Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

KRDB Research Centre
Knowledge Representation meets DataBases

Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
Tel: +39 04710 16000, fax: +39 04710 16009

KRDB Research Centre Technical Report:

RAW-SYS a Practical Framework for Data-aware Business Process Verification

Riccardo De Masellis[1], Chiara Di Francescomarino[1], Chiara Ghidini[1], Marco Montali[2], Sergio Tessaris[2]

Affiliation	1: FBK-IRST, Via Sommarive 18, 38050 Trento, Italy 2: Free University of Bozen–Bolzano, piazza Università, 1, 39100 Bozen-Bolzano, Italy
Corresponding author	Sergio Tessaris: `tessaris@inf.unibz.it`
Keywords	workflow-nets with data, data-centric dynamic systems, action languages, planning, Business Process Management systems
Number	KRDB16-1
Date	March 21, 2016
URL	`http://www.inf.unibz.it/krdb/pub/tech-rep.php#KRDB16-01`

1. Introduction

The need to extend business processes with the capability to handle complex data objects has been increasingly recognised in the BPM area [15, 18, 9], and has lead to significant practical and theoretical advances in the field. On the practical side, several well-established suites capturing both the process control-flow and its relevant data are nowadays available to end users as commercial and non-commercial tools. Notable examples are Bizagi BPM Suite, Bonita BPM, YAWL and Activiti. Despite different modeling choices (e.g., in the "front-end" graphical modelling language) and advanced functionalities, all such tools share a set of common features. Business processes are expressed by means of typical workflow constructs, and the data dimension is added by using variables of complex data types, usually stored in a (relational) database. More importantly, *how data are modified* is often hidden inside the logic of activities, implemented, e.g., with Java classes, hence resulting in an essentially activity-centric model, where data are introduced in an ad-hoc way, as a sort of "procedural attachment" [9]. As a consequence, when coming to the formal verification of data related aspects, these tools either offer only basic features (such as Data Type checks) without considering the interaction between control- and data-flow, or they fail to correctly compute the answers to verification questions. This does not come as a surprise, as verification of standard properties like reachability is intrinsically undecidable when applied to data-aware processes, unless constraints are introduced to limit how the control-flow interacts with the data component. On the theoretical side, this problem has been thoroughly studied during the last 15 years, and there is a significant body of literature on the boundaries of decidability and complexity for the verification of data-aware processes against different classes of formal properties. We can divide this literature in two main streams. In the first stream, variants of Petri nets are enriched, by making tokens able to carry various forms of data, and by making transitions aware of such data. Examples in this stream are the well-known Colored Petri Nets, or the many different forms of "data-enabled" nets [17, 23, 4, 24]. Such models suffer of two main limitations. On the one hand, the modelling paradigm they adopt is different from the one used in the aforementioned BPM suites, which do not only make tokens and transitions data-aware, but also explicitly account for persistent data repositories (i.e., typically, of how the process operates over full-fledged relational databases). On the other hand, relevant properties remain either undecidable, or become decidable only after the data-related aspects are extremely weakened. The second stream contains proposals that take a different approach: instead of taking a control-flow model and making it increasingly data-aware, they consider standard data models (such as relational databases and XML repositories) and make them increasingly "dynamics-aware". Notable examples in this stream are relational transducers [3], active XML [2], the artifact-centric paradigm [22, 15], and the recently proposed model of data-centric dynamic systems (DCDSs) [5]. These models maintain decidability with expressive data models and are, in principle, in line with the way BPM suites deal with processes and data. Nonetheless, the languages they propose are abstract and far from concrete BP modelling languages and architectures. Moreover, they lack any tool support. As a consequence, they struggle to produce an impact in the BPM community.

In this report we introduce a framework (RAW-SYS) aiming at bridging the gap between theory and practice, for modelling and verifying data-aware processes as represented by well established BPM suites. From the modelling point of view, we achieve our goal by combining three approaches: *(i)* workflow nets [26], the most popular formal language for specifying the process control-flow; *(ii)* relational database with expressive constraints (such as integrity constraints, keys, foreign keys, and so on), the most widely adopted model for describing data; and *(iii)* DCDS, one of the richest formalisms to express how an atomic task in the process may update the underlying relational database, possibly introducing fresh data not already present in the system [5]. This combination leverages on, and overcomes the limitations of,

Petri Nets and data-centric approaches. It also ensures RAW-SYS to reflect concrete BPM architectures while maintaining solid formal basis.

2. The RAW-SYS Framework

Our intention is to propose a framework for modeling data-aware processes that accounts for the concrete models used inside BPM suites, but also generalises from the specificities of a single suite. This is achieved in three steps: first, we shape RAW-SYS to capture the intuition behind most of the adopted BPM tools, by using workflow nets as a language-independent way for representing the control-flow related aspects; second we rely on the standard relational technology for the data-related aspects; and third we borrow some key concepts introduced in formal models for data-aware processes [5, 20], to capture how atomic tasks operate over the underlying data, a crucial aspect that suites typically leave underspecified and heavily dependent on ad-hoc implementations.

Figure 1 depicts the conceptualisation of the framework where different processes (*cases*) execute in parallel using private data stores (the *local* database). The dynamic aspects of processes are described by means of WFNets, while tasks are specified by updates on the local database and fetch data from the environment via web services or user interaction. Running cases do not interact directly but store results of their processing on a global data store (the *shared* database). Cases cannot access the shared database, which is updated only when a new case is created or terminated. Periodically, data from the shared database are transferred

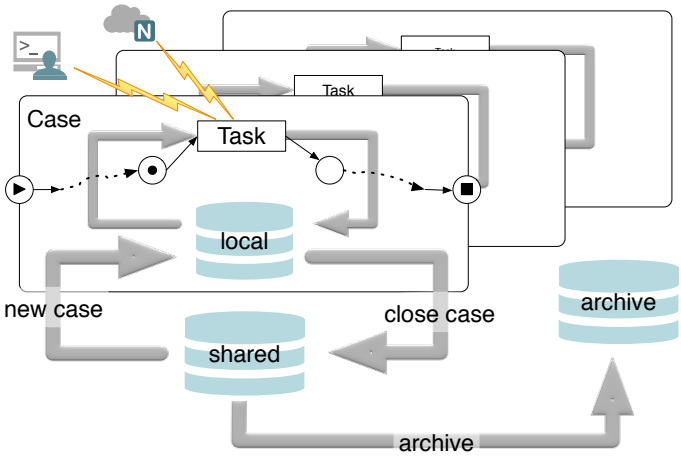

Figure 1: RAW-SYS architecture.

to the *archive* database where they are no longer accessible to the system and can be used only for external auditing or analysis; this transferral can be controlled by queries over the current state of the system.

Example 1. *Assume a scenario in which company* GoodsKit *is equipped with an informative system which manages data and processes. Such a system can be represented, in an abstract manner, by the architectural view in Figure 1 which can easily be mapped to real process modelling suites such as the ones mentioned in Section 1. The informative system manages data and processes about processes that cover all the business needed by* GoodsKit, *from good's production to employees' reimbursement procedures (*$\mathbb{RB}$*) for their business trips (whose workflow is depicted in Figure 2 and illustrated in detail in the next section). At runtime, each process is instantiates by a number of processes instances, or* cases, *that an be active at the same time, progressing through different tasks by usually interacting with external agents and*

modifying data as a result of such interactions. For example, assume that one of the instances active in the system refers to execution of $\mathbb{RB}$*, handling the reimbursement of the business trip of employee* john *to* NewYork*. The instance progresses when, e.g.,* john *fills a reimbursement form for his trip.*

Data model. The data model is built on top of a standard relational database with a schema (i.e., a set of relations) and a set of denial constraints, expressed as safe range FO formulae over a *fixed common* countably infinite set of constants [1].[1] In addition, we associate to the relational database an *initial database instance* satisfying the constraints.

Example 2. *The schema for the global data store contains, in general, data of interest for the whole organisation, so data manipulated by all processes. Due to lack of space, we restrict to relations accessed by the* $\mathbb{RB}$ *process. Global relations* $\mathtt{Empl}(\underline{empl})$ *and* $\mathtt{Dest}(\underline{dest})$ *contain the employee entitled to have business trips and the allowed destinations. Relation* $\mathtt{Pending}(\underline{empl, dest})$ *contains requests that are still pending, while* $\mathtt{Accepted}(empl, dest, amnt)$ *and* $\mathtt{Rejected}(empl, dest)$ *those that are respectively successfully completed or rejected (in the first case, also keeping track of the* $amnt$ *refunded to the employee). Underlined attributes are primary keys. Moreover, we have that* $\mathtt{Pending}[empl]$ *and* $\mathtt{Pending}[dest]$ *are two foreign keys respectively referencing* $\mathtt{Empl}[empl]$ *and* $\mathtt{Dest}[dest]$ *(similarly for* $\mathtt{Accepted}$ *and* $\mathtt{Rejected}$*).*

The $\mathbb{RB}$ *local relations are: (i) CurrReq*$(empl, dest, status)$*, listing the employees that requested a reimbursement, the trip destination and the status of the request; (ii) TrvlMaxAmnt*$(maxAmnt)$*, containing the maximum amount estimated by the travels office for the trip; (iii) TrvlCost*$(cost)$*, storing the costs sustained by the employee.*

Control-flow. The process control-flow is modelled in a prescriptive fashion leveraging standard workflow nets [26] enriched with data. We call the resulting nets *DWF-nets*. We assume familiarity with the well-known framework of workflow nets, and consequently focus on our data-aware extension. Intuitively, DWF-nets are standard workflow nets extended with a (local) relational database. The local database is populated with initial data when a token is inserted in the input place of the net. In addition, transitions are extended to deal with data: their execution is guarded by queries over the local database, and leads to induce an update over such database. As for the workflow nets constituting the basis of our DWF-nets, we concentrate on 1-safe nets, which generalise the class of *structured workflows* and are the basis for best practices in process modelling [16].[2] It is important to notice that our approach can be seamlessly generalized to other classes of Petri nets, as long as it is guaranteed that they are k-safe. This reflects the fact that the process control-flow is well-defined.

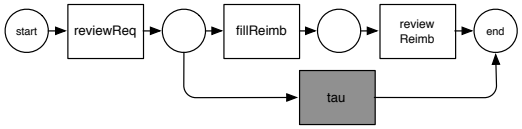

Figure 2: Travel Management Workflow-net

Example 3. *We resume the reimbursement (*$\mathbb{RB}$*) example by reporting its control-flow in terms of a workflow net (see Figure 2).*

Each net has an associated local database that stores private data not accessible by other running process. Each transition of the net interacts with the underlying local database. In particular, each transition expresses the execution of a task in terms of three different components:

[1] This is standard. Recall that relational algebra is safe range by construction.

[2] Recall that a workflow net is k-safe if each place in the initial marking – and in all the reachable markings – contains at most k tokens.

- *guard*, a FO query over the local database determining whether the transition can trigger or not, and binding its parameters accordingly;
- *(nondeterministic) external services*, functions representing the the interface to external sources of information, such as input forms used by human users, or calls to external web services, in the style of [5, 20];
- *action*, a specification of (possibly bulk) updates over the local database, which depend on the current database content, the binding provided by the guard, and the additional data obtained through the external services.

The valid firing of a transition – in addition to the usual marking conditions on the input places, as specified by the workflow net – is conditioned by the satisfiability of the guard and the successful application of the action. Note that an action might be unsuccessful because of a violation of a constraint in the local database.

The dynamic of the system is defined using *actions* that update database instances. Many different update formalisms exist in the literature. We adopt the approach in [5, 20], which allows to express virtually any pattern of update, including CRUD operations over the local database, but also bulk operations that simultaneously manipulate large portions of the database at once. An action, in turn, is described by a set of add and remove operations on the database instance "conditioned" by a safe range query. Formally an action is an expression $\textsc{act}(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$ with each e_i of the form:

$$Q(\vec{x}) \rightsquigarrow \textbf{add } R(\vec{x}) \text{ or } Q(\vec{x}) \rightsquigarrow \textbf{del } R(\vec{x})$$

where Q is a safe range query over the database schema, and $R(\vec{x})$ a literal over the schema. Effects sharing the same query can be grouped into a single expression. The effects e_i are assumed to take place simultaneously.[3]

In addition $R(\vec{x})$ may include external input functions that model the interaction with external services: at runtime function occurrences are substituted by the value returned by issuing the corresponding service call. Some of those services can be guaranteed to return *fresh values* (i.e. constants not included in the current database). This feature is essential to enable the creation of new objects via new identifiers (e.g., the creation of a new order by obtaining a unique order identifier).

Example 4. *Recall that activity reviewRequest examines the employee's request to leave for a business trip and evaluates the maximum reimbursable amount (possibly based on the selected destination). The effect of the corresponding action* RvwReq() *is specified as:*

$$\begin{aligned} \mathit{CurrReq}(e,d,s) \rightsquigarrow\ & \textbf{\textit{del}}\ \{\mathit{CurrReq}(e,d,s)\} \\ & \textbf{\textit{add}}\{\mathit{CurrReq}(e,d,\textbf{status}()),\ \mathit{TrvlMaxAmnt}(\textbf{maxAmnt}())\} \end{aligned}$$

In order to update the request status, the tuple representing the current travel request in the CurrReq relation must be first deleted and then added with the new status (recall that additions have higher priority than deletions). Query $\mathit{CurrReq}(e,d,s)$ *selects such a tuple, effect* $\textbf{\textit{del}}\{\mathit{CurrReq}(e,d,s)\}$ *deletes it while* $\textbf{\textit{add}}\{\mathit{CurrReq}(e,d,\textbf{status}())\}$ *adds the same tuple but with a new value for the status. Also the value of the max reimbursable amount is added by the fact* $\mathit{TrvlMaxAmnt}(\textbf{maxAmnt}())$*. Notice the use of functions* **status**() *and* **maxAmnt**() *to model unknown values coming from the external environment, in our case the company travels office.*

2.1. Data model

Relational database schema. The data component is a full-fledged relational database with constraints. Technically, $\mathcal{D} = \langle \Delta, \mathcal{R}, \mathcal{C}, \mathcal{I}_0 \rangle$, where:

[3] As customary in planning and reasoning about actions, additions have higher priority than deletions; this automatically sorts out the case in which the same fact is asserted to be deleted and added at once.

- Δ is a countably infinite set of constants.
- $\mathcal{R}$ is a *database schema*, i.e., a set of relation schemas. We will equivalently adopt the positional notation or the attribute-based named notation for relations; When the latter notation is used, an n-ary relation R is represented as $R(U)$, where U is a set of n named attributes. Furthermore, given a set $A \subseteq U$ of attributes, $R[A]$ represents the projection of R over A.
- $\mathcal{C}$ is a set of *safe range FO-constraints* over $\mathcal{R}$, capturing the real-world constraints of the targeted application domain.
- $\mathcal{I}_0$ is the *initial database instance* of $\mathcal{S}$, i.e., a database instance conforming to $\mathcal{R}$, satisfying the constraints $\mathcal{C}$, and made of values in Δ.

Among all possible FO-constraints, we consider the following specific constraints, which are widespread in database modelling and standard conceptual modelling languages such as UML class diagrams, E-R diagrams, and the ORM notation:

- Standard *key* and *primary key* constraints. We use notation KEY($R[A]$) (resp., $\underline{R}A$) to model that the set A of attributes is a key (resp., primary key) for relation R.
- Standard *foreign key* constraints. We use notation $R[A] \longrightarrow S[B]$ to model that the set A of attributes in R is a foreign key pointing to the set B of attributes in S, such that $\underline{S}B$ holds.
- *Cardinality-constraints*, which resemble cardinality/frequency constraints of conceptual modelling languages. Cardinality-constraints generalize key constraints by bounding the minimum and maximum number of tuples allowed in a relation when the value of some attributes is maintained unaltered. Given a relation $R(U)$ and a set $A \subseteq U$ of attributes, notation CARD($R[A], m..n$), where m and n are positive integers, denotes the *cardinality constraint* requiring that the number of R-tuples with the same values for attributes A ranges between m and n.
- A combination of cardinality and foreign key constraints, where a foreign key has an associated cardinality constraint guaranteeing that the number of tuples pointing to the same target primary key is bounded. We denote by $A[R] \xrightarrow{m..n} B[S]$ the database constraint corresponding to the conjunction of $A[R] \longrightarrow B[S]$ and CARD($A[R], m..n$), and call such a conjunction a *cardinality-bounded foreign key* constraint. Notice that $A[R] \xrightarrow{1..1} B[S]$ is equivalent to the combination of KEY($R[A]$) and $A[R] \longrightarrow B[S]$.

In the following we assume a fixed shared domain Δ. An instance of a database is a mapping from each relations in $\mathcal{R}$ to a *finite* set of tuples over Δ. Each tuple is labeled by a constant in Δ which represent *provenance* information; a special symbol $\epsilon \in \Delta$ labels tuples in the initial database instance.

Functions. The interface to a (nondeterministic) *external service* (e.g. user interaction) is modelled by means of finite sets of functions from $\bigcup_{n \geq 0}(\Delta^n \mapsto \Delta)$. We assume a set of skolem terms $f(\vec{x})$ associated with a function of the same arity. For the sake of simplicity, in the following we'll use $\mathcal{F}$ to indicate both a set of functions and corresponding skolem constants.

Actions. Given a source $\mathcal{D}^s = \langle \Delta, \mathcal{R}^s, \mathcal{C}^s, \mathcal{I}_0^s \rangle$ and target $\mathcal{D}^s = \langle \Delta, \mathcal{R}^s, \mathcal{C}^s, \mathcal{I}_0^s \rangle$ relational databases, and a finite set of functions $\mathcal{F}$, an *action* over $\mathcal{D}^s$, $\mathcal{D}^t$, and $\mathcal{F}$ is an expression $\textsc{act}_{\mathcal{D}^s,\mathcal{D}^t,\mathcal{F}}(\mathrm{p}_1, \ldots, \mathrm{p}_n) : \{e_1, \ldots, e_m\}$, where:

- $\textsc{act}(\mathrm{p}_1, \ldots, \mathrm{p}_n)$ is the action *signature*, constituted by a name $\textsc{act}$ and a sequence $\mathrm{p}_1, \ldots, \mathrm{p}_n$ of *parameters*, to be substituted with values when the action is invoked;
- $\{e_1, \ldots, e_m\}$, also denoted as $\textsc{effect}(\textsc{act})$, is a set of *effects*, which are assumed to take place simultaneously.

Each effect e_i has the form

$$Q(\vec{\mathrm{p}}, \vec{x}) \rightsquigarrow \textbf{add } A(\vec{\mathrm{p}}, \vec{x}, \vec{y}) \textbf{ del } D(\vec{\mathrm{p}}, \vec{x})$$

where:

- Q is a safe range FO query over $\mathcal{R}^s$ whose terms are variables, action parameters, and constants from $\mathcal{I}_0^s$. Intuitively, Q selects the tuples to instantiate the effect with. During the execution, the effect is applied with a ground substitution $\vec{d}$ for the action parameters, and for every answer θ to the query $Q(\vec{d}, \vec{x})$.
- A is a set of facts over $\mathcal{R}^t$, which include as terms: free variables $\vec{x}$ of Q, action parameters $\vec{p}$, and/or Skolem terms $f(\vec{x}', \vec{p}')$ (with $\vec{x}' \subseteq \vec{x}$, and $\vec{p}' \subseteq \vec{p}$) corresponding to functions in $\mathcal{F}$. At runtime, whenever a ground Skolem term is produced by applying substitution θ to A, the corresponding service call is issued, replacing it with the result (from Δ) returned by the invoked service. The ground set of facts so obtained is *added* its current database instance.
- D is also a set of facts over $\mathcal{R}^t$, which include as terms free variables $\vec{x}$ of Q and action parameters $\vec{p}$. At runtime, the ground facts obtained by applying substitution θ to D are *removed* from the current database instance.

As in STRIPS, we assume that additions have higher priority than deletions (i.e., if the same fact is asserted to be added and deleted during the same execution step, then the fact is added). The "**add** A" part (resp., the "**del** D" part) can be omitted if $A = \emptyset$ (resp., $D = \emptyset$).

2.2. Petri Nets with Data

Definition 1 (Petri Net [13]). *A Petri Net is a triple $\langle P, T, F \rangle$ where*

- *P is a set of places;*
- *T is a set of transitions;*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation describing the "arcs" between places and transitions (and between transitions and places).*

The preset *of a transition t is the set of its input places: ${}^\bullet t = \{p \in P \mid (p, t) \in F\}$. The* postset *of t is the set of its output places: $t^\bullet = \{p \in P \mid (t, p) \in F\}$. Definitions of pre- and post-sets of places are analogous.*

The marking *of a Petri net is a total mapping $M : P \mapsto \mathbb{N}$.*

Definition 2 (WF-net [25]). *A Petri net $\langle P, T, F \rangle$ is a workflow net (WF-net) if it has a single source place start, a single sink place end, and every place and every transition is on a path from start to end; i.e. for all $n \in P \cup T$, $(start, n) \in F^*$ and $(n, end) \in F^*$, where F^* is the reflexive transitive closure of F.*

The semantics of a PN is defined in terms of its markings and *valid firing* of transitions which change the marking. A firing of a transition $t \in T$ from M to M' is valid – denoted by $M \xrightarrow{t} M'$ – iff:

- t is enabled in M, i.e., $\{p \in P \mid M(p) > 0\} \supseteq {}^\bullet t$; and
- the marking M' satisfies the property that for every $p \in P$:

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + 1 & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) & \text{otherwise} \end{cases}$$

Definition 3 (safeness). *A marking of a Petri Net is k-safe if the number of tokens in all places is at most k. A Petri Net is k-safe if the initial marking is k-safe and the marking of all traces is k-safe.*

In this document we focus on 1-safeness, which is equivalent to the original safeness property as defined in [26].[4] Note that for safe nets the range of markings is restricted to $\{0, 1\}$. This class of networks generalises *structured workflows* and are the basis for best practices in process modelling [16]. It is important to notice that our approach can be seamlessly generalised to other classes of Petri nets, as long as it is guaranteed that they are k-safe. This reflects the fact that the process control-flow is well-defined.

Definition 4 (DP-net [13]). *A Petri Net with data is a tuple $\langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ where*

- *$\mathcal{D}$ is a relational database schema;*
- *$\langle P, T, F \rangle$ is a Petri Net;*
- *$\mathcal{F}$ is a function that associates each transition with a finite set of* functions*, each representing the interface to a (nondeterministic)* external service*;*
- *$\mathcal{A}$ is a function that associates each transition t with an action over $\mathcal{D}, \mathcal{D}$ and $\mathcal{F}(t)$;*
- *$\mathcal{G}$ is a function that associates each transition with a safe range query over $\mathcal{D}$ (the guard).*

Task actions may include parameters, in that case the guard free variables must match the parameters of the corresponding tasks.

A state of a DP-net $\langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ is a pair (M, I) where M is a marking for $\langle P, T, F \rangle$ and I is an instance of the relational database $\mathcal{D}$.

Definition 5 (Valid DP-net Firing). *Given a DP-net $\langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$, a valid firing of a transition t in (M, I) resulting in a state (M', I') (written as $(M, I) \xrightarrow{t} (M', I')$) iff:*

- *t is enabled in M, i.e., $\{p \in P \mid M(p) > 0\} \supseteq {}^\bullet t$; and*
- *the guard $\mathcal{G}(t)(\vec{c})$ is satisfied (i.e. evaluates to true) w.r.t. I and a ground instantiation $\vec{c}$ of its free variables;*
- *the marking M' satisfies the property that for every $p \in P$:*

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + 1 & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) & \text{otherwise} \end{cases}$$

- *the new database instance I' is the results of the application of $\mathcal{A}(t)(\vec{c})$;*
- *I' satisfies the constraints in $\mathcal{D}$.*

A *Workflow net with data* (DWF-net) is a tuple $\langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ where $\langle P, T, F \rangle$ is a WF-net.

Within Δ we identify a set of constants $\wp$ which represents the set of *process identifiers*.

Process case. A *(process) case* is a tuple $\langle i, w, (M, I) \rangle$ where $i \in \wp$ is a process id, w a DWF-net and (M, I) a state of w. A case represents the snapshot of a single process instance of a DFW-net.

[4] In the following we will use safeness as a synonym of 1-safeness.

2.3. Business Process Model and Semantics

A RAW-SYS model of a business process consists of a set of DWF-nets together with a shared relational database. In addition, the model specifies, for each DWF-net, an initialisation and an update actions that respectively detail how a local database is initialised upon a case activation for that process, and how the shared database is updated upon the termination of a process. The initialisation actions, in addition to the initialisation of the local database instance, could update the shared database; e.g. a purchase order taken in charge by a process is removed from the queue. The activation of new processes is regulated by means of queries over the shared database that verify whether a new case for a specific net can can be created.

Definition 6 (Business Process Model)**.** *A Business Process Model is a tuple* $\langle \mathcal{D}_G, \mathcal{W}, \mathcal{S}, \mathcal{E}, \sigma \rangle$*, where*

- $\mathcal{D}_G$ *is a (shared) relational database;*
- $\mathcal{W}$ *is a set of workflow nets with data;*
- $\mathcal{S}$ *is a function that associates each WF-net in* $w \in \mathcal{W}$ *having the local database* $\mathcal{D}_w$ *with an action over* $\mathcal{D}_G, \mathcal{D}_w \cup \mathcal{D}_G, \emptyset$ *that initialises the local database using the shared database;*[5]
- σ *is a function that associates each WF-net in* $w \in \mathcal{W}$ *with a safe range query over the shared database to verify wether a new case of that net can be instantiated.*
- $\mathcal{E}$ *is a function that associates each WF-net in* $w \in \mathcal{W}$ *having the local database* $\mathcal{D}_w$ *with an action over* $\mathcal{D}_w \cup \mathcal{D}_G, \mathcal{D}_G, \emptyset$ *that updates the shared database using the local database;*

Case initialisation actions may include parameters, in that case the corresponding query free variables must match these parameters.

Semantics is provided in term of global states (*snapshots*) that include the set of all active cases as well as the instances of the shared and archive databases. The behaviour of the system is described by means of all the possible sequences of snapshots evolving from the initial one. The initial snapshot has an empty set of cases (none of the processes is active), an empty archive database instance, and the shared database instance specified. It is worth noting that, due to the presence of external service calls and also due to the possibility of nondeterministically spawning new process cases, the execution semantics of a RAW-SYS needs in general to account for an infinite number of states, as well as truly infinite runs that may visit infinitely many different databases. This is a standard issue when processes are made data-aware [9].

Definition 7 (Business Process Model Snapshot)**.** *A* business process model snapshot *of* $\langle \mathcal{D}_G, \mathcal{W}, \mathcal{S}, \mathcal{E}, \sigma \rangle$ *is a tuple* $\langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C} \rangle$ *where:*

- $\mathcal{I}_S$ *is a (shared) database instance of* $\mathcal{D}_G$*;*
- $\mathcal{I}_A$ *is an (archive) database instance of* $\mathcal{D}_G$*;*
- $\mathcal{C}$ *is a set of cases from* $\mathcal{W}$*.*

Sequences of valid snapshots are defined according to the following four types of transitions that can nondeterministically evolve the system.

Case firing: one of the tasks of an active case can be executed by picking a suitable substitution for its parameters and for the involved service calls, updating the local database instance and the net marking.

[5] The action can update the shared database as well.

Creation of a new case: queries over the shared database establish which RAW-NET can be activated and the associated parameters. Among these a new case is created and included among the set of active cases. The local database is initialised using the data from the shared database instance and query parameters, while the marking is initialised with a token in the initial, input place. Each new case is associated with a unique fresh identifier, and the initialisation is specified by means of an action that might update also the shared database (so as to globally inform the system that a new case indeed started).

Termination of a completed case: one of the active cases having a token in the final, output place is removed from the set of active cases, and the shared database instance is updated according to the associated action. If the update violates one of the constraints the transition is not valid, therefore the case cannot be terminated nor removed from the set of active cases.

Archival of data: given a set of terminated process ids, the tuples in the shared instance marked with those ids are moved to the archive database instance. The transition is valid only if the shared database constraints are satisfied both in the resulting shared and archive instances. If any information must be guaranteed to be always present in the shared database, and not transferable to the archive, then this can be enforced by means of integrity constraints.

Definition 8 (Valid Transitions). *Given a business process model snapshot $\langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C} \rangle$, a* valid transition *leading to a new snapshot $\langle \mathcal{I}'_S, \mathcal{I}'_A, \mathcal{C}' \rangle$ is one of the following type of transitions.*

1. ***Case firing**: if $(M, I) \xrightarrow{t} (M', I')$ valid DWF-net firing for a case $\omega = \langle i, w, (M, I) \rangle \in \mathcal{C}$. Then the new snapshot is $\langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C}' \rangle$ where $\mathcal{C}' = \mathcal{C} \setminus \{\omega\} \cup \{\langle i, w, (M', I') \rangle\}$.*
2. ***Activation of a new case**: given a DWF-net $w = \langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ s.t. $\sigma(w)(\vec{c})$ is satisfied on $\mathcal{I}_S$ for a ground instantiation $\vec{c}$ of its free variables, a fresh process id $i \in \wp$, M a marking of w with a single token in the start place, and I, $\mathcal{I}'_S$ are legal database instances of $\mathcal{D}_w$ and $\mathcal{D}_G$ generated by the action $\mathcal{S}(w)(\vec{c})$ over the shared database $\mathcal{I}_S$. Then the new snapshot is $\langle \mathcal{I}'_S, \mathcal{I}_A, \mathcal{C}' \rangle$ where $\mathcal{C}' = \mathcal{C} \cup \{\langle i, w, (M, I) \rangle\}$. Tuples added to $\mathcal{I}'_S$ by the action are marked with the new fresh process id.*
3. ***Termination of a completed case**: given a case $\omega = \langle i, w, (M, I) \rangle \in \mathcal{C}$ with a token in the end place, $\mathcal{I}'_S$ a the legal database instance of $\mathcal{D}_G$ generated by the action $\mathcal{E}(w)$ using the database instance I. Then the new snapshot is $\langle \mathcal{I}'_S, \mathcal{I}_A, \mathcal{C}' \rangle$ where $\mathcal{C}' = \mathcal{C} \setminus \{\omega\}$.*
4. ***Archival of data**: given a set of inactive process ids $\mathcal{P}$ – that is for any case $\omega = \langle i, w, (M, I) \rangle \in \mathcal{C}$, $i \notin \mathcal{P}$ – the set of tuples τ in $\mathcal{I}_S$ marked by an id in $\mathcal{P}$, $\mathcal{I}'_S = \mathcal{I}_S \setminus \tau$ a legal instance of $\mathcal{D}_G$, and $\mathcal{I}'_A = \mathcal{I}'_A \cup \tau$ a legal instance of $\mathcal{D}_A$. Then the new snapshot is $\langle \mathcal{I}'_S, \mathcal{I}'_A, \mathcal{C} \rangle$.*

Note that tuples marked by terminated processes can always be archived unless doing so is prevented by the violation of integrity constraints on the shared database. Moreover, initial tuples (those marked with ϵ) are never archived and can only be deleted/updated by means of task actions.

If any information must be guaranteed to be always present in the shared database, then this can be enforced by means of integrity constraints.

Definition 9 (RAW-SYS trajectories). *A trajectory is a sequence of snapshots starting from the initial snapshot and where each pair of adjacent snapshots is compatible with a valid transition.*

3. Reachability Analysis in RAW-SYSs

We now consider the reasoning task of reachability in RAW-SYSs. This amounts to check whether there exists a run of the input RAW-SYS that reaches a state in which a desired

boolean query, expressed over the global data store, holds. In order to enable queries over the state of the processes we include an additional meta-predicate bind to the marking of the active cases. This additional predicate does not affect the properties of the system as described below. Moreover, as clarified in the following sections, its encoding in the DCDS model does not require any additional infrastructure.

Undecidability of reachability. It is well-known that, once the pure control-flow perspective of BPM is enriched with the explicit manipulation of data, reachability turns out to be undecidable, even by severely restricting the modeling capabilities of the data-aware process language at hand [9]. This strong negative result immediately carries over our setting. Many different techniques have been proposed in the literature so as to regain decidability of reachability and of more sophisticated forms of temporal model checking (see [27, 9] for a survey of results). In this work, we rely on the notion of *state-boundedness*, which has been extensively exploited for providing strong, robust decidability conditions in a plethora of data-aware process frameworks, including systems working over complete, relational data [8, 5], systems with incomplete data enriched with ontologies [7, 11], as well as Situation Calculus action theories [12]. Essentially, state-boundedness requires to limit, a-priori, the number of objects that can co-exist in the same state. For a relational database, this means that the size of the database cannot exceed a fixed threshold. Note that a state-bounded system still accepts unboundedly many objects to appear within and across runs.

In all such previous works, there is a single, global data storage, whose size is subject to the bound. In our setting, there are three information sources to which boundedness can be applied: the shared data store, the local data store, and the number of active cases. Two research questions then arise: *(1)* Can state-boundedness help towards decidability of planning over RAW-SYSs? *(2)* If so, which of the information sources must be necessarily bounded towards decidability?

We start by attacking the second research question, showing that as soon as one of such three information sources is not bounded, reachability of a query constituted by an atomic proposition is undecidable. All proofs are via reduction from the halting problem for deterministic, two-counter machines (2CMs), well-known to be undecidable [19].

We first consider two cases where there can be only one active case, and either the global or the local data stores are unbounded, while the other data store is bounded.

Theorem 1. *Checking reachability of an atomic proposition is undecidable over RAW-SYSs where: (i) the shared data store contains a single proposition only; (ii) there is exactly one RAW-NET, of which at most one instance is running; (iii) the local database this RAW-NET contains only two unary relations, whose extension is not bounded.*

Proof sketch. Given a 2CM $\mathcal{M}$, we construct a RAW-SYS $\mathcal{S}$ that reaches a target atomic proposition if and only if $\mathcal{M}$ halts. $\mathcal{S}$ has a shared data store that is initially empty, and whose schema contains only a single atomic proposition Hit. Furthermore, it contains a single RAW-NET $\mathcal{N}$ that simulates the computation of $\mathcal{M}$. The structure of $\mathcal{M}$ is encoded into the workflow net of $\mathcal{N}$: *(i)* each place corresponds to a control-state of $\mathcal{M}$; *(ii)* the input and output places correspond to the start and halting control-states of $\mathcal{M}$; *(iii)* each transition corresponds to a counter operation, where the workflow net expresses how the operation updates the current control-state of $\mathcal{M}$, whereas the update induced by the transition over the local data store mimics the counter manipulation; *(iv)* decrement transitions and transitions executed when a counter is 0 are guarded by two corresponding queries over the local data store (see below). The values maintained by the two counters correspond to the size of the extension of two unary relations C_1 and C_2, which constitute the schema of the local data store. Incrementing counter i amounts to introduce a fresh value in the extension of C_i, decrementing counter i amounts to delete an object in the extension of C_i, and testing whether counter i is (not) 0 amounts to check whether the extension of C_i is (not) empty. Finally, the update

induced by $\mathcal{N}$ on the shared data store upon completion simply amounts to toggle the *Hit* flag. In this way, $\mathcal{S}$ reaches *Hit* if and only if $\mathcal{M}$ halts. □

Theorem 2. *Checking reachability of an atomic proposition is undecidable over RAW-SYSs where: (i) the global data store contains only two unary relations, whose extension is not bounded; (ii) there is exactly one RAW-NET, of which at most one instance is running; (iii) the local database this RAW-NET is always empty.*

Proof sketch. The proof is similar to that of Theorem 1, with the difference that the unary relations used to simulate the counters are now in the shared data store. The shared data store is also equipped with a finite set of atomic propositions, one per state of the 2CM. In a given snapshot, only one of such atomic propositions holds (modeling that the 2CM is in a certain current state). The single RAW-NET has a trivial structure with an input and output places connected by a single, tau transition. Upon creation of an instance of such a RAW-NET with, it is checked which of such propositions currently holds, triggering a suitable, immediate update of the current state (i.e., substituting the current proposition with the next one, in accordance to the control-state update of the 2CM), and an update on the extension of the two counter relations, with the same strategy discussed in the proof of Theorem 1. □

The most tricky case is the one in which the shared and local data stores are all bounded, but the number of simultaneously active cases is not. Towards undecidability, we cannot anymore exploit the same technique adopted in the previous two cases, since it is not possible anymore to exploit the local/shared data store to remember the value of the two counters. Furthermore, when a certain process case becomes active, its evolution cannot be affected by that of other, simultaneously active cases, since it only works on its local data. In spite of such two strong limitations, we get the following.

Theorem 3. *Checking reachability of an atomic proposition is undecidable over RAW-SYSs where: (i) there is exactly one RAW-NET, of which unboundedly many instances can simultaneously coexist; (ii) the global data store, as well as the local data store of RAW-NET, contain unary relations only, whose size is bounded.*

Proof sketch. Let $\mathcal{N}$ be the single RAW-NET of the considered RAW-SYS. The workflow net of $\mathcal{N}$ is again the trivial net containing a single, no-op transition. However, it is now associated to two sophisticated updates to be applied when an instance of $\mathcal{N}$ is created or terminates its execution,

The two counters are simulated using two "chains" of active cases, where the value of the counter is the length of the chain minus 1. The main difficulty is how to rigidly keep track of the ordering between cases in a chain, and how to properly manipulate the chain, given the fact that the information about the chain itself cannot be stored anywhere, being all the data stores bounded. We attack this problem as follows. First of all, each instance of $\mathcal{N}$ exposes itself via a "ticket", i.e., a unique identifier that is made explicit in the global data store (this is necessary, since the internal case identifiers are not visible). The global data store remembers the current, control-state of the 2CM by adopting the same technique as in the proof of Theorem 2. It also remembers the extreme points of the two chains (i.e., the tickets of the instances at their top/bottom).

Since the 2CM is deterministic, we can assume that each control-state has a unique successor state obtained via a counter-increment operation, or two successor states, one achieved when a counter is positive and gets decremented, the other achieved when the same counter is zero. Increment is then simulated by allowing for the creation of a new $\mathcal{N}$-instance only if the current control-state has an increment transition. Upon creation, the instance consumes the information about the instance that is currently at the top of the corresponding chain, remembering the corresponding ticket in a local, "previous ticket" relation. At the same time,

it generates a fresh ticket identifying itself, and updates the global data store by declaring that this ticket is now at the top of the chain. This immediately simulates increment, and therefore the very same update also updates the control-state.

Decrement transitions for a counter are simulated by the termination of the active $\mathcal{N}$-instance that is at the top of the corresponding chain. However, termination is not explicitly controllable in the specification: all active $\mathcal{N}$-instances evolve in parallel and in isolation to each other, and consequently there is no explicit way of selecting only the instance at the top of the chain and induce its termination. To enforce this, we leverage the fact that a RAW-NET case can properly terminate only if the corresponding update satisfies the constraints of the shared data store. In particular, we make sure that whenever a $\mathcal{N}$-instance wants to terminate, a constraint is violated if the ticket of that instance does not correspond to that at the top of the chain. When the top-instance is picked, it successfully terminates by updating the control-state, and declaring that the top ticket is now the one store in its "previous ticket" relation.

Transitions triggered by a zero test are simulated in a similar way, discriminating them from decrement transitions by simply checking whether the selected, active $\mathcal{N}$-instance is not only at the top of the chain, but also at the bottom (i.e., the chain contains only such an instance). □

Decidability for Bounded RAW-SYSs. To attack the given strong undecidability results, we now concentrate on the situation were all thre three information sources for RAW-SYSs are bounded. We simply refer to such systems as *bounded RAW-SYSs*. We stress that such systems are by no means finite-state, since they still allow for storing unboundedly many data within and across system runs, provided that they do not accumulate in the same snapshot. In particular, bounding the number of simultaneously active instances can be seen as a sort of "limited resources" assumption, for which the resources of a company can be allocated to boundedly many cases, but once a case is finished, its resource is freed and ready to be reallocated to a new case (making hence possible to execute infinitely many cases along time). For bounded RAW-SYSs, we finally prove the following positive result, thanks to a reduction to the framework of data-centric dynamic systems (DCDSs) [5].

Theorem 4. *Checking reachability over bounded RAW-SYSs is decidable in PSPACE in the size of the initial shared data store.*

Proof sketch. The proof is done in three steps: *(1)* we provide a behavior-preserving encoding of RAW-SYSs into DCDSs; *(2)* we argue that if the input RAW-SYS is bounded, then the DCDS obtained via the encoding is state-bounded in the sense of [5]; *(3)* we formulate reachability over the input RAW-SYS as a verification problem over the corresponding DCDS - decidability is then obtained by [5], in which verification over state-bounded DCDSs has been shown to be decidable. The main guideline for the encoding is as follows. The DCDS data component is obtained by combining the global data store together with the local data stores. All such relations are augmented with an additional attribute that explicitly accounts for the "provenance" of each tuple. In addition, an accessory relation is added so as to track the control-flow state of each such instance. The dynamic component is obtained by introducing dedicated actions for the creation/archival of cases, and for the update induced by a case when it terminates. In addition, each RAW-NET net is translated into a set of dedicated actions (one per transition in the workflow net), following the same strategy as in [6]. The fact that the obtained DCDS is state-boundedness derives from the boundedness if the input RAW-SYS, and the fact that we focus on 1-safe workflow nets. As for the complexity, n general verification over state-bounded DCDSs requires to explore, in the worst-case, a number of states that is exponential in the size of the initial database. However, in the case of reachability, this exploration can be done on-the-fly, using space that is polynomial in the size of the initial data store. □

4. Encoding RAW-SYS framework in DCDS

Let us consider the BPM $\langle \mathcal{D}_G, \mathcal{W}, \mathcal{S}, \mathcal{E}, \sigma \rangle$. We show how the behaviour of the model can be encoded in a DCDS $\langle \mathcal{D}, \mathcal{P} \rangle$. To simplify the description of the encoding we assume that every task and place in $\mathcal{W}$ DP-nets are distinct, as well as action names and relations.

4.1. Data Component

Although in the framework we consider different relational databases, DCDS condition-action rules operates on a single database that must include all the necessary relations. Without loss of generality we can assume that all the schema signatures are distinct (including the ones of the shared and archive databases).

- The schema of $\mathcal{D}$ is the union of the schema $\mathcal{D}_G, \mathcal{D}_A$ and all the schemata of the nets in $\mathcal{W}$.
- Each relation include an additional attribute restricted to the set of process identifiers (the *ID attribute*). For any original relation R we indicate with $R^{\wp}$ the corresponding relation with the additional attribute and with $R^{\wp(x)}(\vec{y})$ the literal $R^{\wp}(x, \vec{y})$.[6]
- Let $Q(\vec{x})$ be a query using local or shared schemata, then its translation $Q(\vec{x})^{\wp(i)}$ it's a formula with the same structure of Q were each literal $R(\vec{y})$ is substituted by $R^{\wp(i)}(\vec{y})$ if R is local or $\exists i R^{\wp(i)}(\vec{y})$ if R is a shared or archive relation.
- The domain Δ include the set of places of all the nets in $\mathcal{W}$. We assume that the set of places of each net are distinct.
- The schema include an additional relations used for "process bookkeeping"

  Marking binary relation ranging over the process ids and the places. The relation store the information about "active" places, i.e. those holding a token.

  Active unary relation holding the set of terminated process ids whose tuples are not yet archived.

  ToArchive unary relation holding the set of process ids to be archived.
- Integrity constraints of shared and archive databases holds "across" ID attributes; i.e. the original constraints in the DCDS database hold on the of the new relations were the id attribute is "projected away". Given an the integrity constraint γ (closed safe range FOL formula) over the shared/archive database, the corresponding DCDS constraint is $\gamma^{\wp}$. Since all relations in γ are shared, then the process id variable is not relevant.
- Integrity constraints of local databases are "contextual" w.r.t. the ID attribute. That means that the all the additional PID arguments will be bound to the same variable universally quantified. Given an the integrity constraint γ for a local database, the corresponding DCDS constraint is

  $$\forall i(\exists p Marking(i, p) \rightarrow \gamma^{\wp(i)})$$

  Local databases are relevant only while processes are running, i.e. there are tokens in some place.

[6] We assume that the attribute is always the first one.

4.2. Process Component

Each of the kind of transactions are encoded into condition-action rules evolving the data component according to the above defined semantics.

Actions defined in the framework are translated into DCDS over the data component by taking into account the additional process id attribute. In particular, each action will have an additional parameter that is going to be instantiated with the process id of the case to which the action is applied.

Let $\textsc{act}(\mathrm{p}_1, \ldots, \mathrm{p}_n) : \{e_1, \ldots, e_m\}$ be a RAW-SYS action over $\mathcal{D}^s, \mathcal{D}^t, \mathcal{F}$. Without loss of generality we can assume that each effect contains a single relation on the rhs; i.e. is in one of the two forms

$$Q(\vec{\mathrm{p}}, \vec{x}) \rightsquigarrow \textbf{add } R(\vec{y}) \tag{1}$$

$$Q(\vec{\mathrm{p}}, \vec{x}) \rightsquigarrow \textbf{del } R(\vec{y}) \tag{2}$$

where R is a local or global relation and $\vec{y}$ is a subset of $\vec{\mathrm{p}}, \vec{x}$ or some skolem constants. The corresponding translation $\textsc{act}(\mathrm{p}_1, \ldots, \mathrm{p}_n)^{\wp(i)}$ includes a new parameter i corresponding to the process id, and it's defined as $\textsc{act}(i, \mathrm{p}_1, \ldots, \mathrm{p}_n) : \{e_1^{\wp(i)}, \ldots, e_m^{\wp(i)}\}$ where an effect $Q(\vec{\mathrm{p}}, \vec{x}) \rightsquigarrow op\ R(\vec{y})$ translated according to the kind of update op and whether R is local or shared:

(add) $Q(\vec{\mathrm{p}}, \vec{x})^{\wp(i)} \rightsquigarrow \textbf{add } R^{\wp(i)}(\vec{y})$ in both shared and local cases;

(del, local) $Q(\vec{\mathrm{p}}, \vec{x})^{\wp(i)} \rightsquigarrow \textbf{del } R^{\wp(i)}(\vec{y})$;

(del, shared) $Q(\vec{\mathrm{p}}, \vec{x})^{\wp(i)} \wedge \exists \vec{y'} R^{\wp(i')}(\vec{y'}) \rightsquigarrow \textbf{del } R^{\wp(i')}(\vec{y})$.

Since there might be several tuples in the DCDS database corresponding to the same tuple in the shared database, each one with a different process id, then the delete must remove all the occurrences regardless of the process id.

Given a model $\langle \mathcal{D}_C, \mathcal{W}, \mathcal{S}, \mathcal{E}, \sigma \rangle$, the corresponding process component of the DCDS system is defined by the following ca-rules simulating the above described transitions.

Case firing Each task is translated into a DCDS condition-action rule where the query of the condition-action rule include the original guard (literals enriched by the process id argument) conjoined with a condition ensuring that all of the input places are active.

The action include an additional parameter for the process id that is included added to all the literals and an additional effect that "deactivate" the input places and "activate" the output ones; updating the *Marking* relation.

Let t be a task in one of the DP-nets $\langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$, $\mathcal{G}(t)$ the guard of t, $\mathcal{A}(t) = \textsc{act}_t(\vec{c}) : \{e_1, \ldots, e_m\}$ its action, $\{p_1^i, \ldots, p_k^i\} = {}^\bullet t$ its input places, and $\{p_1^o, \ldots, p_\ell^o\} = t^+$ its output places. The corresponding ca-rule became:

$$Marking(i, p_1^i) \wedge \ldots \wedge Marking(i, p_k^i) \wedge \mathcal{G}(t)(\vec{c})^{\wp(i)} \mapsto \left\{ \begin{array}{l} e_1^{\wp(i)}, \ldots, e_m^{\wp(i)} \\ \text{true} \rightsquigarrow \textbf{del } \{Marking(i, p_1^i), \ldots, Marking(i, p_k^i)\} \\ \text{true} \rightsquigarrow \textbf{add } \{Marking(i, p_1^o), \ldots, Marking(i, p_\ell^o)\} \end{array} \right\}$$

Activation of a new case For each net in $\mathcal{W}$ there is a condition-action rule that, conditioned by the query that ensures that a new case of the specific net can be activated, initialises the relations of the local data store and activates the start place.

The rule makes use of a 0-ary service call **newId** that returns a fresh value for the process identifier (see A.4).

The action includes the effects of the initialisation for the local relations and the update of the *Marking* relation.

Let $w \in \mathcal{W}$ be a DP-net in the model, $\mathcal{S}(w) = \textsc{act}_w(\vec{c}) : \{e_1, \ldots, e_m\}$ its initialisation action, and $start$ its start place. The corresponding rule is:

$$\sigma(w)(\vec{c})^{\wp(i)} \mapsto \left\{ \begin{array}{l} e_1^{\wp(\textbf{newId})}, \ldots, e_m^{\wp(\textbf{newId})} \\ \text{true} \rightsquigarrow \textbf{add } \mathit{Marking}(\textbf{newId}, start) \end{array} \right\}$$

Termination of a completed case For each net in $\mathcal{W}$ there is a condition-action rule which is conditioned by the query over the *Marking* relation verifying that the end place of the net is active and the process id as free variable. The action – with the process id as parameter, bound to the result of the condition free variable – includes the effects for the update of the shared database relations (using the process id) and the removal of all the *Marking* tuples corresponding to the given process id. As the activation of a new case the update effects over the shared database are modified according to their kind. PID of terminated process is added to the relation *Active*, among the candidates for archival.

Let $w \in \mathcal{W}$ be a DP-net in the model, $\mathcal{E}(w) = \textsc{act}_w() : \{e_1, \ldots, e_m\}$ its shared update action, and end its end place. The corresponding rule is:

$$\mathit{Marking}(i, end) \mapsto \left\{ \begin{array}{l} e_1^{\wp(i)}, \ldots, e_m^{\wp(i)} \\ \mathit{Marking}(i, p) \rightsquigarrow \textbf{del } \mathit{Marking}(i, p) \\ \text{true} \rightsquigarrow \textbf{add } \mathit{Active}(i) \end{array} \right\}$$

Archival of data Archival of tuples is based on a two phases process: a c-a rule nondeterministically moves process ids from *Active* to *ToArchive*, while a second rule moves all the tuples marked with ids in *ToArchive* from shared to archive relations. These two steps are necessary because it might be the case that archiving the tuples of a single case would violate constraints that would be satisfied if were archived those generated by more than a case simultaneously.

The archiving step succeeds only if constraints on both shared and archive are satisfied.

Let $R_1, \ldots, R_k$ be the relations in $\mathcal{D}_G$ and $R_1^a, \ldots, R_k^a$ the corresponding relations in the archive database. The ac-rules for archival are:

$$\mathit{Active}(i) \mapsto \{\text{true} \rightsquigarrow \textbf{add } \mathit{ToArchive}(i), \text{true} \rightsquigarrow \textbf{del } \mathit{Active}(i)\}$$

$$\text{true} \mapsto \left\{ \begin{array}{l} \mathit{ToArchive}(i) \wedge R_1(\vec{y})^{\wp(i)} \rightsquigarrow \{\textbf{del } R_1(\vec{y})^{\wp(i)}, \textbf{add } R_1^a(\vec{y})^{\wp(i)}\} \\ \vdots \\ \mathit{ToArchive}(i) \wedge R_k(\vec{y})^{\wp(i)} \rightsquigarrow \{\textbf{del } R_k(\vec{y})^{\wp(i)}, \textbf{add } R_k^a(\vec{y})^{\wp(i)}\} \\ \mathit{ToArchive}(i) \rightsquigarrow \textbf{del } \mathit{ToArchive}(i) \end{array} \right\}$$

4.3. Correctness and Completeness of DCDS Encoding

The main idea behind the proof is to define a correspondence between RAW-SYS snapshots and DCDS states and showing that:

1. integrity constraints are satisfied in snapshots iff their translation is satisfied in the corresponding state
2. for every transaction between snapshots there is a corresponding pair in the transition relation between states, and vice versa

The above proof plan is complicated by the fact that archival is split in a sequence of steps that first generate the set of ids to archive and finally the actual archival step. From the

RAW-SYS to DCDS it's easy, just sequence the right selection and then the archival. In the other direction there are DCDS transitions (the selection) that do not correspond to RAW-SYS transitions. Idea: show the property for non-selection transitions and then show that selection doesn't alter the outcome.

In the following we consider an arbitrary RAW-SYS model $\mathcal{R} = \langle \mathcal{D}_C, \mathcal{W}, \mathcal{S}, \mathcal{E}, \sigma \rangle$, and the corresponding DCDS transition system $\Upsilon_{\mathcal{S}} = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$ defined by the translation of Section 4.

Definition 10. *Let $\mathcal{S} = \langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C} \rangle$ be a snapshot of $\mathcal{R}$, then the corresponding DCDS database state $\Xi(\mathcal{S})$ is defined as following:*

1. *for each relation R in $\mathcal{I}_S$, and tuple $\vec{t} \in R$ with provenance i, the tuple $\vec{t}^{\wp(i)}$ is in $R^{\wp}$. No other tuples are in $R^{\wp}$.*
2. *for each relation R in $\mathcal{I}_A$, and tuple $\vec{t} \in R$ with provenance i, the tuple $\vec{t}^{\wp(i)}$ is in $R^{\wp}$. No other tuples are in $R^{\wp}$.*
3. *for each case $\langle i, w, (M, I) \rangle \in \mathcal{C}$:*
   - *for each relation R in $\mathcal{I}$, and tuple $\vec{t} \in R$, the tuple $\vec{t}^{\wp(i)}$ is in $R^{\wp}$. No other tuples are in $R^{\wp}$;*
   - *for each place p in the domain of M the tuple $\langle i, p \rangle \in$ Marking iff $M(p) > 0$;*
4. *relations ToArchive and Archived are empty.*

Let $db(s)$ for some $s \in \Sigma$, then the corresponding RAW-SYS snapshot $\langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C} \rangle = \Xi^{-}(db(s))$ is defined as following:

1. *for each relation R in the schema of $\mathcal{I}_S$, $R^{\wp} \in db(s)$, and tuple $\vec{t}^{\wp(i)} \in R^{\wp}$, the tuple $\vec{t}$ is in $R \in \mathcal{I}_S$ with provenance i. No other tuples are in R.*
2. *for each relation R in the schema of $\mathcal{I}_A$, $R^{\wp} \in db(s)$, and tuple $\vec{t}^{\wp(i)} \in R^{\wp}$, the tuple $\vec{t}$ is in $R \in \mathcal{I}_A$ with provenance i. No other tuples are in R.*
3. *for each i s.t. exists $\langle i, p \rangle \in$ Marking there is a case $\langle i, w, (M, I) \rangle \in \mathcal{C}$ s.t.:*
   - *w is the DP-net corresponding to the place p;*[7]
   - *for each relation R in the schema of $\mathcal{I}$, $R^{\wp} \in db(s)$, and tuple $\vec{t}^{\wp(i)} \in R^{\wp}$, the tuple $\vec{t}$ is in $R \in \mathcal{I}$. No other tuples are in R;*
   - *for each p' place in w, $M(p') = 1$ if $\langle i, p' \rangle \in$ Marking and $M(p') = 0$ otherwise.*

   no other cases are in $\mathcal{C}$.

First we show that the translation of queries is preserved across the translation between the two systems.

Lemma 1. *Let q be a query over local or shared databases of $\mathcal{R}$, then for each snapshot $\mathcal{S} = \langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C} \rangle$ the query is satisfied iff its translation is satisfied in $d = \Xi(\langle \mathcal{I}_S, \mathcal{I}_A, \mathcal{C} \rangle)$.*

Proof. Let's consider first the case that q is a query over the shared database.

If q is false on $\mathcal{I}_S$ then $q^{\wp}$ must be false in d because all the process ids are projected away therefore its non-satisfiability would imply non-satisfiability of q in $\mathcal{I}_S$.

On the other hand, if $q^{\wp}$ is false in d, then q must be false in $\mathcal{I}_S$ because its relations are the ones in d where the process ids are projected away.

An analogous argument can be stated for the archive instance $\mathcal{I}_A$.

[7] We assumed distinct places, so each one can be associated to a single network.

If q is a query over a local database, then in $q^{\wp}$ the process id argument is universally quantified over all the process ids. Therefore if q is false in all the local databases, then it is false in the union of all the databases parameterised by the pids. On the other hand, if $q^{\wp}$ is false in the union, then q must be false in every partition w.r.t. the pids. □

Now we can relate trajectories in R and in its translation and shows that that they can be replayed in each other in such a way that any query satisfied in the final snapshot of one are satisfied in the final state of the translation and vice-versa.

Lemma 2.

1. *Let $s_0, \ldots, s_n$ be a trajectory in $\mathcal{R}$, then there exists a trajectory $s'_0, \ldots, s'_k$ in $\Xi(\mathcal{R})$ such that $s'_k = \Xi(s_n)$.*
2. *Let $s'_0, \ldots, s'_k$ be a trajectory in $\Xi(\mathcal{R})$, then then there exists a trajectory $s_0, \ldots, s_n$ in $\mathcal{R}$ s.t. $s_n = \Xi^{-}(s'_k)$.*

Proof. We prove both items in the lemma by induction on the length of the trajectories.

For trajectories of length 1 both items are satisfied by construction because both snapshots and states are the initial ones.

1. Let $s_0, \ldots, s_n, s_{n+1}$ be a trajectory in $\mathcal{R}$ and $s'_0, \ldots, s'_k$ in $\Xi(\mathcal{R})$ such that $s'_k = \Xi(s_n)$. The transition from s_n to s_{n+1} is among the ones in Def 8. All but the last (archival) are in one to one mapping with DCDS ca-rule defined in Section 4.2. The behaviour of the transitions is defined by means of DCDS actions and the control – including activation and termination of cases – is implemented by appropriate effects on the *Marking* relation.

   Archival transition on the other hand is implemented by a sequence of ca-rules. We can assume that *ToArchive* relation is empty and terminated pids are in the *Active* relation. If that is not the case then transitions corresponding to ca-rules

   $$\mathit{Active}(i) \mapsto \{\text{true} \rightsquigarrow \textbf{add } \mathit{ToArchive}(i), \text{true} \rightsquigarrow \textbf{del } \mathit{Active}(i)\}$$

   can be removed from $s'_0, \ldots, s'_k$ without affecting the equality $s'_k = \Xi(s_n)$ because those relations are not considered in the conversion. Let $i_1, \ldots, i_\ell$ be the pids of tuples archived by the s_n to s_{n+1} transition. Then the sequence of ca-rules

   $$\begin{aligned}
   &\mathit{Active}(i_1) \mapsto \{\text{true} \rightsquigarrow \textbf{add } \mathit{ToArchive}(i_1), \text{true} \rightsquigarrow \textbf{del } \mathit{Active}(i_1)\}\\
   &\qquad\vdots\\
   &\mathit{Active}(i_\ell) \mapsto \{\text{true} \rightsquigarrow \textbf{add } \mathit{ToArchive}(i_\ell), \text{true} \rightsquigarrow \textbf{del } \mathit{Active}(i_\ell)\}\\
   &\qquad\text{true} \mapsto \left\{\begin{array}{l}
   \mathit{ToArchive}(i) \wedge R_1(\vec{y})^{\wp(i)} \rightsquigarrow \{\textbf{del } R_1(\vec{y})^{\wp(i)}, \textbf{add } R_1^a(\vec{y})^{\wp(i)}\}\\
   \vdots\\
   \mathit{ToArchive}(i) \wedge R_k(\vec{y})^{\wp(i)} \rightsquigarrow \{\textbf{del } R_k(\vec{y})^{\wp(i)}, \textbf{add } R_k^a(\vec{y})^{\wp(i)}\}\\
   \mathit{ToArchive}(i) \rightsquigarrow \textbf{del } \mathit{ToArchive}(i)
   \end{array}\right\}
   \end{aligned}$$

   generates a sequence of additional states $s'_{k+1}, \ldots, s'_{k+\ell}, s'_{k+\ell+1}$ where $s'_{k+\ell+1} = \Xi(s_{n+1})$.
2. Let $s'_0, \ldots, s'_k, s'_{k+1}$ be a trajectory in $\Xi(\mathcal{R})$, and $s_0, \ldots, s_n$ in $\mathcal{R}$ s.t. $s_n = \Xi^{-}(s'_k)$.

   If the transition from s'_k to s'_{k+1} is different from the selection of process id

   $$\mathit{Active}(i) \mapsto \{\text{true} \rightsquigarrow \textbf{add } \mathit{ToArchive}(i), \text{true} \rightsquigarrow \textbf{del } \mathit{Active}(i)\}$$

   then it corresponds to the translation of one of the RAW-SYS transitions in Def. 8 and its application generates a snapshot s_{n+1} s.t. $s_{n+1} = \Xi^{-}(s'_{k+1})$.

   On the other end, if the transition is the selection of process id, then $\Xi^{-}(s'_{k+1}) = \Xi^{-}(s'_k) = s_n$; therefore the trajectory $s_0, \ldots, s_n$ satisfies the property.

□

The following theorem follows from the previous lemmata and shows that reachability problems defined over RAW-SYS can be solved by translating them into an appropriate DCDS problem.

Theorem 5. *For each RAW-SYS model $\mathcal{R}$, any reachability problems defined by means of safe range queries over $\mathcal{R}$ can be translated into a reachability problem over $\Xi(\mathcal{R})$.*

Proof. Let q be a safe range query over $\mathcal{R}$, we consider the problem of verifying whether there is a trajectory of $\mathcal{R}$ s.t. in its final snapshot q is satisfied.

If there is a trajectory $s_1, \ldots, s_n$ of $\mathcal{R}$ s.t. q is satisfied in s_n, by Lemma 2, there is a trajectory $s'_0, \ldots, s'_k$ in $\Xi(\mathcal{R})$ such that $s'_k = \Xi(s_n)$, and by Lemma 1 q is satisfied in s_n iff $q^{\wp}$ is satisfied in s'_k. On the other end, if there is a trajectory $s'_0, \ldots, s'_k$ in $\Xi(\mathcal{R})$ s.t. $q^{\wp}$ is satisfied in s'_k then there is a trajectory $s_1, \ldots, s_n$ of $\mathcal{R}$ s.t. $s'_k = \Xi(s_n)$, therefore q is satisfied in s_n by Lemma 1. □

5. Verification of DCDS using an action language

5.1. Data component $\mathcal{D} = \langle \Delta, \mathcal{R}, \mathcal{C}, \mathcal{I}_0 \rangle$

- Once the bound b for the specific DCDS has been established, the finite part of Δ of size b, let us call it Δ' should go in the background knowledge. Δ is such that $\textsc{adom}(\mathcal{I}_0) \subseteq \Delta'$ and also Δ' contains all constants appearing elsewhere in the system (such as, in actions).

- For each $R \in \mathcal{R}$ of arity n we have a fluent declaration:

  $$p(X_1, \ldots, X_n) \quad \text{(no } \texttt{requires} \text{ part)}$$

  Fluents corresponding to database relations do not change unless updated:
  inertial $p(X_1, \ldots, X_n)$.

- For each (safe range) query $Q(x_1, \ldots, x_k)$ appearing in the model (both in data and process component) we have a fluent declaration

  $$\rho_Q(X_1, \ldots, X_n) \quad \text{(no } \texttt{requires} \text{ part)}$$

  and a set of causation rules of the form
  caused $head$ if $body$.

  corresponding to the translation in clausal form of the query Q.[8]

- Each denial constraint $c \in \mathcal{C}$ introduces a causation rule of the form:
  caused false if c.

- $\mathcal{I}_0$ is translated as the initial state of the system. Hence, for each tuple $\vec{d}$ of relation R in $\mathcal{I}_0$, we have a fact:
  initially: $R(\vec{d})$.

[8] Translation could introduce additional fluents corresponding to the subqueries.

5.2. Process component $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$.

- For each condition-action rule $Q(\vec{p}) \mapsto \text{ACT}(\vec{p}) \in \varrho$ an executability rule is introduced:
  executable $\text{ACT}(\vec{\mathsf{P}})$ if $\rho_Q(\vec{\mathsf{P}})$.

- For each action $\text{ACT}(\vec{\mathsf{p}}) : \{e_1, \ldots, e_m\} \in \mathcal{A}$ we can assume without loss of generality that each effect e_i is in one of the two form:

$$Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow \textbf{add } R(\vec{\mathsf{p}}, \vec{x})$$
$$Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow \textbf{del } R(\vec{\mathsf{p}}, \vec{x})$$

  Where R is a single predicate. According to DCDS semantics addition of tuples takes the precedence over deletions, therefore we need to make sure that unnecessary deletions are not applied. To this end we use "default" rules, but we need to introduce an auxiliary predicate to avoid unwanted interactions with the inertial rules. For each relation R we consider a (non-inertial) predicate R^δ which intuitively holds the actual updates that must be applied to R.

  Add and deletes are defined over R^δ and the first kind of actions introduces causation rules like
  caused $R^\delta(\vec{\mathsf{P}}, \vec{X})$ after $\rho_Q(\vec{\mathsf{P}}, \vec{X})$, $\text{ACT}(\vec{\mathsf{P}})$.

  while the second
  caused $-R^\delta(\vec{\mathsf{P}}, \vec{X})$ if not $R^\delta(\vec{\mathsf{P}}, \vec{X})$ after $\rho_Q(\vec{\mathsf{P}}, \vec{X})$, $\text{ACT}(\vec{\mathsf{P}})$.

  the negated part in the body of the rule makes sure that in the tuple is added the deletion is not applied. The actual relation R is "updated" by means of R^δ:
  caused $R(\vec{X})$ if $R^\delta(\vec{X})$.
  caused $-R(\vec{X})$ if $-R^\delta(\vec{X})$.

  Skolem functions
  All occurrences of nondeterministic functions are replaced with fluents with an additional argument representing the value of the function, and functional constraints to ensure that there's a single value for the given tuple of arguments.

  Let $f(\vec{x})$ be a Skolem term of arity n in $\mathcal{F}$, then we introduce the fluents ρ_f of arity $n+1$ and $\rho_f^\emptyset$ of arity n to enforce non-emptiness. We also assume a typing predicate adom/1 (not a fluent) holding the active domain for the Skolem constants.

  If an **add** (or **del**) term include includes Skolem functions, these are substituted by a new fresh variable in the head and the previously introduced predicate in the body, e.g.:
  caused $R^\delta(\vec{\mathsf{P}}, \vec{X}, X_f)$ if $\rho_f(\vec{\mathsf{P}}, \vec{X}, X_f)$ after $\rho_Q(\vec{\mathsf{P}}, \vec{X})$, $\text{ACT}(\vec{\mathsf{P}})$.

  and analogously for **del** updates.

  We need to make sure that the "functional" predicate ρ_f is defined for the appropriate tuple (the one selected by $\rho_Q(\vec{\mathsf{P}}, \vec{X})$ and $\text{ACT}(\vec{\mathsf{P}})$). We achieve that by introducing a non-deterministic choice over the last argument of ρ_f:
  caused $\rho_f(\mathsf{X}_1, \ldots, \mathsf{X}_n, \mathsf{Y})$ if adom(Y), not $-\rho_f(\mathsf{X}_1, \ldots, \mathsf{X}_n, \mathsf{Y})$ after $\rho_Q(\vec{\mathsf{P}}, \vec{X})$, $\text{ACT}(\vec{\mathsf{P}})$.
  caused $-\rho_f(\mathsf{X}_1, \ldots, \mathsf{X}_n, \mathsf{Y})$ if adom(Y), not $\rho_f(\mathsf{X}_1, \ldots, \mathsf{X}_n, \mathsf{Y})$ after $\rho_Q(\vec{\mathsf{P}}, \vec{X})$, $\text{ACT}(\vec{\mathsf{P}})$.

  and making sure that at least a value is selected for the necessary tuples
  $\rho_f^\emptyset(\mathsf{X}_1, \ldots, \mathsf{X}_n) :- \rho_f(\mathsf{X}_1, \ldots, \mathsf{X}_n, \mathsf{Y})$.
  caused false if not $\rho_f^\emptyset(\vec{\mathsf{P}}, \vec{X})$ after $\rho_Q(\vec{\mathsf{P}}, \vec{X})$, $\text{ACT}(\vec{\mathsf{P}})$.

  Moreover we must impose a functional dependency of the last argument wrt the other ones:

caused false if $\rho_f(X_1, \ldots, X_n, Y)$, $\rho_f(X_1, \ldots, X_n, Z)$, Z != Y.

Note that some of these rules do not depend on the actual action.

5.3. Verification tasks

The verification of reachability properties of the model is encoded with a query describing the goal state for the planner. These properties can be related to the state of the (global) database as well as more general conditions over the dynamic of the system; e.g. the verification that there are no running processes can be performed by checking that no places are in the *Marking* relation (i.e. there are no tokens).

5.4. Correctness and Completeness of DCDS Encoding

The correctness and completeness of our encoding is based on the results presented in [10] where reachability problem is encoded into an ADL planning problem [14].

Although $\mathcal{K}$ language is strictly more expressive than ADL, therefore the encoding proposed in the paper can be directly used, we take advantage of its expressiveness to optimise the encoding.

In particular, we improved the following aspects:

- in $\mathcal{K}$ integrity constraints can be encoded as rules, without the need of including additional actions interleaved with the ones corresponding to DCDS ca-rules;
- functional symbols are simulated by $\mathcal{K}$ non-determinism leveraging stable models semantics of $\mathcal{K}$ rules, therefore they can be directly used in the DCDS actions instead of limiting to ca-rule parameters.

The above improvements do not affect the structure of the proofs backing the results presented in [10] which are still applicable to our translation into $\mathcal{K}$.[9]

6. Example formalization

We resume the example described in Section 2 by assuming to have only the $\mathbb{RB}$ process. Hence, the shared database essentially contains the same schema of the local ones, but in the general case it will not be the case. $\mathcal{M} = \langle \mathcal{D}_G, \mathcal{W}, \mathcal{S}, \mathcal{E}, \sigma \rangle$ where:

- $\mathcal{D}_G = \mathbb{Empl}(empl)$, $\mathbb{Dest}(dest)$, $\mathbb{Pending}(empl, dest)$, $\mathbb{Accepted}(empl, dest, amount)$, $\mathbb{Rejected}(empl, Dest)$;
- $\mathcal{W} = \langle W \rangle$;
- σ is such that $\sigma(W) = \textsc{startW}$;
- $\mathcal{E}$ is such that $\mathcal{E}(W) = \textsc{endW}$;

The workflow net with data $W = \langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ where:

- $\mathcal{D} = Pending(empl, dest)$, $CurrReq(empl, dest, status)$, $TrvlMaxAmnt(maxAmnt)$, $TrvlCost(cost)$;
- P, T, F are as in Figure 2;
- $\mathcal{F}$ is such that:

[9] Full details and proofs can be found on the document available at `https://www.dropbox.com/sh/q1scvtz278xyzfb/AACTWp8vslOlVsWjaWsPmNTVa/main.pdf`

- $\mathcal{F}(reviewReq) = \{\textbf{maxAmnt}(), \textbf{status}()\}$
- $\mathcal{F}(fillReimb) = \{\textbf{cost}()\}$

- $\mathcal{A}$ is such that:
  - $\mathcal{A}(reviewReq) = \textsc{reviewReq}$
  - $\mathcal{A}(fillReimb) = \textsc{fillReimb}$
  - $\mathcal{A}(reviewReimb) = \textsc{reviewReimb}$

- $\mathcal{G}$ is such that:
  - $\mathcal{G}(reviewReq) = \text{true}$
  - $\mathcal{G}(fillReimb) = \exists e, d.CurrReq(e, d, \text{accepted})$
  - $\mathcal{G}(reviewReimb) = \text{true}$

6.1. Actions

$$\textsc{startW}() : \left\{ \begin{array}{lcll} \mathbb{P}\text{ending}(e,d) & \rightsquigarrow & \textbf{add} & \{Pending(e,d)\} \\ & & & \\ true & \rightsquigarrow & \begin{array}{l}\textbf{add}\\ \textbf{del}\end{array} & \begin{array}{l}\{CurrReq(\textbf{empl}(), \textbf{dest}(), \text{submttd})\}\\ \{\mathbb{P}\text{ending}(\textbf{empl}(), \textbf{dest}())\}\end{array} \end{array} \right\}$$

$$\textsc{rvwReq}() : \left\{ \begin{array}{lcll} CurrReq(e,d,s) & \rightsquigarrow & \textbf{del} & \{CurrReq(e,d,s)\} \\ & & \textbf{add} & \left\{ \begin{array}{l} CurrReq(e,d,\textbf{status}()), \\ TrvlMaxAmnt(\textbf{maxAmnt}()) \end{array} \right\} \end{array} \right\}$$

$$\textsc{fillRmb}() : \{ \text{true} \rightsquigarrow \textbf{add} \{TrvlCost(\textbf{cost}())\} \}$$

$$\textsc{revwReimb}() : \left\{ \begin{array}{lcll} \begin{array}{l} CurrReq(e,d,s) \wedge \\ Cost(c) \wedge \\ TrvlMaxAmnt(ma) \wedge \\ c \leq ma \end{array} & \rightsquigarrow & \begin{array}{l}\textbf{del}\\ \textbf{add}\end{array} & \begin{array}{l}\{CurrReq(e,d,s)\}\\ \{CurrReq(e,d,\text{reimbursed})\}\end{array} \\ & & & \\ \begin{array}{l} CurrReq(e,d,s) \wedge \\ Cost(c) \wedge \\ TrvlMaxAmnt(ma) \wedge \\ c > ma \end{array} & \rightsquigarrow & \begin{array}{l}\textbf{del}\\ \textbf{add}\end{array} & \begin{array}{l}\{CurrReq(e,d,s)\}\\ \{CurrReq(e,d,\text{rejected})\}\end{array} \end{array} \right\}$$

$$\textsc{endW}() : \left\{ \begin{array}{lcll} \begin{array}{l} CurrReq(e,d,\text{reimbursed}) \\ \wedge TrvlCosts(c) \end{array} & \rightsquigarrow & \textbf{add} & \{\mathbb{A}\text{ccepted}(e,d,c)\} \\ & & & \\ \begin{array}{l} CurrReq(e,d,\text{rejected}) \\ \wedge TrvlMaxAmnt(ma) \end{array} & \rightsquigarrow & \textbf{add} & \{\mathbb{R}\text{ejected}(e,d)\} \end{array} \right\}$$

Appendices

A. Data-Centric Dynamic Systems

In this section, we provide an overview of Data-Centric Dynamic Systems (DCDSs). More specifically, we introduce a variant of the original DCDS framework (first introduced in [5]). In this variant, actions are described following the action formalism of [21], which provides STRIPS-like abstractions on top of the original DCDS action formalism. This variant is expressively equivalent to the original one [21].

A.1. The DCDS Framework

A DCDS $\mathcal{S}$ is a pair $\langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D}$ is the data component of $\mathcal{S}$, and $\mathcal{P}$ is its process component.

Data component. The data component is a full-fledged relational database with constraints. Technically, $\mathcal{D} = \langle \Delta, \mathcal{R}, \mathcal{C}, \mathcal{I}_0 \rangle$, where:

- Δ is a countably infinite set of constants.
- $\mathcal{R}$ is a *database schema*, i.e., a set of relation schemas. We will equivalently adopt the positional notation or the attribute-based named notation for relations; When the latter notation is used, an n-ary relation R is represented as $R(U)$, where U is a set of n named attributes. Furthermore, given a set $A \subseteq U$ of attributes, $R[A]$ represents the projection of R over A.
- $\mathcal{C}$ is a set of *domain-independent FO-constraints* over $\mathcal{R}$, capturing the real-world constraints of the targeted application domain.
- $\mathcal{I}_0$ is the *initial database instance* of $\mathcal{S}$, i.e., a database instance conforming to $\mathcal{R}$, satisfying the constraints $\mathcal{C}$, and made of values in Δ.

Among all possible FO-constraints, we consider the following specific constraints, which are widespread in database modelling and standard conceptual modelling languages such as UML class diagrams, E-R diagrams, and the ORM notation:

- Standard *key* and *primary key* constraints. We use notation KEY($R[A]$) (resp., $\underline{R}A$) to model that the set A of attributes is a key (resp., primary key) for relation R.
- Standard *foreign key* constraints. We use notation $R[A] \longrightarrow S[B]$ to model that the set A of attributes in R is a foreign key pointing to the set B of attributes in S, such that $\underline{S}B$ holds.
- *Cardinality-constraints*, which resemble cardinality/frequency constraints of conceptual modelling languages. Cardinality-constraints generalize key constraints by bounding the minimum and maximum number of tuples allowed in a relation when the value of some attributes is maintained unaltered. Given a relation $R(U)$ and a set $A \subseteq U$ of attributes, notation CARD($R[A], m..n$), where m and n are positive integers, denotes the *cardinality constraint* requiring that the number of R-tuples with the same values for attributes A ranges between m and n.
- A combination of cardinality and foreign key constraints, where a foreign key has an associated cardinality constraint guaranteeing that the number of tuples pointing to the same target primary key is bounded. We denote by $A[R] \xrightarrow{m..n} B[S]$ the database constraint corresponding to the conjunction of $A[R] \longrightarrow B[S]$ and CARD($A[R], m..n$), and call such a conjunction a *cardinality-bounded foreign key* constraint. Notice that $A[R] \xrightarrow{1..1} B[S]$ is equivalent to the combination of KEY($R[A]$) and $A[R] \longrightarrow B[S]$.

All these constraint types can be easily encoded as FO formulae. Some examples will be shown in Section A.3.

Process component. The process component $\mathcal{P}$ defines the progression mechanism for the DCDS. It is constituted by a process, which queries the current data maintained by $\mathcal{D}$ and determines which actions are executable, and with which parameters; parameterised actions,

in turn, query and update $\mathcal{D}$, possibly introducing new values from the external environment, by issuing service calls. Technically, $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where

- $\mathcal{F}$ is a finite set of *functions*, each representing the interface to a (nondeterministic) *external service*;
- $\mathcal{A}$ is a finite set of *actions*, whose execution updates the data component, and may involve external service calls;
- ϱ is a finite set of *condition-action rules* that form the specification of the overall *process*, which tells at any moment which actions can be executed.

Actions. An *action* of $\mathcal{A}$ is an expression $\textsc{act}(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$, where:

- $\textsc{act}(p_1, \ldots, p_n)$ is the action *signature*, constituted by a name $\textsc{act}$ and a sequence $p_1, \ldots, p_n$ of *parameters*, to be substituted with values when the action is invoked;
- $\{e_1, \ldots, e_m\}$, also denoted as $\textsc{effect}(\textsc{act})$, is a set of *effects*, which are assumed to take place simultaneously.

Each effect e_i has the form

$$Q(\vec{p}, \vec{x}) \rightsquigarrow \textbf{add } A \textbf{ del } D$$

where:

- Q is a domain independent FO query over $\mathcal{R}$ whose terms are variables, action parameters, and constants from $\mathcal{I}_0$. Intuitively, Q selects the tuples to instantiate the effect with. During the execution, the effect is applied with a ground substitution $\vec{d}$ for the action parameters, and for every answer θ to the query $Q(\vec{d}, \vec{x})$.
- A is a set of facts over $\mathcal{R}$, which include as terms: free variables $\vec{x}$ of Q, action parameters $\vec{p}$ and/or Skolem terms $f(\vec{x}', \vec{p}')$ (with $\vec{x}' \subseteq \vec{x}$, and $\vec{p}' \subseteq \vec{p}$). We use $\textsc{skolem}(A)$ to denote all Skolem terms mentioned in A. At runtime, whenever a ground Skolem term is produced by applying substitution θ to A, the corresponding service call is issued, replacing it with the result (from Δ) returned by the invoked service. The ground set of facts so obtained is *added* by the DCDS to its current database instance.
- D is also a set of facts over $\mathcal{R}$, which include as terms free variables $\vec{x}$ of Q and action parameters $\vec{p}$. At runtime, the ground facts obtained by applying substitution θ to D are *removed* from the current database instance.

As in STRIPS, we assume that additions have higher priority than deletions (i.e., if the same fact is asserted to be added and deleted during the same execution step, then the fact is added). The "**add** A" part (resp., the "**del** D" part) can be omitted if $A = \emptyset$ (resp., $D = \emptyset$).

Process. The *process* ϱ is a finite set of *condition-action rules*, each of the form $Q(\vec{x}) \mapsto \textsc{act}(\vec{x})$, where $\textsc{act}$ is an action in $\mathcal{A}$ and Q is again a FO query over $\mathcal{R}$ whose free variables are exactly the parameters of $\textsc{act}$, and whose other terms can be quantified variables or constants mentioned in $\mathcal{I}_0$.

Finally, notice that effects and condition-action rules can be rearranged in a *modular way*, by observing that:

- A single effect of the form

  $$Q(\vec{p}, \vec{x}) \rightsquigarrow \textbf{add } A \textbf{ del } D$$

  can be equivalently re-expressed as a set of effects

  $$\begin{array}{rcl} Q(\vec{p}, \vec{x}) & \rightsquigarrow & \textbf{add } A_1 \textbf{ del } D_1 \\ & \cdots & \\ Q(\vec{p}, \vec{x}) & \rightsquigarrow & \textbf{add } A_n \textbf{ del } D_n \end{array}$$

  where some A_i or D_i could possibly be $\emptyset$, and we have that $A = \bigcup_{i \in \{1,\ldots,n\}} A_i$ and $D = \bigcup_{i \in \{1,\ldots,n\}} D_i$.

- Unions in condition-action rules can be implicitly obtained by composing multiple rules, since a single rule of the form

$$\bigvee_{i\in\{1,\ldots,n\}} Q_i(\vec{x}) \mapsto \text{ACT}(\vec{x})$$

can be equivalently re-expressed as a set of rules

$$Q_1(\vec{x}) \mapsto \text{ACT}(\vec{x}) \qquad \cdots \qquad Q_n(\vec{x}) \mapsto \text{ACT}(\vec{x})$$

This equivalent rearrangement will be useful for the class of DCDSs introduced in Section 3, for which add and delete facts are required to obey to some restrictions.

A.2. Execution semantics

The execution semantics of a DCDS $\mathcal{S}$ is a possibly infinite transition system $\Upsilon_{\mathcal{S}}$ whose states are labeled by database instances. It represents all possible computations that the process component can do on the data component. Specifically, $\Upsilon_{\mathcal{S}} = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where: *(i)* Σ is a set of states; *(ii)* $s_0 \in \Sigma$ is the initial state; *(iii)* db is a function that, given a state $s \in \Sigma$, returns the database instance of s, which is made of values in Δ and conforms to $\mathcal{R}$ and $\mathcal{C}$; *(iv)* $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation over states.

Given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{D} = \langle \Delta, \mathcal{R}, \mathcal{C}, \mathcal{I}_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, the transition system $\Upsilon_{\mathcal{S}}$ is intuitively constructed as follows. Starting from $\mathcal{I}_0$, all condition-action rules in ϱ are evaluated, determining which actions are executable, and with which ground parameter assignments. Non-deterministically, one such action ACT with parameter assignment ρ is selected and executed over $\mathcal{I}_0$. To do so, every effect e of ACT (partially grounded with the parameter assignment ρ) is evaluated, by calculating all the answers of its left-hand side, and grounding the right-hand side accordingly. If the right-hand side of e contains service calls, they are issued, receiving back for each of them a value nondeterministically chosen from Δ. This value is then used to substitute the service call with the actual result; notice that, within an execution step, multiple occurrences of the same service call are substituted with the same value. The overall set of ground facts obtained by evaluating all effects of ACTρ in this way finally constitutes the next database instance. Notice that upon the execution of an action, the content of a relation is lost unless it is explicitly maintained through dedicated effects of the action. The transition system construction then proceeds by constructing all possible successors, each of which is obtained by selecting one of the executable actions with parameters, and one result for each of the involved service calls. The construction then recursively proceeds over such newly generated states. For a formal description of the execution semantics, see [5].

A.3. Robin Hood and the Archery Training Process

We now introduce a simple DCDS that summarizes all the key ingredients that will be discussed in the remainder of the paper.

Robin Hood is a renown archer, and needs an information system to keep track of the archery courses he delivers to his apprentices among the merry men. To this end, he creates a DCDS $\mathcal{S}_{rh}$ that supports him in maintaining the information of interest, and manipulate it over time.

The schema and constraints of $\mathcal{S}_{rh}$ are listed in Figure 3. As for the schema:

- $Group(id)$ states that there is an archery group identified by id.
- $Meets(weekSlot, group, where)$ indicates that the weekly time slot $weekSlot$ is dedicated to the training of $group$ in the location specified by the code $where$.
- $MerryM(id, name, birthdate, combatLevel, group)$ states that the person identified by id is a merry man named $name$ and born on $birthdate$, who has currently an archery ability corresponding to $combatLevel$, and is enrolled in $group$. We reserve a special constant null to model the case where a person is not enrolled in any group.

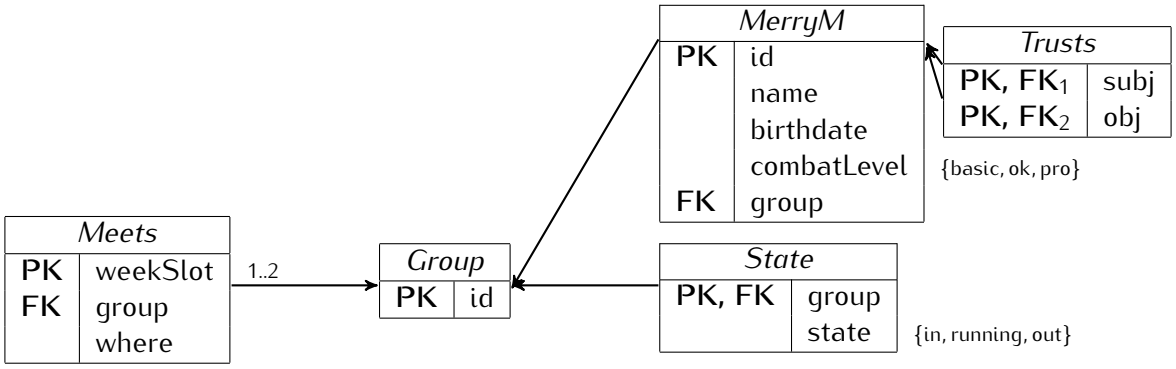

Figure 3: The archery training data component

- *Trusts*$(subj, obj)$ models that merry man $subj$ trusts merry man obj.
- *State* is a relation that glues the data component with the process component, in particular to keep track of the current state of each group—an information that is used to "locate" the group inside the process. Specifically, *State*$(group, s)$ indicates that $group$ is currently in state s, which may be either in (the group is being assembled), running (the group is under training), or out (the group has completed the training).

The schema of Figure 3 is equipped with the following constraints:

- In each state, the *combatLevel* of a merry man is one of three pre-defined levels:

$$\forall id, n, b, c, g.MerryM(id, n, b, c, g) \rightarrow (c = \text{basic} \vee c = \text{ok} \vee c = \text{pro})$$

  Similarly, for group states we have:

$$\forall id, s.State(id, s) \rightarrow (s = \text{in} \vee s = \text{running} \vee s = \text{out})$$

- The first columns of *MerryM*, *State*, and *Meets* are the (primary) keys of the corresponding relations[10]:

$$\forall id, n_1, b_1, c_1, g_1, n_2, b_2, c_2, g_2.$$
$$MerryM(id, n_1, b_1, c_1, g_1) \wedge MerryM(id, n_2, b_2, c_2, g_2) \rightarrow n_1 = n_2 \wedge b_1 = b_2 \wedge c_1 = c_2 \wedge g_1 = g_2$$
$$\forall id, s_1, s_2.State(id, s_1) \wedge State(id, s_2) \rightarrow s_1 = s_2$$
$$\forall s, g_1, w_1, g_2, w_2.Meets(s, g_1, w_1) \wedge Meets(s, g_2, w_2) \rightarrow g_1 = g_2 \wedge w_1 = w_2$$

- The two attributes of *Trusts* reference both a merry man. There are therefore two foreign key constraints, formalized as:

$$\forall s, o.Trusts(s, o) \rightarrow \exists n, b, c, g.MerryM(s, n, b, c, g)$$
$$\forall s, o.Trusts(s, o) \rightarrow \exists n, b, c, g.MerryM(o, n, b, c, g)$$

  Similarly for the foreign key starting from the *State* relation.
- The foreign key starting from the *MerryM* relation does not start from an attribute that is part of the primary key for the source relation. Hence, differently from the previous case, the FO formalization needs to consider also the fact that the attribute is nullable:

$$\forall id, n, b, c, g.MerryM(id, n, b, c, g) \rightarrow g = \text{null} \vee Group(g)$$

[10]Thanks to set semantics, there is no need to explicitly encode that the only column of *Group* is its primary key, and similarly for the combination of the only two columns of *Trusts*.

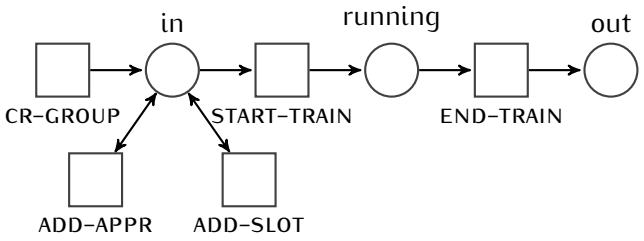

Figure 4: The archery training process control-flow. Tokens denote process cases, which in this example correspond to groups of people

- *Meets* has a cardinality-bounded foreign key pointing to *Group*, of the form $Meets[group] \xrightarrow{1..2} Group[id]$. This can be formalized in FOL as:

$$\begin{aligned}&\forall s, g, w.Meets(s, g, w) \rightarrow g = \text{null} \vee Group(g)\\&\forall g, s_1, w_1, s_2, w_2, s_3, w_3.\\&\quad Meets(s_1, g, w_1) \wedge Meets(s_2, g, w_2) \wedge Meets(s_3, g, w_3)\\&\qquad \rightarrow s_1 = s_2 \vee s_1 = s_3 \vee s_2 = s_3\end{aligned}$$

  where null is again used to model that the foreign key may be null.

Finally, the data component of $\mathcal{S}_{rh}$ populates the *MerryM* relation with all the merry men that live together with Robin Hood in the Sherwood forest, together with their personal information and trust relations. We also assume that, at the beginning, no group exists, and consequently all merry men have null in the corresponding attribute.

Figure 4 shows a Petri net that sketches the archery training process. Inuitively, places represent group states (and in fact they correspond to the possible values that the second column of relation *State* can take). Transitions correspond to DCDS actions manipulating groups and their related informations, whose executability depends on the group state.

Specifically, the special CR-GROUP action is always executable, and has the effect of creating a new group in the information system, putting it into the in state. The group identifier is injected into the system by calling the **newId** service.

$$\text{true} \mapsto \textsc{cr-group}()$$

$$\textsc{cr-group}() : \left\{ \text{true} \rightsquigarrow \textbf{add} \left\{ \begin{array}{l} Group(\textbf{newId}()), \\ State(\textbf{newId}(), \text{in}) \end{array} \right\} \right\}$$

Robin Hood can add an apprentice to a newly created group, provided that the apprentice is not already enrolled in a group. The effect of the action is to update the $group$ attribute of the selected merry man; this is modeled by removing the current tuple of that merry man, and reinserting it with the updated $group$ attribute.

$$\begin{array}{l} Group(g) \wedge State(g, \text{in}) \wedge \\ \quad \exists n, b, c.MerryM(id, n, b, c, \text{null}) \mapsto \textsc{add-appr}(id, g) \end{array}$$

$$\begin{array}{l} \textsc{add-appr}(\text{m}, \text{g}) : \\ \left\{ \begin{array}{lcll} MerryM(\text{m}, n, b, l, g_o) & \rightsquigarrow & \textbf{del} & \{MerryM(\text{m}, n, b, l, g_o)\} \\ & & \textbf{add} & \{MerryM(\text{m}, n, b, l, \text{g})\} \end{array} \right\} \end{array}$$

At the same time, a newly created group can be updated by providing a weekly slot in which Robin inputs when and where a certain group meets.

$$Group(g) \wedge State(g, \text{in}) \mapsto \textsc{add-slot}(g)$$

$$\textsc{add-slot}(\mathsf{g}):\quad \left\{\mathsf{true} \rightsquigarrow \mathbf{add}\ \{\mathit{Meets}(\mathbf{inWhen}(\mathsf{g}), \mathsf{g}, \mathbf{inWhere}(\mathsf{g}))\}\right\}$$

To model the two user inputs, service calls **inWhen** and **inWhere** are used, both taking as parameter the identifier of the group for which the slot is being created. We can imagine that such service calls are actually realised as a user form that asks Robin Hood to provide the time and location of the group given as input.

Interestingly, although no explicit indication is given in the process control-flow of Figure 4, the combination between such control-flow and the data constraints tells us that at it will be possible to add at most two weekly slots for the same group. This example attests how much involved process analysis becomes once the combination between the data and process component is fully tackled.

A group in the in state can be turned into a running group by executing the START-TRAINING action. This has the effect of making the group not eligible anymore for adding new apprentices.

$$\mathit{Group}(g) \wedge \mathit{State}(g, \mathsf{in}) \mapsto \textsc{start-train}(g)$$

$$\textsc{start-train}(\mathsf{g}):\quad \left\{\mathit{State}(\mathsf{g}, s) \rightsquigarrow \mathbf{del}\{\mathit{State}(\mathsf{g}, s)\}\ \mathbf{add}\{\mathit{State}(\mathsf{g}, \mathsf{running})\}\right\}$$

The end of the training for a running group is marked by executing the END-TRAINING action. Besides the state update for the group, this has a twofold effect:

- the combat level of each group member is updated according to the quality of his performance;
- the group members are dissociated from the group, becoming again free to be enrolled in another group.

$$\mathit{Group}(g) \wedge \mathit{State}(g, \mathsf{running}) \mapsto \textsc{end-train}(g)$$

$$\textsc{end-train}(\mathsf{g}):\quad \left\{\begin{array}{r} \mathit{State}(\mathsf{g}, s) \rightsquigarrow \mathbf{del}\{\mathit{State}(\mathsf{g}, s)\}\ \mathbf{add}\{\mathit{State}(\mathsf{g}, \mathsf{out})\} \\ \mathit{MerryM}(m, n, b, l, \mathsf{g}) \rightsquigarrow \mathbf{del}\{\mathit{MerryM}(m, n, b, l, \mathsf{g})\} \\ \mathbf{add}\{\mathit{MerryM}(m, n, b, \mathbf{assess}(m), \mathsf{null})\} \end{array}\right\}$$

Notice that the combat level assessment is input by Robin Hood for each of the involved merry men. To model such a user input, service call **assess** is used, which takes as parameter the identifier of the merry man to be assessed. We can again imagine this service call to be realised as a user form for Robin. Differently from the weekly slot case, though, the service call result is implicitly subject to the database constraint that enumerates the acceptable values for the *combatLevel* attribute of *MerryM*. This implies that the provided input needs to correspond to one of the three pre-defined levels.

A.4. Fresh Value Injection

A technical, but important, aspect related to DCDSs is that issuing a service call does not guarantee that the obtained result is a fresh value, that is, a value that is not present in the current active domain[11]. In the archery training DCDS of Section A.3, this is perfectly fine with the **assess** service call, which in fact is forced to return one of the three pre-defined combat levels, but is not satisfactory with the **newId** service call. In fact, when creating a new group, the implicit requirement is that the identifier assigned to that group is not already assigned to another group. In this respect, the formalization of the CR-GROUP action is not correct, as it could result in a no-op if **newId** returns an identifier that is already assigned to another idle group in the in state.

[11] Recall that the *active domain* of a database instance is the set of values explicitly appearing in its tuples.

In this section, we show that DCDSs can easily model the injection of a new value that is guaranteed to be fresh w.r.t. the values present in a given column of the current database instance (this can be easily generalized to multiple columns, or even the entire active domain). This is particularly useful in all those cases where a new primary key has to be generated for a certain relation, such as that of CREATE-GROUP.

Let $\mathbf{f}$ be a 0-ary service call, and let R be an n-ary relation. We want to ensure that whenever $\mathbf{f}$ is called, the obtained result is fresh w.r.t. the i-th column of R, i.e., different from all values appearing in the i-th position of R-tuples in the current database instance. This can be guaranteed by modifying the original DCDS specification as follows:

1. The database schema of the original DCDS is augmented with two additional relations: a unary relation $Temp_f$ used to store a copy of the value returned by $\mathbf{f}$, and an n-ary relation R_{prev}, whose extension corresponds to the extension of R in the *previous* state.
2. Each action of the original DCDS is augmented with two additional effects, used to populate R_{prev} in the next state with the current extension of R. This is done by emptying the content of R_{prev}, and filling R_{prev} with the content currently stored by R:

$$\begin{array}{lcll} R_{prev}(\vec{x}) & \rightsquigarrow & \textbf{del} & \{R_{prev}(\vec{x})\} \\ R(\vec{x}) & \rightsquigarrow & \textbf{add} & \{R_{prev}(\vec{x})\} \end{array}$$

3. Every action that employs $\mathbf{f}$ in (the head of) its effects is augmented with an additional effect, which enforces that a copy of the result returned by $\mathbf{f}$ is stored into relation $Temp_f$:

$$\text{true} \rightsquigarrow \textbf{add}\ \{Temp_f(\mathbf{f}())\}$$

4. The data component of the original DCDS is augmented with a constraint that enforces the freshness of the results returned by $\mathbf{f}$ w.r.t. the i-th component of (the previous extension of) R:

$$\forall x_1, \ldots, x_n. R_{prev}(x_1, \ldots, x_n) \rightarrow \neg Temp_f(x_i)$$

Thanks to this (linear) transformation, we can introduce the surface syntax $\mathbf{f}_{R[id]}$ to indicate that $\mathbf{f}$ is fresh w.r.t. the i-th column of R, remembering that a DCDS employing such a syntactic sugar can always be transformed into a standard DCDS.

In this respect, the CR-GROUP action of the archery training DCDS in Section A.3 can be correctly rephrased as follows:

$$\begin{array}{l} \text{CR-GROUP}() : \\ \quad \left\{ \text{true} \rightsquigarrow \textbf{add}\ \left\{ \begin{array}{l} Group(\textbf{newId}_{Group[id]}()), \\ State(\textbf{newId}_{Group[id]}(), \text{in}) \end{array} \right\} \right\} \end{array}$$

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Modeling and verifying Active XML artifacts. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 32(3):10–15, 2009.

[3] Serge Abiteboul, Victor Vianu, Bradley S. Fordham, and Yelena Yesha. Relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, 61(2):236–269, 2000.

[4] Eric Badouel, Loïc Hélouët, and Christophe Morvan. Petri nets with semi-structured data. In *Proc. of 36th International Conference on Application and Theory of Petri Nets and Concurrency*, 2015.

[5] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. of PODS*, pages 163–174. ACM Press, 2013.

[6] Babak Bagheri Hariri, Diego Calvanese, Alin Deutsch, and Marco Montali. State-boundedness for decidability of verification in data-aware dynamic systems. In *Proc. of KR*, 2014.

[7] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Description logic Knowledge and Action Bases. *JAIR*, 46:651–686, 2013.

[8] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. An abstraction technique for the verification of artifact-centric systems. In *KR*, 2012.

[9] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: A database theory perspective. In *Proc. of PODS*, pages 1–12. ACM Press, 2013.

[10] Diego Calvanese, Marco Montali, Fabio Patrizi, and Michele Stawowy. Plan synthesis for knowledge and action bases. In *Proc. of IJCAI*, 2016. To appear.

[11] Diego Calvanese, Marco Montali, and Ario Santoso. Verification of generalized inconsistency-aware knowledge and action bases. In *Proc. of IJCAI*, 2015.

[12] Giuseppe De Giacomo, Yves Lesperance, and Fabio Patrizi. Bounded situation calculus action theories and decidable verification. In *Proc. of KR*, pages 467–477, 2012.

[13] Massimiliano de Leoni and Wil M. P. van der Aalst. Data-aware Process Mining: Discovering Decisions in Processes Using Alignments. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1454–1461, New York, NY, USA, 2013. ACM.

[14] Conrad Drescher and Michael Thielscher. A fluent calculus semantics for ADL with plan constraints. In *Proc. of the 11th Eur. Conf. on Logics in Artificial Intelligence (JELIA)*, volume 5293 of *LNCS*, pages 140–152. Springer, 2008.

[15] Richard Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *Proc. of OTM*, LNCS. Springer, 2008.

[16] Bartek Kiepuszewski, Arthur Harry Maria ter Hofstede, and Christoph J. Bussler. On structured workflow modelling. In *Seminal Contributions to Information Systems Engineering*. Springer, 2013.

[17] Ranko Lazić, Tom Newcomb, Joël Ouaknine, A. W. Roscoe, and James Worrell. Nets with Tokens Which Carry Data. In *Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, (ICATPN 2007)*, LNCS, pages 301–320. Springer, 2007.

[18] Andreas Meyer, Sergey Smirnov, and Mathias Weske. Data in business processes. Technical Report 50, Hasso-Plattner-Institut for IT Systems Engineering, Universität Potsdam, 2011.

[19] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

[20] Marco Montali and Diego Calvanese. Soundness of data-aware, case-centric processes. *International Journal on Software Tools for Technology Transfer*, pages 1–24, 2016.

[21] Marco Montali, Diego Calvanese, and Giuseppe De Giacomo. Verification of data-aware commitment-based multiagent systems. In *Proc. of AAMAS*, pages 157–164, 2014.

[22] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, July 2003.

[23] Fernando Rosa-Velardo and David de Frutos-Escrig. Decidability and complexity of petri nets with unordered data. *Theoretical Computer Science*, 412(34):4439 – 4451, 2011.

[24] Natalia Sidorova, Christian Stahl, and Nikola Trcka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Inf. Syst.*, 36(7):1026–1043, 2011.

[25] Natalia Sidorova, Christian Stahl, and Nikola Trčka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Information Systems*, 36(7):1026–1043, November 2011.

[26] W. M. P. Van Der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08:21–66, February 1998.

[27] Victor Vianu. Automatic verification of database-driven systems: a new frontier. In *Proc. of ICDT*, pages 1–13, 2009.