-the-art rule engines, e.g., JBoss Drools.

2 Background and Basic Concepts

2.1 Data Centric Dynamic Systems

Data Centric Dynamic Systems (DCDSs) [1] are systems that fully capture the interplay between the data and the process component, providing an explicit account on how the actions belonging to the process manipulate the data. More specifically, a DCDS $\mathcal{S}$ is a pair $\langle \mathcal{D}, \mathcal{P} \rangle$ formed by two interacting layers: a *data layer* $\mathcal{D}$ and a *process layer* $\mathcal{P}$ over $\mathcal{D}$. Intuitively, the data layer keeps all the data of interest, while the process layer reads and evolves such data.

The data layer is constituted by a relational schema $\mathcal{R}$ equipped with (denial) constraints, and by an initial database instance $\mathcal{I}_0$ that conforms to the schema and satisfies the constraints. Constraints must be satisfied at each time point, and consequently it is forbidden to apply an action that would lead the data layer to a state that violates the constraints.

The process layer defines the progression mechanism for the DCDS. The main idea is that the current instance of the data layer can be arbitrarily queried, and consequently updated through action executions, possibly involving external service calls to get new values from the environment. More specifically, $\mathcal{P}$ is a triple $\langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where: $\mathcal{A}$ is a set of *actions*, which are the atomic update steps on the data layer; $\mathcal{F}$ are *external services* that can be called during the execution of actions; and ϱ is a set of condition-action rules that provide a declarative modeling of the *process*, and that are in particular used to determine which actions are executable at a given time.

Actions. Actions are used to evolve the current state of the data layer into a new state. To do so, they query the current state of the data layer, and use the answer, possibly together with further data obtained by invoking external service calls, to instantiate the data layer in the new state. Formally, an *action* $\alpha \in \mathcal{A}$ is an expression $\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$, where: (i) $\alpha(p_1, \ldots, p_n)$ is its *signature*, constituted by a name α and a sequence $p_1, \ldots, p_n$ of *parameters*, to be substituted with actual values when the action is invoked, and (ii) $\{e_1, \ldots, e_m\}$, denoted by $\text{EFFECT}(\alpha)$, is a set of *specifications of effects*, which are assumed to take place simultaneously. Each e_i has the form $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$, where:

- $q_i^+ \wedge Q_i^-$ is a query over $\mathcal{R}$ whose terms are variables, action parameters, and constants from $\text{ADOM}(\mathcal{I}_0)$[2], where q_i^+ is a union of conjunctive queries, and Q_i^- is an arbitrary first-order formula whose free variables are among those of q_i^+. Intuitively, q_i^+ is applied to extract the tuples used to instantiate the effect, and Q_i^- filters away some of such tuples.

[2] $\text{ADOM}(\mathcal{I}_0)$ is the set of constants/values mentioned in the initial database instance $\mathcal{I}_0$.

- E_i is the effect, i.e., a set of facts over $\mathcal{R}$, which includes as terms: terms in $\textsc{adom}(\mathcal{I}_0)$, free variables of q_i^+ and Q_i^- (including action parameters), and in addition Skolem terms formed by applying a function $f \in \mathcal{F}$ to one of the previous kinds of terms. Each such Skolem term f represent a call to an external service identified by f, and are typically meant to model the incorporation of values provided by an external user/environment when executing the action.

Process. The process is used to determine which actions can be executed at a given time, and with which parameters. To do so, it relies on condition-action rules, which constitute a flexible, declarative way of specifying the process, and can be used to accommodate more "concrete" process specification languages. condition-action rules of the form $Q \mapsto \alpha$, where α is an action in $\mathcal{A}$ and Q is a first-order query over $\mathcal{R}$ whose free variables are exactly the parameters of α, and whose other terms can be either quantified variables or constants in $\textsc{adom}(\mathcal{I}_0)$.

Example 1. In this work, we rely on the example presented in [1], where an audit system that manages the process of reimbursing travel expenses in a university is modeled as a DCDS. In particular, we report selected parts of the *request subsystem* that manages the submission of reimbursement requests by an employee. A reimbursement request is associated with the name of the employee (represented in the data layer as a relation $\mathsf{Travel} = \langle \mathsf{eName} \rangle$) and comprises information related to the corresponding flight and hotel costs ($\mathsf{Hotel} = \langle \mathsf{hName}, \mathsf{date}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD} \rangle$ and $\mathsf{Flight} = \langle \mathsf{date}, \mathsf{fNum}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD} \rangle$ relations). In addition, the data layer keeps the state of the request subsystem ($\mathsf{Status} = \langle \mathsf{status} \rangle$ relation, holding the fact $\mathsf{Status}(\text{'}\mathit{readyForRequest}\text{'})$ in the initial state), which take three different values: '*readyForRequest*', '*readyToVerify*', and '*readyToUpdate*', and a list of approved hotels ($\mathsf{ApprHotel} = \langle \mathsf{hName} \rangle$ relation).

The process layer includes a set of service calls, each modeling an input of an external value by the employee (e.g., $\textsc{inEName}()$ for the name of the employee, $\textsc{inHName}()$ for the hotel name, $\textsc{inHDate}()$ for the hotel arrival date, etc.). In particular, the $\textsc{decide}()$ service call models the decision of the human monitor, returning '*accepted*' if the request is accepted, and '*readyToUpdate*'if the request needs to be updated by the employee. The set of actions includes *InitiateRequest*, *VerifyRequest*, *UpdateRequest*, and *AcceptRequest*. When a request is initiated (action *InitiateRequest*), the system status is set to '*readyToVerify*'and the employee provides travel details (her name and hotel and flight details), as modeled by the subset of action effects

$$\mathsf{true} \rightsquigarrow \mathsf{Travel}(\textsc{inEName}())$$
$$\mathsf{true} \rightsquigarrow \mathsf{Hotel}(\textsc{inHName}(), \textsc{inHDate}(), \textsc{inHPrice}(), \textsc{inHCurrency}(), \textsc{inHPInUSD}())$$

Action *VerifyRequest* models the preliminary check by the monitor. Travel event, hotel, and flight information are copied unchanged to the next state. If the hotel is on the approved list, then the request is automatically accepted and the system

status is set accordingly. Otherwise, the request is handled by a human monitor (cf. DECIDE()). Action *VerifyRequest* includes as effects

$$\mathsf{Hotel}(x_1,\ldots,x_5) \wedge \mathsf{ApprHotel}(x_1) \rightsquigarrow \mathsf{Status}(`accepted')$$
$$\mathsf{Hotel}(x_1,\ldots,x_5) \wedge \neg\mathsf{ApprHotel}(x_1) \rightsquigarrow \mathsf{Status}(\textsc{decide}())$$
$$\mathsf{Travel}(n) \rightsquigarrow \mathsf{Travel}(n), \mathsf{Hotel}(x_1,\ldots,x_5) \rightsquigarrow \mathsf{Hotel}(x_1,\ldots,x_5)$$
$$\mathsf{Flight}(x_1,\ldots,x_5) \rightsquigarrow \mathsf{Flight}(x_1,\ldots,x_5), \mathsf{ApprHotel}(x) \rightsquigarrow \mathsf{ApprHotel}(x)$$

In case of rejection,the action *UpdateRequest* is triggered and the employee needs to modify the information regarding hotel and flight, moving the status to '*readyToVerify*'. Finally, action *AcceptRequest* returns the system in the state '*readyForRequest*'. The overall process is defined by condition-action rules that guard the actions by the current system's state and include (among the others): $\mathsf{Status}(`readyToVerify') \mapsto$ *VerifyRequest*, $\mathsf{Status}(`accepted') \mapsto$ *AcceptRequest*.

2.2 Process and Action Execution

To understand the potential of DCDS models as key enablers towards a model-driven process execution and management approach, we define an abstract execution semantics for condition-action rules and actions that determines the actual behavior of an abstract execution engine for DCDSs. Basically, given an instance $\mathcal{I}$ of the data layer and a process specification ϱ, the engine undertakes a set of steps that lead to instantiate the data layer in a new state. The approach is in accordance with the formal execution semantics defined in[1].

Rules Evaluation and Executable Actions. For each CA rule $Q \mapsto \alpha$ the corresponding query Q is executed over the data layer. Whenever a tuple $\vec{d}$ of values is returned by issuing Q over the current database instance, then the condition-action rule states that α is executable by fixing its parameters according to $\vec{d}$. Basically, the eligibility of a rule corresponds to the executability of the corresponding action, under one or more bindings for its parameters. In general, at a given time multiple actions are executable, and the same action can be parametrized in several ways. Notice that this approach provides a notion of concurrency tailored to the one of interleaving, as typically done in formal verification.

Action Execution. Among the executable actions, a strategy has to be implemented to select which action to pick. As pointed out in [10], many possible strategies can be implemented on top of a process-aware information system to allocate actions to resources. These strategies are orthogonal to the execution semantics, and can be therefore seamlessly realized on top of the abstract execution engine described here.

When an action α with parameters σ is chosen, the engine is responsible for the application of the action. In particular, the execution of α instantiated with σ corresponds to evaluating and applying the corresponding effects, according to the following steps:

1. The effects of α are partially instantiated using the parameter assignment σ.

2. The left-hand side of each effect is evaluated by posing the corresponding query over the current relational instance, obtaining back a result set that consists of all possible assignments $\theta_1, \ldots, \theta_n$ that satisfy the query.
3. The right-hand side E_i is considered, so as to obtain, for each θ_i, the set of facts instantiated with σ and θ_i, denoted as $E_i\sigma\theta_i$.
4. $E_i\sigma\theta_i$ may contain service calls. In this case, the engine handles the interaction with such services, so as to obtain the result values for each call[3]. Notice that how these values are obtained is orthogonal to the abstract execution semantics, and could be managed by the execution engine in several different ways, such as interaction with external web services, or with human stakeholders via forms.
5. The new instance of the data layer is obtained by putting together the results obtained from the application of effects and the incorporation of service call results.

3 Basic Effect Patterns for a Reference Implementation

The DCDS semantics described above intuitively defines the execution of an action as a two-step process consisting in: *(i)* the generation of all the facts implied by an action's effects, and *(ii)* the construction of the successor data-layer instance, containing exactly such facts. It is not hard to see that a direct, naïve implementation of this semantics yields, in general, a waste of computational resources. For instance, for an action with effect specifications $e_1 : R(\vec{x}) \rightsquigarrow R(\vec{x})$ and $e_2 : q^+(\vec{x}) \rightsquigarrow R(\vec{x})$, the construction of the successor instance requires to evaluate both $R(\vec{x})$ and $q^+(\vec{x})$ against the current instance $\mathcal{I}$, and then define the successor instance $\mathcal{I}'$ as the union of the obtained facts. However, since the action only adds new facts – those generated by e_2, there is no need to generate the facts that persist from $\mathcal{I}$ – those generated by e_1. That is, according to the the common sense law of inertia, the transition can be efficiently realized in an incremental fashion.

What we do next, is to propose a modeling approach for DCDS that facilitates this incremental approach, by offering specific constructs for actions that only add, update or delete facts from the current instance. For more complex cases we still assume a direct implementation, and leave the analysis of these cases for future investigation.

From a practical perspective, supporting the specification and the execution of actions in terms of `create`, `delete` and `update` operations brings several advantages: 1. The way of specifying DCDS fits the usual attitude of designers and their familiarity with manipulating data through CRUD[4] operations. 2. CRUD operations enable automated techniques for a model- and data-driven generation of user forms for supporting the execution of actions that involve human

[3] We assume here two-way blocking service calls.

[4] Notice that the `read` operation is in fact already supported through queries over the data layer.

performers. 3. Action executions can be efficiently realized through incremental changes over the current instance.

We start by discussing the specification patterns that capture the effects corresponding to create, delete and update. Specifically, we isolate three syntactic patterns that can be used to define the usual create, delete and update operations, and for each of them, we provide an actual syntactic construct. We also introduce a further construct update$^+$, meant to capture a more general form of update where the new values to assign to existing facts can be obtained by answering generic queries over the current instance. This construct, though, needs an additional semantical requirement, in order to guarantee an incremental implementation of action effects. The intended goal of these constructs is to allow DCDS designers to produce action specifications whose effects can be incrementally applied on the current instance. All constructs are detailed below.

In order to be able to define generic DCDS effects, we also introduce the `set` construct, defined as:

$$\texttt{set}\ \vec{t}\ \texttt{for}\ R\ \texttt{if}\ q^+ \wedge Q^-,$$

where $\vec{t}$ is a tuple of terms that can be either action parameters, terms from $\text{ADOM}(\mathcal{I}_0)$, Skolem terms (as discussed in previous section), or free variables from $q^+ \wedge Q^-$. This construct corresponds to the effect specification $q^+ \wedge Q^- \rightsquigarrow R(\vec{t})$.

Next, we detail the constructs introduced above. To guarantee the possibility of applying incremental changes, we require that the effect specification of every action α is such that for every relation R, either: (i) only `insert`'s are present that add facts to R; or (ii) only one `delete` is present that deletes facts from R; or (iii) only one `update` is present that updates facts in R; or iv) only `sets` are present that set R. In addition, all relations R_i for which no effects are defined are preserved unchanged, i.e., we add the effect $R_i(\vec{x}) \rightsquigarrow R_i(\vec{x})$. Notice that the unrestricted use of `set`'s guarantees the full expressive power of DCDS, although it may negatively affect the efficiency of execution.

Adding Tuples to a Relation. A create (or ADD) operation that instantiates new facts to add to a relation R corresponds to the definition of the following effect specifications:

$$\text{ADD} : \{q^+ \wedge Q^- \rightsquigarrow R(\vec{t}), R(\vec{x}) \rightsquigarrow R(\vec{x})\}$$

The effect specification $q^+ \wedge Q^- \rightsquigarrow R(\vec{t})$ corresponds to the generation of a set of facts to be used to instantiate the relation R in the new state, and no specific restrictions are imposed with respect to the general effect specification form. Specifically, the terms $\vec{t}$ can include constants in $\text{ADOM}(\mathcal{I}_0)$, action parameters (recall that effect specifications occur within action's), free variables of q^+ and function calls that represent external service invocations. The effect specification $R(\vec{x}) \rightsquigarrow R(\vec{x})$ ensures the persistence of existing facts in the relation. The new instance of R can thus be incrementally built from R by adding the set of facts $R(\vec{t})$ obtained according to the general effect execution procedure. The ADD specification pattern is specified by the following construct:

$$\texttt{insert}\ \vec{t}\ \texttt{into}\ R\ \texttt{if}\ q^+ \wedge Q^-,$$

where $\vec{t}$ and $q^+ \wedge Q^-$ are as defined before. Notice that multiple ADD operations for the same relation R are allowed as effect specifications in an action α. The corresponding effect specifications are of the form:

$$q_1^+ \wedge Q_1^- \rightsquigarrow R(\vec{t}_1), \ldots, q_n^+ \wedge Q_n^- \rightsquigarrow R(\vec{t}_n), R(\vec{x}) \rightsquigarrow R(\vec{x})$$

where each effect specification of the form $q_i^+ \wedge Q_i^- \rightsquigarrow R(\vec{t}_i)$ corresponds to the generation of a set of facts to be added to the relation R , and the effect specification $R(\vec{x}) \rightsquigarrow R(\vec{x})$ ensures the persistence of existing facts in the relation.

Deleting/Retaining Tuples From a Relation. Intuitively, deleting from a relation R a set of tuples that match a condition, requires to copy to the next instance (i.e., retain) all the existing tuples in the relation that do not match the deletion condition. To achieve this, we can exploit the explicit distinction made in DCDS effects between q^+ and Q^-, by defining an effect such that: (i) q^+ selects all the tuples in the relation; (ii) Q^- filters away the tuples to be deleted; (iii) the resulting tuples are used to instantiate the new set of facts for the relation.

A `delete` operation that removes a set of tuples from a relation R can be represented as a DCDS effect of the form

$$\textsc{Retain}: \quad R(\vec{x}) \wedge Q^- \rightsquigarrow R(\vec{x})$$

The effect retains all the tuples in R that satisfy Q^-, or, equivalently, deletes all those that do not satisfy Q^-. The new instance of R can thus be incrementally built from R by deleting all the tuples that satisfy $R(\vec{x}) \wedge \neg(Q^-)$. Notice that the specification restricts the terms $\vec{x}$ to be only variables, i.e., constants in ADOM($\mathcal{I}_0$), action parameters and, for the right-hand side of the effect specification, function calls are not allowed, although constants and action parameters can still be used as terms in Q^-, according to the general effect specification form. The `delete` construct is defined as:

$$\texttt{delete from } R \texttt{ where } Q^-,$$

where Q^- has to fulfill the requirements discussed above. Notice that this corresponds to the effect specification: $R(\vec{x}) \wedge \neg Q^- \rightsquigarrow R(\vec{x})$, which is in the same form as the RETAIN pattern above. As already mentioned, only one DELETE operation for R can be defined in the set of effect specifications, for an action α. Notice, however, that multiple retain/delete conditions over the tuples of R can alway be properly combined in the Q^- part of a single DELETE operation.

Updating Tuples in a Relation. An `update` operation corresponds to the following effect specifications:

$$\textsc{Update}: \{R(\vec{x}) \wedge Q^- \rightsquigarrow R(\vec{t}), R(\vec{x}) \wedge \neg(Q^-) \rightsquigarrow R(\vec{x})$$

The effect specification $R(\vec{x}) \wedge Q^- \rightsquigarrow R(\vec{t})$ selects from R all the tuples that match the update condition Q^-, and for each of them a tuple $\vec{t}$ for R is instantiated. As in the general case, constants in ADOM($\mathcal{I}_0$), action parameters, free variables of q^+ (i.e., $R(\vec{x})$) and function calls are allowed in the right-hand side of the effect specification. The effect specification $R(\vec{x}) \wedge \neg(Q^-) \rightsquigarrow R(\vec{x})$ selects from

$R(\vec{x})$ all the tuples that do not match the update condition Q^-, and copies them unchanged to the next state, so as to ensure their persistence. The new instance of R can thus be incrementally built from R by updating each tuple that matches the update condition. The `update` construct we provide is defined as:

$$\texttt{update}\ R\ \texttt{set}\ \vec{t}\ \texttt{where}\ Q^-,$$

where $\vec{t}$ and Q^- are as discussed above. Analogously to the DELETE operation, we constrain the specification so that only one UPDATE operation for R can occur in EFFECT(α) for action α. Multiple update conditions over the tuples of R can still be combined in the Q^- part of a single UPDATE.

Updating Tuples in a Relation – Update$^+$. In the UPDATE operation defined above, the updated tuples are generated using constants in ADOM($\mathcal{I}_0$), action parameters, function calls and free variables of the q^+ part (i.e., $R(\vec{x})$). We provide here an extended version of the operation, called UPDATE$^+$, that under specific conditions allows updating tuples in a relation R also with values obtained by querying the current instance of the data layer. The UPDATE$^+$ operator is defined by the following effect specifications:

$$\text{UPDATE}^+ : \{q^+ \wedge Q^- \rightsquigarrow R(\vec{t}), R(\vec{x}) \wedge \widetilde{Q}^- \rightsquigarrow R(\vec{x}), \text{ where}$$
$$q^+(\vec{x},\vec{y}) = R(\vec{x}) \wedge q(\vec{z},\vec{y}), \text{ with } \vec{z} \subseteq \vec{x}, \text{ and } \widetilde{Q}^- = \neg\exists\vec{y}\ (q(\vec{z},\vec{y}) \vee Q^-(\vec{x},\vec{y})).$$

Intuitively, the answer of $Q(\vec{x},\vec{y}) = q^+(\vec{x},\vec{y}) \wedge Q^-(\vec{x},\vec{y})$ produces a result set where each tuple consists of two sub-tuples: one, the projection over $\vec{x}$, is the R-tuple to update, and one, the projection over $\vec{y}$, contains the values needed to instantiate the updated fact $R(\vec{t})$. Notice that $\vec{t}$ can include free variables of q^+ (i.e., variables from $\vec{x}$ and $\vec{y}$), as well as constants in ADOM($\mathcal{I}_0$), action parameters and function calls. The effect specification $R(\vec{x}) \wedge \widetilde{Q}^- \rightsquigarrow R(\vec{x})$ ensures persistence of the tuples in R not subject to updates. Indeed, it is easy to check that its answer contains all the R-tuples not occurring as sub-tuples in the answer of $Q(\vec{x},\vec{y})$.

In this case, the new instance of R can be incrementally built from R by updating a subset of its tuples only if the answer given by evaluating $Q(\vec{x},\vec{y}) = q^+(\vec{x},\vec{y}) \wedge Q^-(\vec{x},\vec{y})$ over the current instance of the data layer is such that $\vec{x}$ functionally determines $\vec{y}$, i.e., for each pair of tuples in the answer having the same values for $\vec{x}$, the corresponding values for $\vec{y}$ are the same. Intuitively, this condition ensures that for each tuple in R (i.e., a tuple with values for $\vec{x}$) to be updated, there is a single tuple with values for $\vec{y}$ to be used in the update. We can thus provide an `update+` operator defined as

$$\texttt{update+}\ R\ \texttt{set}\ \vec{t}\ \texttt{where}\ q^+(\vec{x},\vec{y}) \wedge Q^-(\vec{x},\vec{y})$$

where $\vec{t}$ and $q^+(\vec{x},\vec{y}) \wedge Q^-(\vec{x},\vec{y})$ are as above. Also in this case, we constrain the specification so that only one UPDATE operation for R can be defined in the set of effect specifications EFFECT(α) for an action α.

In the general case no syntactic restrictions can be imposed on $Q(\vec{x}, \vec{y})$ to ensure the existence of the functional dependency from $\vec{x}$ to $\vec{y}$, which can only be checked at query execution time. If the required functional dependency is violated, the new instance of R cannot be incrementally built from R by updating a subset of its tuples. However it can still be constructed by resorting to the general semantics. In this case, though, the intended semantics of the update is not preserved in general, as the resulting instance of R may contain additional tuples, namely those generated by "updating" an R-tuple with different sets of values.

4 Towards a Rule-Eengine Based Implementation

The executable nature of DCDS models, coupled with the efficient implementation induced by the operators defined above, makes them well suited for the realization of a support system for rapid prototyping of data-centric processes. In particular, state-of-the-art rule engines, such as the open-source Java-based Drools Expert rule engine[5] at the heart of the following discussion, represent a viable technological solution for supporting the declarative specification of a DCDS and for providing the run-time environment that supports the DCDS operational semantics of CA rules and actions.

Data Modeling. The relational schema at the heart of the data layer can be directly mapped to an object-oriented representation of the application domain. In particular, each relation schema is represented as a class having as name the name of the relation and as instance variables the attributes defined in the relation schema, so that each object instance of such a class is considered as a fact and corresponds to a tuple of a relation. representing Java beans), or the Drools type declaration language can be exploited for defining fact types and their attributes.

Example 2. In the travel reimbursement example, the Hotel relation can be represented by the following fact type declaration:

```
declare Hotel
  hName:String      date:Date          price:double
  currency:String   priceInUSD:double
end
```

Actions Modeling and Implementation. DCDS action specifications serve as a basis for driving their implementation, and declarative specifications of action effects can be mapped into a concrete procedural implementation of the action. As the Drools framework provides support for the definition of named queries over the data model, the query part of each effect specification is directly represented by a corresponding Drools query. Queries are used to retrieve fact

[5] http://www.jboss.org/drools/

sets based on patterns[6], and a query has an optional set of parameters that we exploit for binding query parameters that refer to action parameters.

Basically, executing an effect requires to execute the corresponding query and the use the query result set according to the specific operation associated with the effect. For `insert`, `delete` and `update/update+` operations, objects representing facts are respectively inserted, deleted and updated in the engine's working memory, by exploiting the `insert()`, `retract()` and `update()` methods provided by the engine. For a `set` operation, existing facts are first retracted, and generated facts are then inserted in the working memory.

Example 3. The effect $\mathsf{Hotel}(x_1, \ldots, x_5) \wedge \mathsf{ApprHotel}(x_1) \rightsquigarrow \mathsf{Status}(`accepted')$ defined in the example, corresponds to a `set` operation of the form

$$\texttt{set}\ `accepted'\ \texttt{for}\ Status(status)\ \texttt{where}\ \mathsf{Hotel}(x_1, \ldots, x_5) \wedge \mathsf{ApprHotel}(x_1)$$

whose query is mapped to the Drools query

```
query "Hotel Approved"
  Hotel($x1:hName,$x2:date,$x3:price,$x4:currency,$x5:priceInUSD)
  ApprHotel(hName == $x1)
end
```

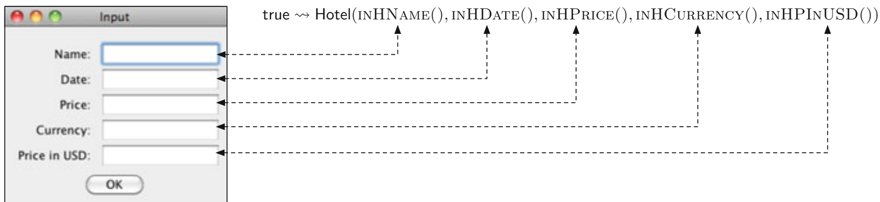

Fig. 1. Form-based user involvement in action executions.

For effects defined as `insert`, `set` and `update/update+` operations, the instantiation/update of new/existing facts may require to obtain new data from the external environment. In general, each functional term can be mapped to a method call, that may consist in a user-defined piece of code, may correspond to a remote method invocation or Web service call, or it may implement the specific logic for generating an input dialog so as to get user's input. updating a fact can be grouped in a single user form, automatically built from the type or class definition (Fig. 1). Attributes whose values are given by the user are shown as input fields, while attributes whose value is already defined (e.g., by a constant/action parameters) are shown as non-editable fields.

Process Modeling. The modeling of a DCDS process as a set of CA rules is directly represented as a set of rules defined in the Drools Rule Language (DRL).

[6] Conditions defined in a query or in CA rules are referred to as *patterns*, and the process of matching patterns against the data is called *pattern matching*.

For each condition-action rule of the form $Q \mapsto \alpha$, a corresponding named rule is created, as `rule "ruleName" when` Q `then` $execute(\alpha)$ `end`.

Example 4. In the travel reimbursement example, the condition-action rule represented as $\mathsf{Status}(`readyToVerify') \mapsto VerifyRequest$ is mapped to the following rule

```
rule "Verify Request"
    when    Status(status == StatusEnum.READY_TO_VERIFY)
    then    Executor.perform(new VerifyRequest());
end
```

Process Execution. The insertion of new data (as well as the update or deletion of existing data), either when the data layer is first instantiated or as a result of action executions, acts as a trigger for the rules evaluation process. and adopts a forward chaining data-driven approach in the process of matching new or existing facts in working memory against the rules, to infer conclusions which result in actions. Rules whose condition part is fully matched become eligible for execution, and the evaluation process can result in multiple eligible rules, i.e executable actions. and according to a conflict resolution strategy determines a single rule activation to be executed. According to our rule definitions, the firing of a rule activation results in the creation and execution of an action instance with a binding for its parameters. The execution of an action results in the insertion, deletion and update of facts in working memory, and the engine starts a new match-resolve-act cycle, where previously activated rules may be de-activated (as their condition is no longer matched by the actual facts) and removed from the agenda, and new instances may be activated, resulting in a new set of executable actions.

5 Conclusions

The DCDS framework induces a data-centric process management approach, where models (i) rely on a complete integration between processes and data, and (ii) are both *verifiable* and *executable*. As a first step towards a reference implementation for DCDSs, we proposed a modeling approach for DCDSs that, on the basis of specific constructs for specifying actions' effects, enables an efficient implementation of action executions, through incremental changes over the current instance of the data model. While rule engines can be exploited for rapid prototyping of DCDSs, several aspects still need to be considered. In particular, the resource perspective must be incorporated into the picture. Data-centric models are able to support an integrated modeling of human resources and data, by combining classical role-based organizational meta-models with a fine-grained modeling of users and their domain-specific roles in relation to data elements. At a process specification level, in line with the well-known resource patterns, this allows declaratively defining possible bindings between actions and human performers on the basis of both user- and data-aware conditions that guard

the executability of actions, going beyond simple role-based assignment policies. Similarly, run-time user involvement in the selection of executable actions has to be considered, investigating both the link with classical worklist-based approaches and the possibility of supporting knowledge workers with decision-support features.

Acknowledgments. This work has been partially supported by the Sapienza grants TESTMED, SUPER and "Premio Ricercatori Under-40", and by the EU FP7-ICT Project ACSI (257593).

References

1. Bagheri Hariri, 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. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of Data-Aware Process Analysis: A Database Theory Perspective. In: Proceedings of PODS (2013)
3. Cohn, D., Dhoolia, P., Heath III, F., Pinel, F., Vergo, J.: Siena: from powerpoint to web app in 5 minutes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 722–723. Springer, Heidelberg (2008)
4. 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)
5. Hull, R., Damaggio, E., De Masellis, R. et al.: Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events. In: Proceedings of DEBS '11 (2011)
6. Kunzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. J. Softw. Maint. Evol.: Res. Pract. **23**(4), 205–244 (2011)
7. Kunzle, V., Weber, B., Reichert, M.: Object-aware business processes: fundamental requirements and their support in existing approaches. Int. J. Inf. Syst. Model. Design (IJISMD) **2**(2), 19–46 (2011)
8. Meyer, A., Smirnov, S., Weske, M.: Data in business processes. EMISA Forum **31**(3), 5–31 (2011)
9. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full support for loosely-structured processes. In: Proceedings of EDOC (2007)
10. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns: identification, representation and tool support. In: Delcambre, L.M.L., Kop, Ch., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 353–368. Springer, Heidelberg (2005)
11. 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)
12. Swenson, K.D. (ed.): Mastering the Unpredictable: How Adaptive Case Management Will Revolutionize the Way That Knowledge Workers Get Things Done. Meghan-Kiffer Press, Tampa (2010)
13. Vaculin, R., Hull, R., Heath, T., Cochran, C., Nigam, A., Sukaviriya, P.: Declarative business artifact centric modeling of decision and knowledge intensive business processes. In: Proceedings of EDOC 2011 (2011)