Implementing Data-Centric Dynamic Systems over a Relational DBMS

Diego Calvanese, Marco Montali, Fabio Patrizi, Andrey Rivkin

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

1 Introduction

We base our work on a model called *data-centric dynamic system* (DCDS) [1], which can be seen as a framework for modeling and verification of systems where both the process controlling the dynamics and the manipulation of data are equally central. More specifically, a DCDS consists of a *data layer* and a *process layer*, interacting as follows: the data layer stores all the data of interest in a relational database, and the process layer modifies and evolves such data by executing actions under the control of a process, possibly injecting into the system external data retrieved through service calls. In this work, we propose an implementation of DCDSs in which all aspects concerning not only the data layer but also the process layer, are realized by means of functionalities provided by a relational DBMS. We present the architecture of our prototype system, describe its functionality, and discuss the next steps we intend to take towards realizing a full-fledged DCDS-based system that supports verification of rich temporal properties.

2 Preliminaries

A DCDS is a tuple $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D}$ is the *data layer* and $\mathcal{P}$ is the *process layer*. The data layer defines the data model of $\mathcal{S}$; it is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, where: $\mathcal{C}$ is a countably infinite set of constants providing data values, $\mathcal{R}$ is a a relational schema equipped with equality and full denial constraints $\mathcal{E}$, and $\mathcal{I}_0$ is an initial database instance over $\mathcal{C}$ that conforms to the schema $\mathcal{R}$ and satisfies the constraints $\mathcal{E}$.

The *process layer* defines the progression mechanism for the DCDS; it is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \rho \rangle$, where: $\mathcal{F}$ is a finite set of functions representing calls to external services, $\mathcal{A}$ is a finite set of (update) actions, and ρ is a process specification.

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

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

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

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

As regards the process execution, it essentially amounts to iterating over the following steps. First, an action α is chosen by the user and the corresponding CA rule in ρ is evaluated over the current database instance $\mathcal{I}$ (initially $\mathcal{I}_0$). This produces a (finite) set of complete *bindings*, for α's parameters. Then, the user is asked to pick one of such bindings, say $\vec{p}$, so as to obtain a *ground action* $\alpha\vec{p}$. The next step is the action execution, which consists of applying all the action effects simultaneously. This requires *(i)* evaluating all queries $q_i\vec{p}$ (with partial assignment to their variables) associated with all effect specifications, *(ii)* binding the values occurring in the answers with the terms in all **del** D_i and all **add** A_i, and *(iii)* first deleting all the facts obtained in all D_i's, and subsequently adding those obtained in all A_i's, from and to the current database instance $\mathcal{I}$. In case some term t in some A_i involves a service call, the corresponding service is called with the appropriate inputs, to obtain the value to be assigned to t. We stress that all deletions take place at the same time, followed by all additions. Importantly, the final update (deletions and additions) is actually performed only if the resulting instance satisfies the constraints in $\mathcal{E}$ (otherwise a new iteration starts again on the current database instance, but the current binding is no longer provided as an option to the user). When the update is performed, a new instance $\mathcal{I}'$ is obtained, over which the process can iterate again. For a detailed description of the execution semantics, we refer the reader to [1]. In the next section, we describe an actual implementation of the system based on RDBMS technology.

3 Specifying and Implementing DCDSs in a RDBMS

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

Specifically, the RDBMSs maintains: *(i)* a *data layer specification* consisting of relational tables, equipped with functional dependencies and additional domain-dependent constraints, and *(ii)* a *process layer specification*, consisting of action metadata in the form of a relational table containing the action names together with their parameters. Moreover, the DBMS stores sets of stored procedures. Each stored procedure represents

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

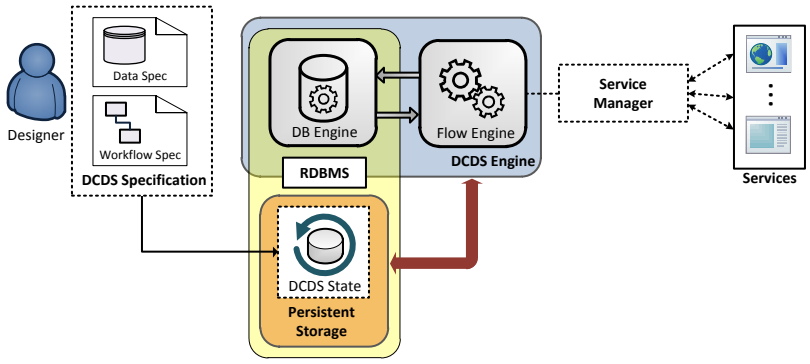

Fig. 1. Architecture of the DCDS implementation

(i) the evaluation of a query in the precondition of a CA rule or a query in an action effect, or *(ii)* an action execution (in terms of database updates).

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

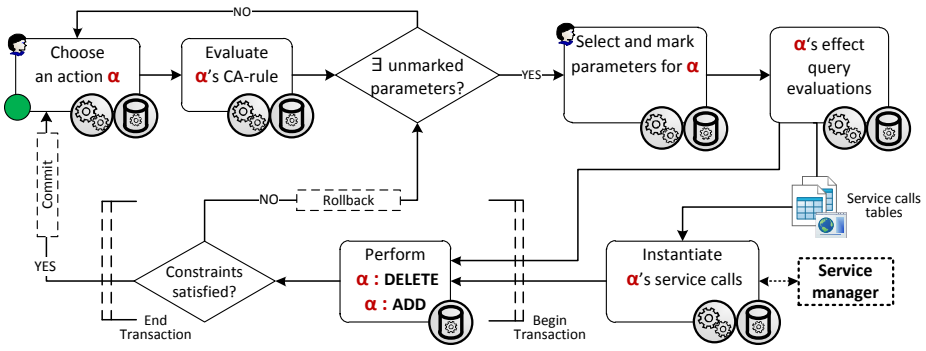

Fig. 2. The action execution cycle

choose. *(3)* If unmarked parameters are present in T, the user is asked to choose one of those, which is marked as unavailable, and the Flow Engine proceeds with evaluating α instantiated with the chosen parameters. *(4)* To do so, first the queries in all the effects of α are executed, which provides values to instantiate both the arguments of the service calls (stored in *service calls tables*, one for each service), and the facts to delete and add from the current snapshot. *(5)* The service calls are executed by calling the *service manager*, and the returned results provide the missing instantiations for the facts to add. *(6)* A transaction is started to perform the delete and add operations. *(7)* If the DCDS constraints are satisfied, the transaction is committed, and the next iteration of the action execution cycle is started. Otherwise, if the constraints are not satisfied, the transaction is aborted so that the DCDS snapshot stays unmodified, and the user is asked to choose a different parameter from the ones still available in T. *(8)* If no unmarked parameters are available, the user is asked to choose another action.

4 Conclusion and Future Work

In this paper we have presented an implementation of Data-Centric Dynamic Systems based on the use of a Relational Database Management System, exploited to handle *all* the aspects of a DCDS, including specification and execution.

The ultimate goal of this research is to build a system for model checking DCDSs. As discussed in [1], the possible evolutions of these systems can be represented by means of a transition system, that can thus be checked against temporal first-order properties, see, e.g., [1,2,4]. The main problem with this task is that the state space can be infinite, thus the standard model checking techniques cannot be applied off-the-shelf. However, under certain conditions, namely when the active domain of all states is guaranteed to be *bounded* [1,2], it is possible to build a faithful abstraction of the system that is finite-state and can thus be checked with such techniques. Our next step will consist in extending our prototype system with the capability to build the finite abstraction, maintaining it in the DBMS, and exploiting state-of-the-art model checking tools, such as NuXMV [3], to perform verification.

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: Proc. of PODS. pp. 163–174 (2013)
2. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of agent-based artifact systems. JAIR 51, 333–376 (2014)
3. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Proc. of CAV. LNCS, vol. 8559, pp. 334–342. Springer (2014)
4. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proc. of ICDT. pp. 252–267 (2009)