2013 IEEE 6th International Conference on Service-Oriented Computing and Applications

Implementing and Running Data-Centric Dynamic Systems

Alessandro Russo, Massimo Mecella, Fabio Patrizi
DIAG, SAPIENZA Università di Roma, Italy
Email: {arusso|mecella|patrizi}@diag.uniroma1.it

Marco Montali
KRDB Centre, Free University of Bozen-Bolzano, Italy
Email: 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, and we discuss how it can be realized into a prototype system which is able to enact processes comprising human actors, services and data. This reference implementation exploits the natural correspondence between DCDSs and state-of-the-art rule engines, e.g., JBoss Drools, and present the interesting feature that the model used for analysis and verification is fully aligned with the one adopted for the execution.**

I. INTRODUCTION

Most of the current approaches to Business Process Management (BPM), and strictly related to composition and orchestration of (Web) services (either SOAP-based or RESTful), 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[1] progression is given by activity completions that enable subsequent tasks according to the control-flow. Languages such as BPMN, YAWL and WS-BPEL (for what strictly concerns services), 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, such as Declare [11][2], 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 [8]. 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 [9]. 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 [5], 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 [10] 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 [8]. 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* [7], 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

[1] Due to the strict relationship between *processes* and *compositions/orchestrations of services* [6], in the following we will use interchangeably the two concepts, and preferably the one of *process*.

[2] In the service arena, WS-CDL can be somehow defined as constraint-based, as a choreography expresses in a declarative way the constraints over possible message exchanges among services.

978-1-4799-2701-2/13 $31.00 © 2013 IEEE
DOI 10.1109/SOCA.2013.37

in terms of: *(i)* abstract procedural process specifications, e.g., expressed as state machines or transition systems, as in SIENA [4]; *(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 [7] supported by the Barcelona GSM environment [15] [3].

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), a paradigm for supporting unstructured, unpredictable and unrepeatable business cases. In this direction, recently the Object Management Group (OMG) has released a first standard version of the Case Management Modeling Notation (CMMN) [4]; 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 [3] for a comprehensive discussion of the relevant literature. Among such frameworks, here we focus on the one recently presented 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 [14] 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. In particular our long-term vision is the definition, design and realization of such a reference implementation for DCDSs. Such an ambitious aim poses several challenges, some of them being preliminarily discussed in [13], 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 [14] produces a DCDS model that resembles an execution engine based on forward rules, and requires to realize some "tricks" and supportive relationships that are very similar to those ones that would serve precisely to manage the execution.

The contribution of this paper is to present an initial proof-of-concept (PoC) of our reference implementation, which is based on the challenges and implementation patterns presented in [13]. In particular, besides considering more challenges and presenting possible solutions, we will discuss how to base the implementation on state-of-the-art rule engines, e.g., JBoss Drools, how to incorporate the user interactions on the basis of the specification model (possibly semi-automatically generating such interactions from the model) on the one side, and the interaction with external services on the other side. The paper is organized as it follows: after briefly summarizing DCDSs for making the paper self-contained (cf. Section II), we will present the overall approach, the architecture and relevant features of our tool in Section III. It is accompanied by an online appendix (cf. https://dl.dropboxusercontent.com/u/14551169/DCDS_Example.jar) showing some code extracts for a very simple running example, and an effective Java application, built on top of JBoss Drools, realizing the running example, to be considered as the proof-of-concept of the whole approach. Finally in Section IV we will briefly discuss some other points to be addressed towards the final implementation of a DCDS-based process management system, thus laying down our next future work.

II. Background and Basic Concepts

A. 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

[3] Both the models and tools are integrated in a recently launched open-source project, leaded by IBM, named *BizArtifact* - cf. http://sourceforge.net/projects/bizartifact/.

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

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 $\text{ADOM}(\mathcal{I}_0)$[5], 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.
- E_i is the effect, i.e., a set of facts over $\mathcal{R}$, which includes as terms: terms in $\text{ADOM}(\mathcal{I}_0)$, 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. Technically, the process ϱ is a finite set of 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 $\text{ADOM}(\mathcal{I}_0)$.

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 $\mathsf{Status}(\textit{'readyForRequest'})$ 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., INENAME() for the name of the employee, INHNAME() for the hotel name, INHDATE() for the hotel arrival date, etc.). In particular, the 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

$$\begin{array}{lcl} \mathsf{true} & \rightsquigarrow & \mathsf{Travel}(\text{INENAME}()) \\ \mathsf{true} & \rightsquigarrow & \mathsf{Hotel}(\text{INHNAME}(), \text{INHDATE}(), \text{INHPRICE}(), \\ & & \text{INHCURRENCY}(), \text{INHPINUSD}()) \end{array}$$

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

$$\begin{array}{rcl} \mathsf{Hotel}(x_1, \ldots, x_5) \wedge \mathsf{ApprHotel}(x_1) & \rightsquigarrow & \mathsf{Status}(\textit{'accepted'}) \\ \mathsf{Hotel}(x_1, \ldots, x_5) \wedge \neg\mathsf{ApprHotel}(x_1) & \rightsquigarrow & \mathsf{Status}(\text{DECIDE}()) \\ \mathsf{Travel}(n) & \rightsquigarrow & \mathsf{Travel}(n) \\ \mathsf{Hotel}(x_1, \ldots, x_5) & \rightsquigarrow & \mathsf{Hotel}(x_1, \ldots, x_5) \\ \mathsf{Flight}(x_1, \ldots, x_5) & \rightsquigarrow & \mathsf{Flight}(x_1, \ldots, x_5) \\ \mathsf{ApprHotel}(x) & \rightsquigarrow & \mathsf{ApprHotel}(x) \end{array}$$

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): $\mathsf{Status}(\textit{'readyToVerify'}) \mapsto$ *VerifyRequest*, $\mathsf{Status}(\textit{'accepted'}) \mapsto$ *AcceptRequest*.

B. 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 [12], 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:

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

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[6]. 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 actors via forms/user interfaces.
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.

Basic Effect Patterns. Starting from the observation that a direct, naïve implementation of the action execution semantics described above yields, in general, a waste of computational resources (as the the data layer is basically re-instantiated from scratch when executing an action), in [13] we introduced a set of patterns that allow for an incremental management of the data layer. These patterns reflect classical Create-Read-Update-Delete (CRUD) operations and are summarized in Table I.

III. Designing and Implementing DCDSs

Data-centric approaches naturally induce a "bottom-up" design methodology, where the definition of the data layer enables the specification of the process layer that operates on it, according to the *data first principle* and the *data centered principle* mentioned in [2]. The main methodological steps for the design of a DCDS can be thus resumed as *(i)* identification and modeling of the business entities and their relationships, that characterize the domain of interest and represent the data layer; *(ii)* identification of the business activities and specification of the corresponding actions that operate on the data layer; and *(iii)* identification and specification of the overall process that drives and constraints action executions.

The executable nature of DCDS models, coupled with the efficient implementation induced by the operators summarized in Table I, makes them well suited for the realization of a support system for rapid prototyping of data-centric processes, and represents the key enabler towards the implementation of a full-fledged model-driven and data-centric process management system. In particular, state-of-the-art rule engines, such as the open-source Java-based Drools Expert rule engine[7] 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.

Figure 1 depicts the overall approach underlying a DCDS reference implementation. At design-time, the *process analyst* specifies the DCDS (model) by using an appropriate specification language, and possibly a (set of) specification (graphical) editor(s). This model, which as previously said, is the same

[6]We assume here two-way blocking service calls.
[7]http://www.jboss.org/drools/

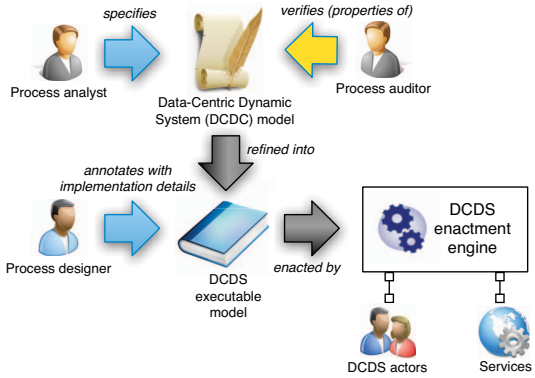

Fig. 1. The approach.

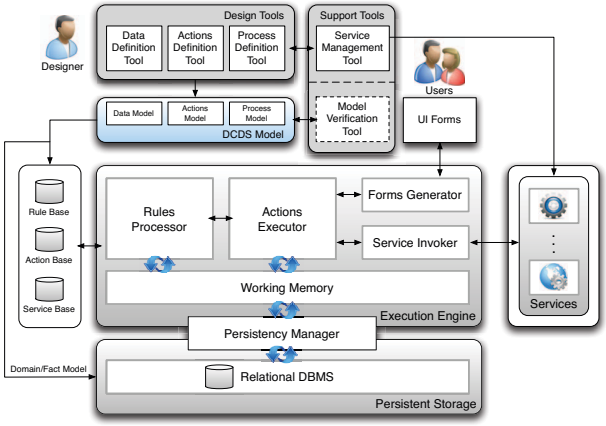

Fig. 2. The architecture.

used for subsequent enactment, is further refined through the specification of some technical aspects crucial for its effective enacting. In particular, as described in the following sections, some annotations should be provided in order to define how a template user interface for the DCDS should be generated, how external (Web) services should be retrieved and invoked, etc. This annotated and executable model is used for enactment of the DCDS during run-time, through the enactment engine.

As of the current status of our implementation, we have developed a proof-of-concept of the engine, as described in the following. The specification language, as well as the annotation one, are currently under development, on the basis of Java-like annotations and the Drools specification language for rules. The DCDS model can be verified for specific properties during the design time, but this aspect is currently out of the scope of this paper and our current implementation effort. In the following, we describe our reference implementation, depicted in Figure 2.

A. Data Modeling

As a first step in the concrete realization methodology, an explicit representation of the data model defining the domain of interest has to be provided. On the one side, a relational schema that constitutes a DCDS data layer finds its natural implementation in a relational DBMS; on the other side, rule engines typically rely on an in-memory object-oriented representation of the fact/data model. In order to bridge this gap, we rely on Object-Relational Mapping (ORM) techniques, so as

TABLE I. BASIC EFFECT PATTERNS FOR DCDS ACTION SPECIFICATIONS (SEE [13] FOR THE DETAILS).

Effect Pattern	DCDS Effect Representation	Description
set $\vec{t}$ **for** R **where** $q^+ \wedge Q^-$	$q^+ \wedge Q^- \rightsquigarrow R(\vec{t})$	Generic DCDS effect that generates tuples $\vec{t}$ to be set in the relation R in the new state.
insert $\vec{t}$ **into** R **where** $q^+ \wedge Q^-$	$\left\{ \begin{array}{rcl} q^+ \wedge Q^- & \rightsquigarrow & R(\vec{t}) \\ R(\vec{x}) & \rightsquigarrow & R(\vec{x}) \end{array} \right.$	Generates tuples $\vec{t}$ to be added to the relation R.
delete from R **where** Q^-	$R(\vec{x}) \wedge \neg Q^- \rightsquigarrow R(\vec{x})$	Deletes from the relation R the tuples that match the condition Q^-.
update R **set** $\vec{t}$ **where** Q^-	$\left\{ \begin{array}{rcl} R(\vec{x}) \wedge Q^- & \rightsquigarrow & R(\vec{t}) \\ R(\vec{x}) \wedge \neg(Q^-) & \rightsquigarrow & R(\vec{x}) \end{array} \right.$	Updates with tuples $\vec{t}$ the tuples in the relation R that match the condition Q^-.
update+ R **set** $\vec{t}$ **where** $q^+(\vec{x},\vec{y}) \wedge Q^-(\vec{x},\vec{y})$	$\left\{ \begin{array}{rcl} q^+ \wedge Q^- & \rightsquigarrow & R(\vec{t}) \\ R(\vec{x}) \wedge \widetilde{Q}^- & \rightsquigarrow & R(\vec{x}) \end{array} \right.$ where $\left[\begin{array}{l} q^+(\vec{x},\vec{y}) = R(\vec{x}) \wedge q(\vec{z},\vec{y}), \text{ with } \vec{z} \subseteq \vec{x} \\ \widetilde{Q}^- = \neg\exists\vec{y}\ (q(\vec{z},\vec{y}) \vee Q^-(\vec{x},\vec{y})) \end{array} \right]$	Updates with tuples $\vec{t}$ (whose values can be obtained also by querying the data layer) specific tuples in the relation R.

to enable an object-oriented representation of the domain/fact model and manage its persistency in a RDBMS. In particular, the designer is provided with a graphical Data Definition Tool for easy authoring of data models. The tool supports and drives the declarative definition of the set of named entities/relations, along with the corresponding typed attributes, that constitute a DCDS data layer. The basic definition of entities/relations and their attributes is further extended with annotations or metadata that can be classified as:

- *Persistency-related*: annotations that enable object-relational mapping techniques in order to manage the persistence of the domain model's objects; they basically reflect Java Persistence API (JPA) annotations and can be defined at the entity/relation level (e.g., `@Entity`) and at the attribute level (e.g., `@Id`).
- *Validation-related*: annotations that allow specifying or constraining the characteristics of object attributes' values. Specifically, it is possible to constrain attributes to a given finite number of values (`@Values`), define min/max values or ranges for numeric attributes (`@Min-value` and `@Max-value`), define min/max length for string-based attributes (`@Min-length` and `@Max-length`), and explicitly mark specific attributes as mandatory/required (`@Required`). Validation-related annotations defined at design-time will be used at run-time to validate data values produced by human performers during action executions.
- *UI-related*: annotations that allow configuring, constraining or customizing how data attributes are shown and visualized in graphical interfaces that enable a form-based interaction with users. Basic annotations allow specifying hidden attributes (`@UiHidden`), the label to be used for an attribute in the UI (`@UiLabel`), attributes with masked values (`@UiMasked`, e.g., for passwords) and read-only (i.e., not-editable) attributes (`@UiReadOnly`). UI-related annotations, along with the attributes' data types, are used at run-time for automatically generating graphical widgets for data attributes to support form-based user interaction, according to Object-User Interface Mapping (OIM) approach.

As a result of design-time data definition activities performed through the Data Definition Tool, an intermediate declarative data representation is produced, inspired by the Drools type declaration language that allows defining fact types and their attributes. The general structure of a data type definition is of the form:

```
declare relation/fact_name
  attribute_name : data_type   [metadata_annotations]
end
```

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

```
declare Hotel
  hName : String  @Required  @UILabel("Hotel Name")
  date : Date  @Required  @UILabel("Check-In Date")
  price : double  @Required  @Min-value(0)
  currency : String  @Required
  priceInUSD : double  @Required  @Min-value(0)
                       @UILabel("Price in USD")
end
```

The declarative data specifications are then compiled into annotated Plain Old Java Objects (POJOs) that collectively represent the domain and fact model. Basically, each declared relation/fact is mapped to a persistent POJO class or entity, whose persistence is transparently managed so that JPA entities can be directly manipulated as facts in the Drools engine working memory. Under this approach, a run-time object representing an instance of a POJO class is considered as a *fact* (from the perspective of the rule engine) and corresponds to a *tuple* of a relation (from a the perspective of the DCDS relational model).

B. Actions Modeling

In a DCDS, actions correspond to business activities that incapsulate an atomic unit of work from a process perspective[8]. Action specifications are supported through a graphical Actions Definition Tool, that drives the designer in the creation of a DCDS action base. As actions are defined in terms of input parameters and effects, from a methodological perspective the specification of an action is strongly data-aware and requires to identify and define (the formal action specification introduce in Section II): *(i)* an action *signature*, given by an action *name* and a (optional) typed sequence of *input parameters* required for executing the action; *(ii)* possible external data sources (in the form of users and/or software services able to provide input values) involved in the execution of an action, although modeled as function/service calls; *(iii)* the set of *effects* that the execution of the action will have on the data layer, in the form of facts being set, added, updated or deleted, according to the basic effect patterns we introduced in [13] and summarized in

[8]cf. the usual definition of *task/activity*, as a piece of work that forms one logical step within a process.

Table I. The general structure of an action definition resulting from the design steps is of the form:

```
action action_name([typed_parameters])
  effect_specification_1
  ...
  effect_specification_m
end
```

Recalling that each effect specification includes a query over the relational schema (cf. Table I), and exploiting the ability of the Drools framework to support the definition of named queries over the data/fact model, the query part of each effect specification is directly represented by a corresponding Drools query. Queries are used to retrieve fact sets based on patterns[9], and a query has an optional set of parameters that we exploit for binding query parameters that refer to action parameters.

Example 3: In the action *VerifyRequest* defined in the example, the effect

$$\mathsf{Hotel}(x_1, \ldots, x_5) \wedge \mathsf{ApprHotel}(x_1) \rightsquigarrow \mathsf{Status}(\textit{'accepted'})$$

corresponds to a `set` operation of the form

```
set 'accepted'   for Status(status)
                 where Hotel(x1,...,x5) ∧ ApprHotel(x1)
```

whose query $\mathsf{Hotel}(x_1, \ldots, x_5) \wedge \mathsf{ApprHotel}(x_1)$ is mapped to the Drools named query

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

The overall declarative action specification is thus of the form

```
action VerifyRequest()
  set "accepted" for Status(status)
                 where "Hotel Approved"
  ...
end
```

In the general case, DCDS actions (and processes that use them) are supposed to be executed in an environment that includes users and services that may provide data elements to be used when instantiating the data layer in the new state. The interaction with the environment and the input of data from the environment is explicitly modeled as service calls in the specification of the effects. In particular, effect specifications that follow the `set`, `insert` and `update/update+` patterns allow the inclusion of functional terms that represent service calls, as the instantiation/update of new/existing facts may require to obtain new data from the external environment. The modeling of service calls abstracts from the actual service implementation, so as to represent both user-provided data and data values produced by software services (e.g., Web services). The specification of action effects thus assumes the availability of external services (cf. the set $\mathcal{F}$ defined in a DCDS process layer). In order to build and maintain a *service base* that collects the set of services to be used in action specifications, the designer is provided with a Service Management Tool. The tool includes a Service Explorer application (Figure 3) that allows the designer to browse external Web services starting from the URL of their WSDL interface and select service operations to be added to the service base.

In the formal DCDS framework, no explicit distinction is made in function calls between user-provided data and service-provided data, as the specific data production/gathering logic is implemented by the services mapped by and hidden behind function calls. From a modeling perspective, the need to include software service calls in action effects is directly supported by allowing the designer to select services/operations from the service base and specify input mappings for services' parameters[10] (if any).

Fig. 3. Service Explorer Tool (a specific widget for exploring available Web services).

Example 4: Assume a domain model that includes information about hotels and their price, with relations $\mathsf{Cur} = \langle Currency \rangle$, $\mathsf{CurHotel} = \langle Hotel, Currency \rangle$, $\mathsf{PriceEntry} = \langle Hotel, Price, Date \rangle$. The process layer provides the possibility of converting the price list of a hotel from USD to another currency, by exploiting a Web service that offers an operation $\textsc{convusd}(price, currency, date)$ for price conversion. A DCDS action that performs the price conversion and updates the price entries and currencies can be specified as follows

```
action Convert(String hotel, String curr)
  update PriceEntry(x1, x2, x3)
         set (hotel, ConvUSD(x2, curr, x3), x3)
         where x1 = hotel && Cur(curr)
  update CurHotel(x1, x2)
         set (hotel, curr)
         where x1 = hotel && Cur(curr)
end
```

According to the `update` pattern, the tuples/facts modified by the first `update` effect (similar considerations also hold for the second `update`) are those that satisfy the query

```
PriceEntry(x1, x2, x3) && x1 = hotel && Cur(curr)
```

that is mapped to the parametrized named Drools query

```
query "Price Entries" (String hot, String cur)
  PriceEntry($x1: hotel == hot , $x2:price , $x3:date)
  Cur(currency == cur)
end
```

Similarly, the activation of a function representing user-provided data could be seen as corresponding to a service invocation that results in the generation of some sort of input field allowing the user to provide data values. In the case of the involvement of a human performer, the actual way that allows presenting work items to the user and the input data gathering logic may be far more complex, typically based on the concept of work-lists and the generation of graphical forms. To support form-based users involvement in action executions, we allow the annotation of effects so as to specify that input values needed for instantiating or updating a relation/fact have to be gathered from the user through a graphical form, that will be automatically built at run-time from the class definition of the relation/fact. Relation/fact attributes whose values are expected to be provided by users are marked with symbolic function symbols (as in the formal DCDS framework) annotated with the `@UIForm` annotation. At this stage, additional *validation-related*

[9]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*.

[10]Recall from Section II-A that functions/services can be applied to constants, free variables of the query part of the effect, and action parameters.

and *UI-related* annotations (cf. Section III-A) can be added, to overwrite or integrate the annotations defined during the data modeling stage.

Example 5: The effect specification of the form

```
set (InHName() @UIForm, InHDate() @UIForm,
     InHPrice() @UIForm, InHCurrency() @UIForm,
     InHPInUSD() @UIForm)
for Hotel where true
```

corresponds to the DCDS effect specification

$$\text{true} \rightsquigarrow \text{Hotel}(\text{INHNAME}(), \text{INHDATE}(), \text{INHPRICE}(), \text{INHCURRENCY}(), \text{INHPINUSD}())$$

for the action *InitiateRequest*, and the attributes of the `Hotel` relation/fact are annotated to specify that their values have to be gathered at run-time by interacting with a user through a graphical form.

Declarative action specifications serve as a basis for driving their implementation and, according to a truly model-driven approach, they enable the automatic generation of executable modules that implement the intended action execution semantics, as defined by the corresponding effects. Executable modules representing concrete procedural implementations of actions rely on the Command behavioral design pattern and implement the general execution strategy sketched in Section II-B, although specialized to reflect the intended semantics of the operator (`set`, `insert`, `delete`, `update/update+`) defined in each effect. While providing the details of how each effect pattern is compiled into executable code is out of the scope of this work, basically executing an effect requires to execute the corresponding query and then use the query result set according to the specific operation associated with the effect. For `insert`, `delete` and `update/update+` operations, objects representing tuples/facts are respectively added, deleted and updated through the persistency manager, and the engine's working memory is kept synchronized by exploiting the `insert()`, `retract()` and `update()` methods provided by the engine. For a `set` operation, existing tuples/facts are first retracted, and generated facts are then inserted in the working memory and persistent storage. Relevant details about service invocations and user forms generation are provided in Section III-D.

C. Process Modeling

Actions represent the atomic building blocks that enable the specification of processes that operate on the data layer and constitute the progression mechanism for a DCDS. In a DCDS, a process specification is given by a (finite) set of condition-action rules (CA rules). Under this approach, the constraints that determine actions executability are completely data-dependent, so that a complete integration of processes and data can be achieved. The Process Definition tool supports a declarative rule-based process specification approach, and 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). Each condition-action rule of the form $Q \mapsto \alpha$ finds its natural representation as a named Drools rule of the form

```
rule "ruleName" when Q then execute(α) end
```

In particular, the right-hand side (RHS) of a rule always defines the instantiation (with possible parameters given by constants and free variables of the query Q defined in the left-hand side of the rule) and execution of an action from the action base built in the actions modeling stage.

Example 6: The condition-action rule

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

directly corresponds to the condition-action rule represented as $\text{Status}(\textit{'readyToVerify'}) \mapsto \textit{VerifyRequest}$ in the travel reimbursement example.

D. Process and Actions Execution

The design steps produce a domain/fact model, and a specification of actions, their effects and the overall process. The set up of the DCDS run-time infrastructure, which relies on the Drools engine, requires to instantiate the Drools environment, by providing the initial instance of the data/fact model and build the so-called knowledge base, by loading into the engine the knowledge definitions represented by the queries used in actions effects and by the overall process specification as a set of CA rules.

The initial instance of the data layer can be created by instantiating a set of persistent objects representing the tuples/facts that constitute the initial relational instance. The initial set of facts are then inserted in the engine's working memory, so that they can be processed against the rules. 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. Drools inference engine is based on the well-known Rete algorithm 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. The overall rules evaluation and firing process implemented in Drools, based on match-resolve-act cycles, directly reflects the intended behavior of the abstract execution semantics we defined for DCDSs in Section II-B. 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. The agenda manages current rule activations, called conflict set, and according to a conflict resolution strategy determines a single rule activation to be executed. In our proof-of-concept implementation we adopt the default conflict resolution strategies available in Drools, based on rule priority or on a last-in-first-out (LIFO) policy. The possibility of defining custom conflict resolution strategies will allow us to implement and support different selection strategies, also involving users in the selection of executable actions, also considering the workflow resource patterns. 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.

As already discussed, the execution of an action corresponds to evaluating and applying the corresponding effects. In the most general and complex case (cf. the effect definition in Section II-A), the attributes of tuples/facts to be set, added or updated are defined in terms of constants, variables, service calls and user-provided inputs (cf. the `@UIForm` annotation). While variables are instantiated according to the binding for

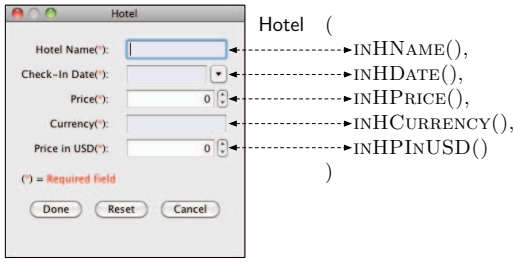

Fig. 4. Automatic generation of UI forms for user-based action executions.

action's parameters and variables of the query part of the effect, service calls and user-provided values require to interact with the environment.

Service calls, as defined and selected during the action modeling phase, result in the concrete invocation of the service operation, so as to obtain a return value for the tuple/fact attribute. Run-time service invocations are issued and managed by exploiting the Apache CXF framework[11] and its ability to support dynamic Web service clients. In the presence of relation/fact definitions with `@UIForm` annotations, form-based user interaction is provided. User forms and their automatic generation are often considered as a key enabler for data and object aware processes [8]. To support the dynamic and automatic run-time generation of user forms, we rely on the Metawidget UI framework[12]. The framework is an Object/User Interface Mapping tool (OIM) and it is able to generate UI widgets by inspecting annotated domain objects. Starting from an object representing a tuple/fact to be instantiated or updated, we are able to generate a user form where each attribute annotated as `@UIForm` is rendered as an input widget, on the basis of the corresponding data type and the validation- and UI-related annotations defined during the data and action modeling stages. In the presence of attributes whose values are given by constants, variables or service calls, the generated form can be configured so that they are hidden to the user or shown as not-editable fields. In addition, validation-related annotations are used to automatically validate user-provided input. As an example, Figure 4 shows the automatically generated form for the action effect defined in Example 5 on the basis of the definition for the `Hotel` relation/fact defined in Example 2. Although currently not supported, data visibility in UI forms can be further refined by considering fine-grained access-control policies that define read/write permissions for data attributes.

IV. Concluding Remarks

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*. In this paper we have presented our reference implementation for DCDSs, on the basis of specific constructs for specifying actions' effects, which enables an efficient implementation of action executions, through incremental changes over the current instance of the data model. In such a way, we can base our implementation on rule engines, thus obtaining rapid prototyping of DCDSs.

Our reference implementation still needs to be refined in several aspects, and validated. 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.

As far as validation of our implementation, besides classical evaluation of performances, we are particularly interested in evaluating analysts and designers inclination and plainness/simplicity in modeling wrt. more traditional activity-centric models. In particular, a user study comparing our reference implementation with well-known tools, e.g., YAWL, is envisioned as soon as our implementation effort is concluded. Indeed, the opportunity of such a user study has driven our effort, in order to assess with scientific rigor whether data-centric approaches are indeed better/equivalent/worse than traditional ones from the point of view of the analysts/designers.

[11] http://cxf.apache.org/

[12] http://metawidget.sourceforge.net/

Acknowledgment

This work has been partially supported by the Sapienza grants TESTMED, SUPER and "Premio Ricercatori Under-40".

References

[1] B. Bagheri Hariri, D. Calvanese et al. "Verification of relational data-centric dynamic systems with external services," in *Proc. PODS 2013*.

[2] K. Bhattacharya, R. Hull, and J. Su, "A data-centric design methodology for business processes," in *Handbook of Research on Business Process Modeling, chapter 23*, 2009, pp. 503–531.

[3] D. Calvanese, G. De Giacomo et al. "Foundations of data aware process analysis: A database theory perspective," in *Proc. PODS 2013*.

[4] D. Cohn, P. Dhoolia, F. Heath, F. Pinel, and J. Vergo, "Siena: From PowerPoint to Web App in 5 Minutes," in *Proc. ICSOC 2008*.

[5] M. Dumas, "On the Convergence of Data and Process Engineering," in *Proc. ADBIS'11*.

[6] T. Erl, *Service-Oriented Architecture: Concepts, Technology and Design*. Prentice Hall.

[7] R. Hull, E. Damaggio, R. De Masellis et al. "Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events," in *Proc. DEBS '11*.

[8] V. Künzle and M. Reichert, "PHILharmonicFlows: towards a framework for object-aware process management," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 4, pp. 205–244, 2011.

[9] V. Künzle, B. Weber, and M. Reichert, "Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 2, no. 2, pp. 19–46, 2011.

[10] A. Meyer, S. Smirnov, and M. Weske, "Data in business processes," *EMISA Forum*, vol. 31, no. 3, pp. 5–31, 2011.

[11] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *Proc. EDOC 2007*.

[12] N. Russell, A. H. M. ter Hofstede et al. "Workflow data patterns: identification, representation and tool support," in *Proc. ER'05*.

[13] A. Russo, M. Mecella, M. Montali, and F. Patrizi, "Towards a reference implementation for data centric dynamic systems," in *2nd International Workshop on Data and Artifact-Centric BPM (DAB'2013)*, to appear.

[14] D. Solomakhin, M. Montali et al. "Verification of Artifact-Centric Systems: Decidability and Modeling Issues," in *Proc. ICSOC 2013*.

[15] R. Vaculin, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya, "Declarative business artifact centric modeling of decision and knowledge intensive business processes," in *Proc. EDOC 2011*.