Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Soundness of Data-Aware, Case-Centric Processes

Marco Montali and Diego Calvanese

KRDB Research Centre for Knowledge and Data,
Free University of Bozen-Bolzano, Italy

Received: date / Revised version: date

Abstract. In recent years, a plethora of foundational results and corresponding techniques and tools has been developed to support the modeling, analysis, execution and improvement of business processes along their entire lifecycle. A major shortcoming of the analysis techniques is that they solely focus on the control-flow dimension of the process, omitting how business objects (i.e., cases) and their data affect and are manipulated by process instances and their tasks. In this work, we aim at filling this gap. We recast the classical notion of case-centric business process in a data-aware context. An emitter action is used to generate new cases, and while a case flows through the process control-flow, corresponding data are created, updated, and deleted by operating over a full-fledged relational database with constraints. To make our investigation concrete, we ground it on the recently introduced framework of Data-Centric Dynamic Systems (DCDSs). We reformulate the standard correctness criterion of soundness into this rich setting, and show that it is in general undecidable to check. We then provide a fine-grained analysis on the role of data in business processes. We substantiate this analysis by introducing a class of case-centric DCDSs that enjoys good modeling principles, and at the same time guarantees decidability of soundness. Decidability is obtained by finding a cutoff on the number of process instances that must be subject to the soundness test. We finally show that the introduced modeling guidelines are strict, in the sense that weakening even one single requirement they pose leads to undecidability.

Send offprint requests to:

1 Introduction

This work is about the role of data in conventional business processes [32,14], and is concerned with verifying the correctness of processes when data and their manipulation over time are fully taken into account. With "conventional" processes, we mean processes whose control-flow is captured by typical process modelling languages such as BPMN, UML activity diagrams, YAWL or EPCs. These are centred around the key notion of *case*, intended as a (concrete or abstract) business object that is manipulated and evolved by (an instance of) the process so as to achieve the company's strategic goals, and in turn produce value to one or more customers/consumers [32,14]. Example of cases are an order within an order-to-cash process, an item within a production chain, or a complaint within an issue-to-resolution process. We refer to this class of processes as *case-centric processes*.

In recent years, a plethora of foundational results and corresponding techniques and tools has been developed to support the modeling, analysis, execution and improvement of case-centric processes along their entire lifecycle. A major shortcoming of such analysis techniques is that they solely focus on the control-flow dimension of the process, omitting how business objects and their data affect and are manipulated by process instances and their tasks. In particular, the static, formal correctness analysis for the control-flow of case-centric processes, has a long tradition, grounded on *workflow nets* and on the property of *soundness* [29,30,16]. In this context, case-centricity is a fundamental requirement, since soundness is typically checked by relying on the conceptual assumption that multiple process executions are separately driven by cases, and that the corresponding business objects are evolved in isolation.

The importance of considering data alongside the process control-flow has been extensively argued in a growing number of works (see, e.g., [20,19,31,13]). In this enriched setting, many interesting decidability results on the static analysis of data-aware processes have been produced over the last 15 years, see [9] for a survey. Within this extensive literature, robust decidability results on the verification of data-aware dynamic systems against rich first-order temporal properties have been shown lately, under the assumption that the system is *state-bounded*.[1] Such results have been studied by considering different modeling variants of data-aware business processes, in particular *(i)* artifact systems [6], *(ii)* data-centric dynamic systems (DCDSs) [2], and *(iii)* description logic-based dynamic systems [10].

Our aim is to merge these two lines of research, and in particular to study the correctness of case-centric processes by jointly taking into account the process control-

[1] Intuitively, state-boundedness requires that the number of data values stored in each single state of the system is bounded. Unboundedly many values can still be encountered within and across the runs of the system.

flow and the manipulation of (case-)data stored in a full-fledged relational database equipeed with first-order constraints/dependencies. A major drawback of the approaches based on state-boundedness is that it is a semantic property undecidable to check, but for which only sufficient, syntactic conditions can be defined [2,3]. To mitigate this issue, modelling guidelines can be introduced in such a way that the system is guaranteed to be state-bounded by design [28]. When applied to case-centric processes, state-boundedness implicitly limits the maximum degree of case parallelism, that is, the number of cases that can simultaneously coexist. To the best of our knowledge, [11] is the only work so far to propose modeling and methodological guidelines to achieve decidability of verification in a setting where no limitation is put on such number.

We adopt this methodological approach here. Using DCDSs as a concrete framework for our investigation, we show that, unsurprisingly, checking data-aware soundness is in general undecidable. In parallel, we provide a fine-grained analysis on the role of data in case-centric business processes. We then substantiate this analysis by applying the methodological guidelines of [11], and introduce a class of case-centric DCDSs that are based on good modeling principles, and for which model checking of a first-order variant of μ-calculus (that can express soundness) is decidable. This class is characterized by simultaneously operating on:

- the database constraints and their mutual relationships, and on
- the shape of queries that can be formulated in the process component of the DCDS so as to retrieve and modify the data maintained in the DCDS data component.

Notably, we pose minimal assumptions on the process control-flow, arguing that any control-flow pattern can be encoded into a DCDS by introducing a fixed set of dedicated database values to support the runtime instantiation of the pattern. This class of DCDSs does not enjoy state-boundedness, but is such that the unboundedly many data present in a state of the system are organized into isolated chunks, each referring to a single case identifier. In this sense, our key decidability result is incomparable to those in [6,2,10]. In addition, we show that the different features of this class are all necessary, in the sense that weakening one of them again leads to undecidability. In particular, our results imply that unboundedly many cases may coexist simultaneously only if they evolve in complete isolation, i.e., the evolution of one case is not affected by the data attached to another case.

Our investigation does not only lift soundness to the data-aware setting, but also indirectly recasts key notions in the theory of business processes, such as that of freedom of choice and case isolation, taking into account the interaction between processes and data.

The modeling restrictions we impose towards decidability, as well as the abstraction techniques we borrow from [2], present some similarities with the different notions of boundedness studied for the formal verification of (infinite-state) distributed systems [22,18,1]. In particular, some of the modeling restrictions we impose in this work resemble the notions of simple path- and depth-boundedness respectively studied in [1] and [22]. Instead, [18] and [2] independently obtained decidability results in their respective settings, by exploiting similar ideas related to recycling of data values/process names. On the other hand, there are key differences both in the modeling frameworks and in the verification results shown in these works. As for modeling, [22,18,1] focus on process algebras/communicating register automata where the exchanged data are atomic data values representing the pure names of the involved processes. Contrariwise, in [2] and the present work, the focus is on the dynamic manipulation of rich relational structures, without taking into account inter-process communication. As for the verification results, [22,18,1] study specific, standard properties such as reachability, coverability and termination. Instead, [2] and the present work show decidability for sophisticated temporal properties expressed in a first-order variant of the μ-calculus. It remains to be investigated how such two lines of research can be interconnected: on the one hand by studying whether the communication graphs of [22,18,1] can be suitably encoded using the relational structures with constraints of [2], and on the other hand by investigating whether arbitrary n-ary relations can be encoded leveraging pure names and the graph structure of inter-process communication.

2 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 [2]). In this variant, actions are described following the action formalism of [24], which provides STRIPS-like abstractions on top of the original DCDS action formalism. This variant is expressively equivalent to the original one [24].

2.1 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; technically, constraints are closed FO formulae over $\mathcal{C}$ and the constants in $\mathcal{I}_0$.
- $\mathcal{I}_0$ 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 $\text{KEY}(R[A])$ (resp., $\text{PK}(R[A])$) 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 $\text{PK}(S[B])$ 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 $\text{CARD}(R[A], m..n)$, 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 $\text{CARD}(A[R], m..n)$, 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 $\text{KEY}(R[A])$ and $A[R] \longrightarrow B[S]$.

All these constraint types can be easily encoded as FO formulae. Some examples will be shown in Section 2.3.

(Open) FO formulae can be used to query an instance of the data component. In particular, let $Q(\mathbf{x})$ be *query*, i.e., a FO formula with free variables $\mathbf{x}$, and let $\mathcal{I}$ be a a database instance over $\mathcal{R}$ and Δ. The *answer* $ans(Q, \mathcal{I})$ *to* Q *over* $\mathcal{I}$ is the set of assignments θ from $\mathbf{x}$ to the active domain of $\mathcal{I}$, such that $\mathcal{I} \models Q\theta$, where $Q\theta$ is the boolean query obtained from Q by substituting each free variable $x_i \in \mathbf{x}$ with $\theta(\mathbf{x})$. Recall that the *active domain* of a database instance is the set of values explicitly appearing in its tuples. Given a query Q with no free variables (also called a *boolean query*), we say that Q *is true in* $\mathcal{I}$, or equivalently that $\mathcal{I}$ *satisfies* Q, if $\langle\rangle \in ans(Q, \mathcal{I})$. In this light, a database instance $\mathcal{I}$ satisfies $\mathcal{C}$ if it satisfies every boolean query in $\mathcal{C}$.

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

Each effect e_i has the form

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

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 d for the action parameters, and for every answer θ to the query $Q(\mathsf{d}, \mathbf{x})$.
- A is a set of facts over $\mathcal{R}$, which include as terms: free variables $\mathbf{x}$ of Q, action parameters $\mathbf{p}$ and/or Skolem terms $f(\mathbf{x}', \mathbf{p}')$ (with $\mathbf{x}' \subseteq \mathbf{x}$, and $\mathbf{p}' \subseteq \mathbf{p}$). We use $\text{SKOLEM}(A)$ 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 $\mathbf{x}$ of Q and action parameters $\mathbf{p}$. 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 $Q(\mathbf{x}) \mapsto \textsc{act}(\mathbf{x})$, 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
$$Q(\mathbf{p}, \mathbf{x}) \rightsquigarrow \textbf{add } A \textbf{ del } D$$
can be equivalently re-expressed as a set of effects
$$\begin{aligned} Q(\mathbf{p}, \mathbf{x}) &\rightsquigarrow \textbf{add } A_1 \textbf{ del } D_1 \\ &\cdots \\ Q(\mathbf{p}, \mathbf{x}) &\rightsquigarrow \textbf{add } A_n \textbf{ del } D_n \end{aligned}$$
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
$$\bigvee_{i \in \{1,\ldots,n\}} Q_i(\mathbf{x}) \mapsto \textsc{act}(\mathbf{x})$$
can be equivalently re-expressed as a set of rules
$$Q_1(\mathbf{x}) \mapsto \textsc{act}(\mathbf{x}) \qquad \cdots \qquad Q_n(\mathbf{x}) \mapsto \textsc{act}(\mathbf{x})$$

This equivalent rearrangment will be useful for the class of DCDSs introduced in Section 4, for which add and delete facts are required to obey to some restrictions.

2.2 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. This transition system 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 $\textsc{act}$ with parameter assignment ρ is selected and executed over $\mathcal{I}_0$. To do so, every effect e of $\textsc{act}$ (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 $\textsc{act}\rho$ in this way finally constitutes the next database instance, provided that no constraint in $\mathcal{C}$ is violated. In case of violation, the database instance is not changed, and $\textsc{act}\rho$ is considered to be inapplicable for the given configuration of service call results. 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.

The formal definition ot the execution semantics for DCDSs can be found in [2]. We recall it here, considering the case where services behave nondeterministically (i.e., calling the same service with the same inputs in two different states could result in different values).

Formally, let $\mathcal{I}$ be a database instance over $\mathcal{R}$ and Δ, such that $\mathcal{I}$ satisfies all constraints in $\mathcal{C}$. Consider an action $\textsc{act}(\mathbf{p}) : \{e_1, \ldots, e_m\}$ in $\mathcal{A}$, where e_i is of the form $Q_i(\mathbf{p}, \mathbf{x}) \rightsquigarrow \textbf{add } A_i \textbf{ del } D_i$, for $i \in \{1, \ldots, m\}$. A *parameter substitution* ρ for $\textsc{act}$ is an assignment that maps the parameters $\mathbf{p}$ of $\textsc{act}$ to corresponding values in Δ. We say that ρ is *legal for* $\textsc{act}$ *in* $\mathcal{I}$ if there exists a condition-action rule $Q(\mathbf{x}) \mapsto \textsc{act}(\mathbf{x})$ in ϱ such that $\rho \in \mathit{ans}(Q, \mathcal{I})$.

When $\textsc{act}\rho$ is applied on $\mathcal{I}$, it produces a set of facts $\textsc{do}(\mathcal{I}, \textsc{act}\rho)$, in accordance to the execution semantics of its effects. Technically, we have

$$\textsc{do}(\mathcal{I}, \textsc{act}\rho) = (\mathcal{I} \setminus F^-) \cup F^+, \text{ where}$$

$$\begin{aligned} F^- &= \textstyle\bigcup_{j \in \{1,\ldots,m\}} \bigcup_{\theta_j \in \mathit{ans}(Q_j\rho, \mathcal{I})} \bigcup_{F_k \in D_i} F_k\theta_j \\ F^+ &= \textstyle\bigcup_{j \in \{1,\ldots,m\}} \bigcup_{\theta_j \in \mathit{ans}(Q_j\rho, \mathcal{I})} \bigcup_{F_k \in A_i} F_k\theta_j \end{aligned}$$

This set of facts provides the footprint of the new database instance, which is completely determined only once the ground service calls contained in $\textsc{do}(\mathcal{I}, \textsc{act}\rho)$ are substituted by corresponding results.

We are now ready to formally define the construction of $\Upsilon_\mathcal{S} = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$:

- $s_0 = \mathcal{I}_0$;
- db is the identity function;
- Σ and $\Rightarrow$ are defined by simultaneous induction as the smallest sets satisfying the following properties:
  - $\mathcal{I}_0 \in \Sigma$

– for $\mathcal{I} \in \Sigma$, then for every action $\textsc{act}(\mathbf{p})$ in $\mathcal{A}$, every legal parameter substitution θ for $\textsc{act}$ in $\mathcal{I}$, and every substitution *sigma* mapping each ground service call contained in $\textsc{do}(\mathcal{I}, \textsc{act}\rho)$ to a corresponding value in Δ, if $\mathcal{I}' = \textsc{do}(\mathcal{I}, \textsc{act}\rho)\sigma$ satisfies all constraints in $\mathcal{C}$, then $\mathcal{I}' \in \Sigma$ and $\mathcal{I} \Rightarrow \mathcal{I}'$.

2.3 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 1. 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.
– $Trusts(subj, obj)$ 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 1 is equipped with the following constraints:

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

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

Similarly, for group states we have:

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

– The first columns of $MerryM$, $State$, and $Meets$ are the (primary) keys of the corresponding relations[2]:

$$\forall id, n_1, b_1, c_1, g_1, n_2, b_2, c_2, g_2.\ MerryM(id, n_1, b_1, c_1, g_1) \wedge MerryM(id, n_2, b_2, c_2, g_2) \rightarrow n_1 = n_2 \wedge b_1 = b_2 \wedge c_1 = c_2 \wedge g_1 = g_2$$

$$\forall id, s_1, s_2. State(id, s_1) \wedge State(id, s_2) \rightarrow s_1 = s_2$$

$$\forall s, g_1, w_1, g_2, w_2. Meets(s, g_1, w_1) \wedge Meets(s, g_2, w_2) \rightarrow g_1 = g_2 \wedge w_1 = w_2$$

– 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:

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

– $Meets$ 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:

$$\forall s, g, w. Meets(s, g, w) \rightarrow g = \texttt{null} \vee Group(g)$$
$$\forall g, s_1, w_1, s_2, w_2, s_3, w_3.\ Meets(s_1, g, w_1) \wedge Meets(s_2, g, w_2) \wedge Meets(s_3, g, w_3) \rightarrow s_1 = s_2 \vee s_1 = s_3 \vee s_2 = s_3$$

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 2 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 $\textsc{cr-group}$ action is always executable, and has the effect of creating a new group in the information system, putting it into the `in` state. The

[2] 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 $Trusts$.

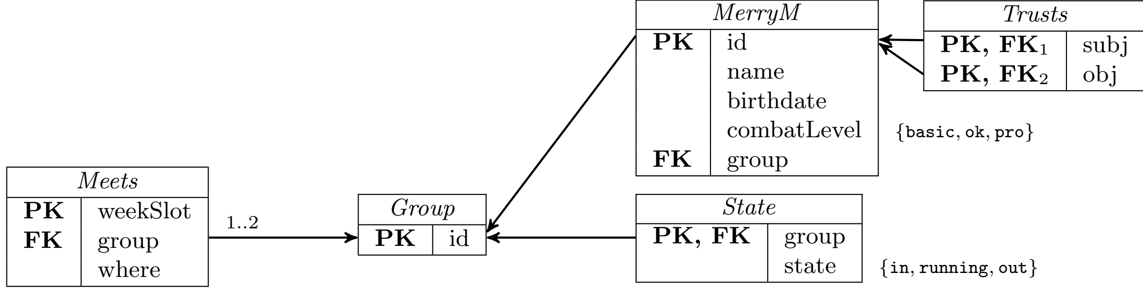


Fig. 1: The archery training data component

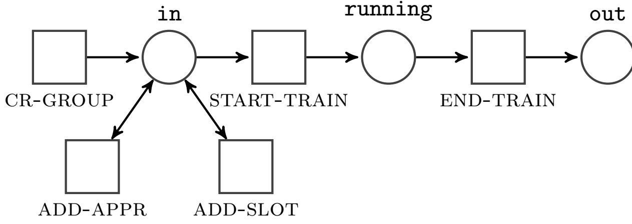


Fig. 2: The archery training process control-flow. Tokens denote process cases, which in this example correspond to groups of people

group identifier is injected into the system by calling the **newId** service.

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

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

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.

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

$$\begin{array}{l} \textsc{add-appr}(\mathsf{m}, \mathsf{g}) : \\ \left\{ \begin{array}{ll} \mathit{MerryM}(\mathsf{m}, n, b, l, g_o) \rightsquigarrow \mathbf{del} & \{\mathit{MerryM}(\mathsf{m}, n, b, l, g_o)\} \\ & \mathbf{add}\ \{\mathit{MerryM}(\mathsf{m}, n, b, l, \mathsf{g})\} \end{array} \right\} \end{array}$$

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.

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

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

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 2, 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.

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

$$\begin{array}{l} \textsc{start-train}(\mathsf{g}) : \\ \{\mathit{State}(\mathsf{g}, s) \rightsquigarrow \mathbf{del}\{\mathit{State}(\mathsf{g}, s)\}\ \mathbf{add}\{\mathit{State}(\mathsf{g}, \mathtt{running})\}\} \end{array}$$

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.

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

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

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.

2.4 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. In the archery training DCDS of Section 2.3, 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.

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 the next instance of R_{prev} with the current extension of R. This is done by emptying the content of R_{prev}, and by filling it with all tuples currently stored in R:
$$\begin{array}{ll} R_{prev}(\mathbf{x}) \rightsquigarrow \mathbf{del} & \{R_{prev}(\mathbf{x})\} \\ R(\mathbf{x}) \rightsquigarrow \mathbf{add} & \{R_{prev}(\mathbf{x})\} \end{array}$$
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 $Temp_f$:
$$\mathsf{true} \rightsquigarrow \mathbf{add}\ \{Temp_f(\mathbf{f}())\}$$
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:
$$\forall x_1, \dots, x_n . R_{prev}(x_1, \dots, x_{i-1}, x_i, \dots, x_n) \rightarrow \neg Temp_f(x_i)$$

The constraint indeed checks that the i-th component of each R_{prev} tuple does not correspond to the result returned by $\mathbf{f}$, which is stored into the $Temp_f$ relation.

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 2.3 can be correctly rephrased as follows:

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

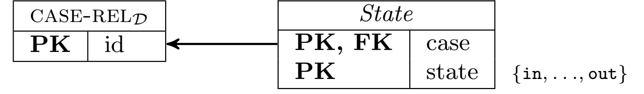


Fig. 3: Core relations and constraints in a case-centric data component $\mathcal{D}$

3 Data-Aware, Case-Centric Processes

In this section, we introduce some minimal modeling guidelines on the shape that a DCDS must have in order to be considered "case-centric". We then provide a fine-grained classification of the different types of data stored in a relational database taking the (business) process perspective, and considering that our main focus is in the static analysis of the system, i.e., before cases are actually executed. We conclude by introducing a formal notion of process correctness, reformulating the well-known notion of *soundness* [29] in our data-aware setting. This constitutes the basis for Section 4, in which we study the boundaries of (un)decidability when checking soundness over (subclasses of) DCDSs.

3.1 Basic Control Structures

To identify the class of *case-centric DCDSs*, we make the following basic assumptions on their shape.

Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS. For $\mathcal{S}$ to be case-centric, we first assume that $\mathcal{D}$ contains two relations (and related constraints) to keep track of case-related information, as shown in Figure 3:

- A unary *case relation*, storing case identifiers. The name of such a relation depends on the specific domain under study. For example, in Section 2.3, the case relation is *Group*. We use CASE-REL$_{\mathcal{D}}$ to denote the case relation of data component $\mathcal{D}$.
- A binary *State* relation, storing the current state(s) of each case.

Similarly to the special input and output places used in workflow nets [29], we assume that each case has two special states: an `in` state in which the case is located when it is created, and an `out` state reached by the case when the execution of the process on it terminates. This data structure generalizes that of *Group* and *State* in Section 2.3. The main difference is that in this general form, the primary key of relation *State* is constituted by the entire relation, so as to support the possibility of associating multiple states to the same case. This is particularly useful to model concurrency and sub-processes, that is, multiple threads for the same case, each located in a different state.

Let us now consider two general requirements on the process component $\mathcal{P}$, in relation to the manipulation of cases independently from the specific domain. First, we assume that $\mathcal{P}$ contains a special action (without parameters) to create a new case. This is the only action that can actually add values to the CASE-REL$_\mathcal{D}$ relation, which we consider to be always initially empty. For compactness of notation, from now on we assume that CASE-REL$_\mathcal{D}$ is relation C with attribute *id*. The action generating a new case is then formalized as:

$$\textsc{new-case}():\left\{\mathsf{true} \rightsquigarrow \mathbf{add}\left\{\begin{array}{l} C(\mathbf{newId}_{C[id]}()),\\ State(\mathbf{newId}_{C[id]}(),\mathtt{in})\end{array}\right\}\right\}$$

The NEW-CASE action can be considered as a data-aware variant of *emitter transitions* in Petri nets. Its executability depends on the targeted domain, i.e., on the domain-specific conditions that allow for the generation of a new case. In the most general situation, e.g., the one in which the creation of a new case depends on external stakeholders (i.e., cannot be controlled by the company that manages the process execution), the condition-action rule determining the executability of NEW-CASE() is simply: $\mathsf{true} \mapsto \textsc{new-case}()$.

The second general requirement concerns the core nature of case-centric processes: each action ACT of $\mathcal{P}$ (except NEW-CASE) is required to have one parameter matching with a case identifier. For this reason, each condition-action rule is required to have the following form, where Φ is a domain-specific condition for ACT:

$$C(c) \wedge \Phi(c,\mathbf{x}) \mapsto \textsc{act}(c,\mathbf{x})$$

In summary, a DCDS $\langle\mathcal{D},\mathcal{P}\rangle$ is *case-centric* if:
- $\mathcal{D}$ contains the relations and constraints shown in Figure 3, and is such that CASE-REL$_\mathcal{D}$ is empty in the initial database of $\mathcal{D}$;
- $\mathcal{P} = \langle\mathcal{F},\mathcal{A},\varrho\rangle$ is such that:
  - $\mathcal{A} = \mathcal{A}_{norm} \uplus \{\textsc{new-case}()\}$, where NEW-CASE() is defined as above;
  - $\varrho = \varrho_{norm} \uplus \{\mathsf{true} \mapsto \textsc{new-case}()\}$, where each condition-action rule in ϱ_{norm} has the form $C(c) \wedge \Phi(c,\mathbf{x}) \mapsto \textsc{act}(c,\mathbf{x})$, in which C corresponds to CASE-REL$_\mathcal{D}$, and ACT $\in \mathcal{A}_{norm}$.

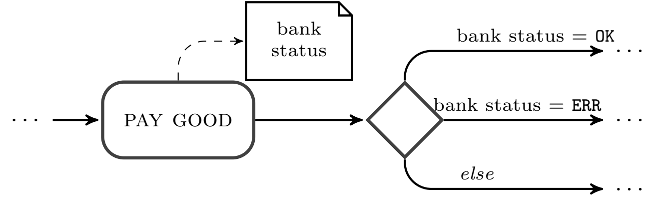


Fig. 4: Control values in a simple BPMN diagram: `OK` and `ERR` are control values that route the process to different alternative flows

3.2 The Different Roles of Data

We now overview how the data maintained in the data component of a data-aware, case-centric process can be classified w.r.t. the process itself. Our focus is on static analysis, i.e., on the study of how the process will manipulate the underlying data before process cases are actually executed in the real-world. We use the archery training process of Section 2.3 to illustrate the different concepts.

Control, distinguished, and pure values. Among all the data values Δ maintained in a database, some values have the peculiar characteristic of being explicitly used by business processes to select which actions can be executed and which not, in turn determining the allowed courses of execution. For this reason, we call such values *control values*. In the archery training example, the control values are:
- All the values maintained in the second column of the *State* relation, to keep track of the state of each group and, in turn, determine which actions can be executed.
- The `null` value, because it determines whether a certain merry man can be selected as apprentice for a group or not.

It is important to notice that these values are typically pre-defined, and *finitely many*. They can be identified by analyzing the process control-flow, and extracting all those values that are explicitly mentioned in its specification. In DCDSs, this is done by scanning all queries used in the conditions of condition-action rules, as well as those specifying the body of action effects. Figure 4 shows a simple example of control values in BPMN.

Obviously, the database does not just contain control values. A second important class of data values is that of *distinguished values*, that is, values that have a special semantics for the domain experts and end users. In particular, those values can be explicitly mentioned by users when posing queries over the database. In this light, control values can be considered as a subset of distinguished values, since they can be mentioned by users to query the state of cases. In the archery training process, distinguished values are all group states, the special value `null` and all combat levels. It is in fact expected that

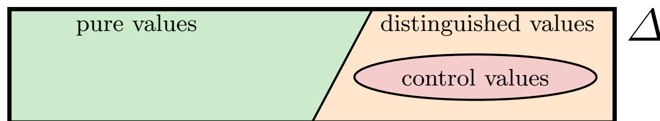


Fig. 5: Control, distinguished and pure values

Robin Hood will be interested, e.g., in knowing which are the merry men that have a `basic` combat level.

All other values of Δ are not explicitly used to route the process nor to formulate user queries, but are simply data values describing a relevant information about the domain. Examples of such kind of values are tuple identifiers (i.e., internal ids used to fill the primary key of a relation), characteristics of products in a catalogue (such as products names and their bar codes), and features of people (such as social security numbers, credit card numbers, names, and addresses). In the archery training example, these are the values appearing in all columns of the *MerryM* relation but the combat level (and the special `null` value). The crucial characteristic of a data value of such kind is that the value does not have an importance per sè, but only in relation with other data. As a consequence, they are only compared for identity with each other. For example, it is important to know that a given person address is associated to the *same* social security number to which a cloned credit card is registered, but the consistent *renaming* of the social security number to a different number would not lead to any information loss, provided that the relationship with the address and credit card are maintained: comparing such values for identity would still lead to the same result. This is especially true when focusing on static analysis, because at design time many values are not yet present in the database, but will be actually injected during the execution of process cases.

This notion of "invariance under renaming" is very well known in computer science. On the one hand, it relates to the classical notion of *genericity* in databases [12], and to invariance under isomorphism in first-order logic. On the other hand, it is reminiscent of name binding in programming languages [17], of name usage in distributed systems [26], process algebras [25], and Petri nets [27], and of nominal sets[3] [7].

Leveraging on the terminology used in many of these papers, we refer to this kind of data values as *pure values*. Notably, both genericity and nominal sets have been exploited to provide key decidability results for the static verification of dynamic systems manipulating data, see respectively [6,2,4,10] and [8].

Figure 5 overviews how the overall set of data values Δ is decomposed into the three sets of control, distinguished, and pure values.

[3] Considering in particular those with equality simmetry.

Read/write data access. When analyzing how a process impacts on data, it is important to understand in a fine-grained way *how* such data are accessed by the process. This has a twofold impact on verification: on the one hand, it may help towards decidability, and on the other hand it may help in reducing the state space to be considered. Consider again the archery training example. Clearly, the *State* and *MerryM* relations play a radically different role in the process: the first one is a core relation in controlling the process execution, whereas the second one keeps domain data that only partially interact with the process, which never uses the columns maintaining names and birthdates.

More in general, a first useful characterization concerns the kind of access that processes can have on certain relations and relation attributes: *(i)* no-access, *(ii)* read-only, *(iii)* write-only, and *(iv)* read-write.

Interestingly, read-write attributes are typically used to store control data values used to drive the evolution of the different process cases. Also notice that the dividing line between "no access" and "read-only" is not crisp, since some attributes are read just to present useful information to the end users, but they do not really participate to the process. This is, e.g., the case of the merry man name and birthday attributes, which, in the context of the archery training process, do not impact the process execution at all, but could still be employed to make Robin Hood able to make sense out of the corresponding identifiers.

Among all relations, of particular interest in this paper are the two classes of cardinality-immutable and read-only relations.

Cardinality-immutable relations are relations that can be accessed in write-mode by the process only in a controlled way. In particular, the attributes forming the primary key of a cardinality-immutable relation are read-only. In this special case, the process can manipulate the content of the relation, but cannot shrink nor extend its set of tuples. To comply with cardinality-immutable relations, the action specification of a DCDS must satisfy the following condition: for each relation R with $\mathrm{PK}(R[K])$, every effect $Q \rightsquigarrow \mathbf{add}\ A\ \mathbf{del}\ D$ is such that:

- either the effect does not mention R in A or in D; or
- Q mentions R in a way that variables $\mathbf{id}$ are used in the positions corresponding to attributes in K, and both A and D mentions R as well, in a way that the same variables $\mathbf{id}$ are again used in the positions corresponding to attributes in K.

Intuitively, the first kind of effect does not touch the extension of R, therefore maintaining it implicitly unaltered. The second kind of effect models instead the case of an *update* for all R-tuples, where the update of each tuple is modelled as a deletion and addition in such a way that the primary key is left untouched.

Read-only relations are cardinality-immutable relations whose attributes are all read-only. To comply with

read-only relations, the action specification of a DCDS must never mention them in the right-hand side of effects.

Throughout the paper, we assume that once a relation is declared to be cardinality-immutable or read-only, then every DCDS complies with this requirement.

Example 1. The archery training DCDS is such that relation *MerryM* is cardinality-immutable. It is easy to see that whenever such a relation is mentioned in the left-hand side of an effect, then it is also mentioned both in the corresponding **add** and **del** sets, in such a way that the primary key is maintained. ■

Relevant vs irrelevant data, and vertical partitioning. The classification of attribute access helps in sharpening the focus on those portions of the database that are really *relevant* for the static analysis of business processes. Specifically: no-access attributes, write-only attributes, and read-only attributes that are just accessed for presentation purposes, can be abstracted away without impacting on the static analysis, given the fact that they do not influence the execution of process cases. Since this operation corresponds to removing columns (or tables) from the database schema, we call it *vertical partitioning*. Figure 6 shows the result of vertical partitioning on the archery database schema.

Related vs unrelated data, and horizontal partitioning. The last dimension concerning the way data can be manipulated by business processes is about query patterns, which determine how data are *related* to each other. If no restriction is made on the way queries are posed over the database, then every datum can in principle be related with any other datum (think about the extreme case of a query that executes the cartesian product of every relation present in the database schema).

This chaotic way of accessing the database is not only unreasonable, but also dashes any hope of getting decidability of static analysis (see, e.g., [2,21,11]). Contrariwise, relations are usually queried in a well-disciplined way, e.g., by applying joins in a compatible way with foreign keys. When business processes follow this principle, their execution tends to produce data organized into *data slices*, each containing data that are related to each other, and such that data of different slices are only partially related, or even completely unrelated. We call this emergent property *horizontal partitioning*. Figure 7 provides an intuition for horizontal partitioning in our running example.

The notion of horizontal partitioning into data slices perfectly fits case-centric business processes, which typically assume that each case evolves independently from the others. This is apparent when considering the control-flow constructs of standard process modelling languages, which do not provide any form of support for inter-case relationships, and translates into the formal notion of *freedom of choice* [29] by considering their underlying Petri net-based semantics. In fact, free-choice nets guarantee that the route a token can take among multiple choices does not depend on the position of other tokens, enforcing this idea of isolation.

Understanding whether a case-centric business process enforces this separation becomes much more problematic in a data-aware setting: even though two cases may seem independent from the control-flow perspective, they could still indirectly interact through the underlying common database. Studying this aspect is exactly the purpose of Section 4 below. Finally, notice that this notion of separation becomes much less definite and even more difficult to be properly understood when adopting an artifact-centric approach to process modelling. In fact, one of the trickiest aspects of business artifacts is the fact that they could establish many-to-many relations with each other [15], mutually affecting their evolution.

3.3 Data-Aware Soundness

Soundness is a fundamental notion in the control-flow analysis of standard, case-centric process models. Since the original definition in [29], many variants and relaxations have been proposed to characterize the correctness of processes [30]. We lift now the classical notion of soundness to account for case-centric, data-aware processes—relaxations can be then easily defined by reconstructing the literature in such a data-aware setting.

Soundness is defined over processes that have two special states: an *input* state denoting the starting point of each case, and an *output* state denoting the ending point. With this minimal assumption, soundness intuitively imposes that, for each case that starts in the input state, we have:

1. *eventual termination*: the output state is always reachable, i.e., in every state there is a sequence of executable tasks that leads to the output state;
2. *proper termination*: the output state is always reached in a clean way, i.e., when the case is in the output state, then there is no executable task;
3. *task executability*: there is no dead task, i.e., for each task there is at least one execution leading to a state in which that task is executable.

We now rephrase soundness in the context of case-centric DCDSs. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a case-centric DCDS with $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$. Intuitively, *data-aware soundness* imposes that, for each case identifier c belonging to the CASE-REL$_{\mathcal{D}}$ relation and such that $State(c, \mathtt{in})$ holds[4], we have:

1. *eventual termination*: along every possible future execution, it is always the case that a state can be reached where $State(c, \mathtt{out})$ holds;

[4] Recall that each case initially starts from the `in` state, in accordance with the specification of NEW-CASE.

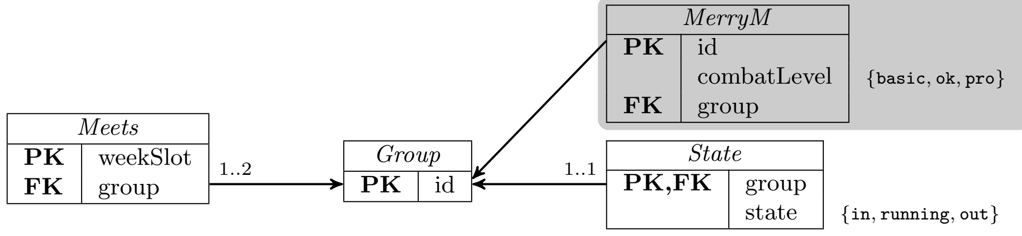


Fig. 6: Vertical partitioning of the archery database schema on read-write attributes and primary keys; notice that *MerryM* is a cardinality-immutable relation

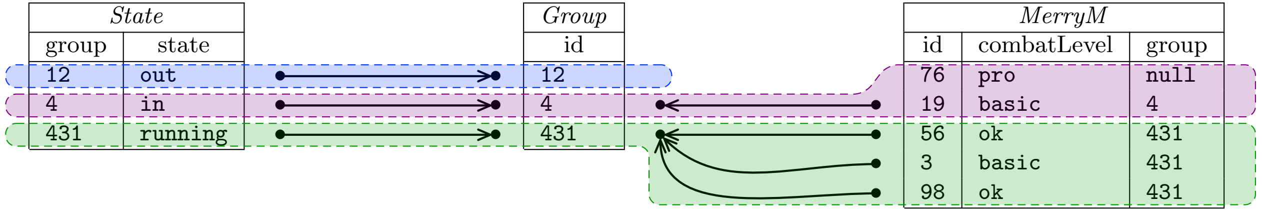


Fig. 7: Horizontal partitioning with three slices considering a possible database instance for the archery training process. The merry man with id `76` is part of the slice of group `4` because, even though it is not one of the members of group `4`, he can be queried and become one of its apprentices; this is in fact the only tuple that is shared by slices belonging to different groups in the `in` state

2. *proper termination*: when $State(c, \texttt{out})$ holds, then there is no condition-action rule in ϱ_{norm} that can fire on c;
3. *task executability*: for each action $\textsc{act}(\mathbf{p})$ in $\mathcal{A}_{norm}$, there exists an execution leading to a state in which $\textsc{act}(\mathtt{d})$ can be executed with some concrete values $\mathtt{d}$ instantiating the parameters $\mathbf{p}$, according to the condition-action rules in ϱ_{norm}.

We now show how data-aware soundness can be actually formalized in the $\mu\mathcal{L}_P$ verification logic introduced in [2]. $\mu\mathcal{L}_P$ is a variant of first-order μ-calculus that allows only for a controlled, "persistent" form of first-order quantification across different states of a DCDS. In particular, quantification only applies as long as data values persist in the active domain of the evolving DCDS database. Given a case-centric DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, we have that $\mathcal{S}$ is *sound* if $\Upsilon_{\mathcal{S}} \models \Phi_{sound}$, where Φ_{sound} corresponds to formula

$$\nu Z.\big(\forall c. C(c) \wedge State(c, \texttt{in}) \rightarrow (\xi(c) \wedge \pi(c) \wedge \tau(c))\big) \wedge Z$$

in which, under the assumption that $C = \textsc{case-rel}_{\mathcal{D}}$:

1. $\xi(c)$ models the eventual termination of c as follows:

$$\nu W.\Big(\mu Y.\big(C(c) \wedge State(c, \texttt{out}) \vee (C(c) \wedge Y)\big)\Big) \wedge (C(c) \wedge W)$$

2. $\pi(c)$ models the proper termination of c as follows:

$$\nu W.\Big(State(c, \texttt{out}) \rightarrow \bigwedge_{C(c') \wedge \Phi(c',\mathbf{p}) \mapsto \textsc{act}(c',\mathbf{p}) \text{ in } \mathcal{A}_{norm}} \neg \exists \mathbf{x}.\Phi(c, \mathbf{x})\Big) \wedge (C(c) \wedge W)$$

3. $\tau(c)$ models task executability on c as follows:

$$\bigwedge_{\textsc{act}(\mathbf{p}) \text{ in } \mathcal{A}_{norm}} \mu Y.\Big(\bigvee_{C(c') \wedge \Phi(c',\mathbf{p}) \mapsto \textsc{act}(c',\mathbf{p}) \text{ in } \mathcal{A}_{norm}} \exists \mathbf{x}.\Phi(c, \mathbf{x})\Big) \vee (C(c) \wedge Y)$$

It is important to observe that, while being expressible in the $\mu\mathcal{L}_P$ logic, checking soundness cannot be directly tackled by the technique presented in [2]. In fact, the key property required by [2] on the shape of DCDSs towards decidability of verification against $\mu\mathcal{L}_P$ properties, is that of *state-boundedness*. State-boundedness requires the existence of a number that provides an overall bound over the size of each database produced along any run of the DCDS under study. This does not hold for case-centric DCDSs, for which there is no bound on the number of cases that may simultaneously coexist in the system.

4 Soundness of Case-Centric DCDSs

We now study verification of (data-aware) soundness over case-centric DCDSs, showing that its decidability holds for an interesting class of case-centric DCDSs. More specifically, our main goal is to introduce a set of modeling guidelines that on the one hand guarantee decidability of soundness, and on the other hand allow the modeler to capture real-life processes. We achieve the formulation of this class incrementally, by showing that each limitation we introduce is necessary towards decidability (i.e., by relaxing it, soundness becomes undecidable). To do so, we take inspiration from the artifact-centric methodology studied in [11]. However, our results are quite different: those in [11] are focused on ver-

ification of termination properties in the context of an UML-based artifact-centric framework, whereas here we are interested in the study of soundness over case-centric process models.

The aspects that we incrementally tackle towards decidability are:

- *Navigationality*: the process cannot arbitrarily work over all relations of the schema, but must access them in a controlled way. By leveraging the notion of case, we require queries to be *case-navigational*, i.e., they start from the case-relation and navigate foreign keys backward so as to reach other relations.
- *Case-width-boundedness*: when navigating a foreign key from a target to a source relation, the process must not have the possibility of creating unboundedly many tuples of the source relation.
- *Case-depth-boundedness*: it must be prevented that the process can create and manipulate unbounded chains of relations.
- *Isolation*: each case manipulates its own data, and does not interfere with the data of the other cases. When different cases are not properly isolated, that is, they share read-write relations, then they could interact in such a way that, in spite of case-depth-boundedness, they could form unbounded chains altogether.

All undecidability proofs that we provide in the following rely on reductions of soundness checking for case-centric DCDSs to the halting problem of two-counter machines [23], which are machines simultaneously operating over a propositional control state and the values maintained by two counters through increment and decrement, in a way that depends on the values of the counters. We just focus on the core of the reductions, showing how each considered class of case-centric DCDSs can encode a counter and corresponding operations of increment, decrement, and test for zero. More specifically, we show how the DCDS under study can simulate the following three operations:

- $\langle \mathtt{k}, \mathtt{c+}, \mathtt{k}' \rangle$, which indicates that when the machine is in control state $\mathtt{k}$, then it moves to state $\mathtt{k}'$ by incrementing the value of counter $\mathtt{c}$ by one unit.
- $\langle \mathtt{k}, \mathtt{c-}, \mathtt{k}' \rangle$, which indicates that when the machine is in control state $\mathtt{k}$ and counter $\mathtt{c}$ holds a positive value, then the machine moves to state $\mathtt{k}'$ by decrementing the value of counter $\mathtt{c}$ by one unit.
- $\langle \mathtt{k}, \mathtt{c==0}, \mathtt{k}' \rangle$, which indicates that when the machine is in control state $\mathtt{k}$ and counter $\mathtt{c}$ is zero, then the machine moves to state $\mathtt{k}'$.

Once these operations are available, a program operating over two counters can be encoded by a DCDS where each case represents an execution of the program, and control states correspond to case states, such that the first instruction corresponds to state `in`, and the last one to state `out`. Furthermore, once a single counter is encoded into a DCDS, a second counter can be immediately obtained by duplicating the involved relations, or by adding a new column that plays the role of counter identifier. This technique has been recently independently adopted in [21] and [11].

4.1 Undecidability of Soundness

Unsurprisingly, given the expressiveness of case-centric DCDSs, the following negative result holds:

Theorem 1. *Checking soundness of case-centric DCDSs is undecidable.*

Proof. Consider a case-centric DCDS equipped with two unary relations $C_1(v)$ and $C_2(v)$. In the general case, no restriction is imposed on how the DCDS can manipulate the extension of these two relations, hence each case can arbitrarily query and manipulate those relations. In particular, the DCDS can simulate a 2-counter machine where the values of the counters correspond to the number of tuples in C_1 and C_2, and, consequently:

- increment is encoded by adding a fresh value into the corresponding counter relation;
- decrement is encoded by (nondeterministically) removing one of the tuples in the corresponding counter relation;
- test for zero is simulated by asking whether the extension of the corresponding counter relation is empty.

This technique resembles the one used in [11] and [21]. More specifically, the DCDS maintains a unary relation *Contr*, which contains a single case identifier responsible for the manipulation of the two counters. An initial action selContr is devoted to select the controller as follows:

- if *Contr* is empty, then the case on which the action is applied becomes the controller of the counter, and is moved to the first control state $\mathtt{k}_0$;
- otherwise, the case immediately terminates (i.e., it is put in the `out` state).

By assuming that *Case* is the case relation, this is formalized as:

$$Case(c) \wedge State(c, \mathtt{in}) \mapsto \textsc{selContr}(c)$$

$$\textsc{selContr}(\mathtt{c}) : \left\{ \begin{array}{r} \neg\exists x.Contr(x) \rightsquigarrow \mathbf{del}\ \{State(\mathtt{c}, \mathtt{in})\} \\ \mathbf{add} \left\{ \begin{array}{l} State(\mathtt{c}, \mathtt{k}_0), \\ Contr(\mathtt{c}) \end{array} \right\} \\ \exists x.Contr(x) \rightsquigarrow \mathbf{del}\ \{State(\mathtt{c}, \mathtt{in})\} \\ \mathbf{add}\{State(\mathtt{c}, \mathtt{out})\} \end{array} \right\}$$

An increment operation of the form $\langle \mathtt{k}, \mathtt{c}_1\mathtt{+}, \mathtt{k}' \rangle$ is encoded as follows (notice the fact that only the controller case can apply the action):

$$Case(c) \wedge Contr(c) \wedge State(c, \mathtt{k}) \mapsto \textsc{inc}_1(c, \mathtt{k}')$$

$$\textsc{inc}_1(\mathtt{c}, \mathtt{n}) : \left\{ \begin{array}{r} State(\mathtt{c}, s) \rightsquigarrow \mathbf{del}\ \{State(\mathtt{c}, s)\} \\ \mathbf{add} \left\{ \begin{array}{l} State(\mathtt{c}, \mathtt{n}), \\ C_1(\mathbf{newVal}_{C_1[v]}()) \end{array} \right\} \end{array} \right\}$$

In the formalization above, the new control state is passed as an action parameter, and increment is simulated by introducing a fresh value into relation C_1, relying on the nullary service call **newVal**.

A decrement operation of the form $\langle \mathtt{k}, \mathtt{c_1}\text{-}, \mathtt{k}' \rangle$ is encoded as follows:

$$Case(c) \land Contr(c) \land State(c, \mathtt{k}) \land C_1(x) \mapsto \textsc{dec}_1(c, \mathtt{k}', x)$$

$$\textsc{dec}_1(\mathsf{c}, \mathsf{n}, \mathsf{e}) : \left\{ \begin{array}{l} State(\mathsf{c}, s) \rightsquigarrow \textbf{del}\ \{State(\mathsf{c}, s), C_1(\mathsf{e})\} \\ \qquad\qquad\qquad \textbf{add}\{State(\mathsf{c}, \mathsf{n})\} \end{array} \right\}$$

The structure is similar to increment, the key difference being that decrement is simulated by nondeterministically picking an element from C_1, and removing it from the extension of such relation. The element is picked in the condition of the condition-action rule, and passed as a parameter. This implicitly means that the decrement action can be executed only if C_1 contains at least one element.

Finally, an instruction of the form $\langle \mathtt{k}, \mathtt{c_1}\texttt{==0}, \mathtt{k}' \rangle$ is modelled as follows:

$$\begin{array}{l} Case(c) \land Contr(c) \\ \quad \land State(c, \mathtt{k}) \land \neg \exists x. C_1(x) \mapsto \textsc{moveTo}(c, \mathtt{k}') \end{array}$$

$$\textsc{moveTo}(\mathsf{c}, \mathsf{n}) : \left\{ \begin{array}{l} State(\mathsf{c}, s) \rightsquigarrow \textbf{del}\ \{State(\mathsf{c}, s)\} \\ \qquad\qquad\qquad \textbf{add}\{State(\mathsf{c}, \mathsf{n})\} \end{array} \right\}$$

We close the proof by remarking that a DCDS structured in this way is indeed case-centric. In fact, case centricity does by no means limit how the DCDS can query the relations of its schema, including C_1. $\square$

4.2 Navigational Case-Centric DCDSs

In Theorem 1, undecidability arises from the fact that the DCDS can manipulate two unary relations without any restriction. We consequently need to discipline how the extension of relations can be manipulated. As a first fundamental requirement, we impose that the DCDS can access facts of a given relation only by navigating from a case identifier. To do so, we impose some modelling guidelines on both the data and process components.

As for the data component, we require that its constraints are either *keys* or *foreign keys*. In this case, the data component is said to be *navigation-supporting*. Following the discussion of Section 3.2, we also explicitly distinguish relations whose extension can be freely manipulated by the process from those that are cardinality-immutable.

As for the process component, we define a class of case-centric DCDSs called *case-navigational*. Intuitively, a case-centric DCDS is case-navigational if the following three conditions hold:

- Every query is a navigational query rooted in a case identifier, that is, it "starts" from a case identifier, and then moves through schema relations by applying joins that navigate foreign key constraints backwards.
- Facts mentioned in the **add** and **del** sets of effects are also subject to navigational patterns, in such a way that tuples can be added, deleted, and modified only if they can be reached from a case identifier by means of a navigational query.

To formally define navigational queries, some preliminary notions are needed. A *weakly-guarded* FO query $Q(\mathbf{x})$ is inductively defined as follows:

(base case) $Q(\mathbf{x})$ is a conjunction of the form $\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \land \Phi_{neg}(\mathbf{x},\mathbf{y})$, where Φ_{neg} is a query whose atoms are either relational atoms, or (in)equalities, composed through boolean operators. Intuitively, Φ_{neg} is used as a filter.

(inductive case) $Q(\mathbf{x})$ is a conjunction of the form $\exists y.R(\mathbf{x}_0,\mathbf{y}) \land \bigwedge_{i\in\{1,\ldots,n\}} \Phi_i(\mathbf{x}_i,\mathbf{v}_i)$ where, for each $i \in \{1,\ldots,n\}$, the following three conditions hold: *(i)* $\mathbf{x}_0 \cup \mathbf{x}_i = \mathbf{x}$, *(ii)* $\mathbf{v}_i \subseteq \mathbf{x}_0 \cup \mathbf{y}$, and *(iii)* Φ_i is a weakly-guarded FO query.

We also say that a weakly-guarded query $\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \land \Psi$ *is rooted* in R. The appellative "weak" is used since, in the inductive case, not all free variables $\mathbf{x}$ of the original query Q are forced to appear in the (weak) guard R.

Given a data component $\mathcal{D}$ and a FO query Φ, we say that Φ is a *case-navigational query over* $\mathcal{D}$ if it obeys to the following conditions:

- Φ is a weakly-guarded query rooted in relation $\textsc{case-rel}_{\mathcal{D}}$.
- For each (weakly-guarded) subformula $\exists \mathbf{y}.S(\mathbf{x}_0,\mathbf{y}) \land \bigwedge_{i\in\{1,\ldots,n\}} \Psi_i(\mathbf{x}_i,\mathbf{v}_i)$, such that for each $i \in \{1,\ldots,n\}$, Ψ_i is rooted in relation R_i, $\mathbf{x}_0 \cup \mathbf{x}_i = \mathbf{x}$, and $\mathbf{v}_i \subseteq \mathbf{x}_0 \cup \mathbf{y}$, we have that:
  - either R_i is a cardinality-immutable relation, or
  - the following three conditions apply:
    1. Variables $\mathbf{v}_i$ appear exactly in those positions B of S that form primary key of S, i.e., $\mathrm{PK}(S[B])$ belongs to the constraints of $\mathcal{D}$.
    2. Variables $\mathbf{v}_i$ all appear in relation R_i.
    3. By denoting with A_i the set of attributes corresponding to the positions of variables $\mathbf{v}_i$, we have that $R_i[A_i] \longrightarrow S[B]$ belongs to the constraints of $\mathcal{D}$.
- The same check is applied to the innermost formula $\exists \mathbf{y}.R(\mathbf{x},\mathbf{y}) \land \Phi_{neg}(\mathbf{x},\mathbf{y})$, by considering each relational atom appearing in Φ_{neg}.

It is easy to see that checking whether a query is case-navigational for a given data component requires time that is linear in the size of the query. The definition is quite involved, but the underlying philosophy is rather intuitive: a case-navigational query compatible with $\mathcal{D}$ starts from a case identifier, and moves from one relation to another by navigating foreign keys backward, i.e., from the "one" side (the pointed primary key) to the "many" side (all tuples referring to such a key). The answers to the query are extracted for some of the attributes visited along the *navigational graph*. Finally, some of the so-obtained results can be filtered away by appending, at the end of the navigation, a query of the

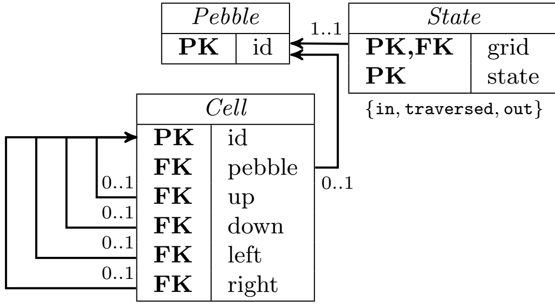


Fig. 8: A navigation-supporting data component where each case corresponds to a pebble, moving and modifying a grid (shared with other pebbles)

form $\Phi_{neg}(\mathbf{x}, \mathbf{y})$ above. An exception is constituted by cardinality-immutable relations, which can be queried arbitrarily (that is, without requiring a navigation driven by foreign-keys).

Example 2. Consider the navigation-supporting data component of Figure 8, where *Pebble* is the case relation. Each case represents a pebble, located over a cell in a (dynamic) planar grid. Each cell, in turn, may contain at most one pebble. We assume that the data component contains also constraints ensuring that its shape is in fact a grid. We do not model a specific case-centric DCDS running on top of such data component, but we instead focus on sample queries that are case navigational. The first query we consider is:

$$TRP(p) = Pebble(p) \land \exists c, d, l. Cell(c, p, \texttt{null}, d, l, \texttt{null})$$

This query returns the pebble that is located in the top-right corner of the grid (provided that there exists one). The query is case-navigational: it first selects a case p from the *Pebble* relation, then uses p to navigate to its cell c, checking that such a cell has no other cell on its right and on its top.

Another interesting example of case-navigational query is the following:

$$\begin{aligned}
&Surrounded(p, c) = Pebble(p)\\
&\land \exists u, d, l, r. Cell(c, p, u, d, l, r)\\
&\quad \land (\exists p_r, u_r, d_r, r_r. Cell(r, p_r, u_r, d_r, c, r_r) \land p_r \neq \texttt{null})\\
&\quad \land (\exists p_l, u_l, d_l, l_l. Cell(l, p_l, u_l, d_l, l_l, c) \land p_l \neq \texttt{null})\\
&\quad \land (\exists p_u, u_u, l_u, r_u. Cell(u, p_u, u_u, c, l_u, r_u) \land p_u \neq \texttt{null})\\
&\quad \land (\exists p_d, d_d, l_d, r_d. Cell(d, p_d, c, d_d, l_d, r_d) \land p_d \neq \texttt{null})
\end{aligned}$$

The query returns those pebbles (toghether with their corresponding locations) that are completely surrounded, i.e., have pebbles in each adjacent cell. The query is case navigational because it navigates from the *Pebble* relation to the *Cell* one, and then from the cell in which the pebble is located to all adjacent cells. Each subquery is paired with a negative filter guaranteeing that the considered adjacent cell indeed contains a pebble.

As last navigational query, consider

$$\begin{aligned}
&MoveR2(p, r_2) = Pebble(p)\\
&\land \left(\begin{array}{l}\exists c_0, u_0, d_0, l_0, r_0. Cell(c_0, p, u_0, d_0, l_0, r_0)\\ \land \left(\begin{array}{l}\exists u_1, d_1, l_1, r_1. Cell(r_0, \texttt{null}, u_1, d_1, l_1, r_1)\\ \land \left(\begin{array}{l}\exists u_2, d_2, l_2, r_2. Cell(r_1, \texttt{null}, u_2, d_2, l_2, r_2)\\ \land \left(\exists u_3, d_3, l_3, r_3. Cell(r_2, \texttt{null}, u_3, d_3, l_3, r_3)\right)\end{array}\right)\end{array}\right)\end{array}\right)
\end{aligned}$$

The query returns those pebbles that can move of two cells on the right, without incurring in any other pebble. The fact that the query is case navigational follows the intuition of navigating inside the grid starting from the position of the pebble, and moving twice on the right. Each right step is captured by navigating backwrd the foreign key that connects a cell with its right cell. ∎

Example 3. Consider again the archery training data component. Query

$$\begin{aligned}
Groups\&Members(g, n) = {} & Group(g) \land\\
& State(g, \texttt{running}) \land\\
\exists m, b. (& MerryM(m, n, b, \texttt{basic}, g)
\end{aligned}$$

is case-navigational over the archery training data component, and returns each known group, together with the names of those merry men that are apprentices in that group and have a basic combat level. Query

$$\begin{aligned}
GSharingS(g) = {} & Group(g) \land \exists s. (State(g, s) \land\\
& \exists g'. (Group(g') \land State(g', s) \land g' \neq g))
\end{aligned}$$

returns all those groups that are in the same state of at least another group. This query is *not* case-navigational over the archery training data component, because it starts navigating from case identifier g and then suddenly "jumps" to querying another case identifier g'. ∎

Given a data component $\mathcal{D}$, a case-navigational query $\Phi(\mathbf{x})$ over $\mathcal{D}$, and a weakly guarded query $\Psi(\mathbf{x'})$ with $\mathbf{x'} \subseteq \mathbf{x}$, we say that $\Psi(\mathbf{x'})$ is *navigationally embeddable* into $\Phi(\mathbf{x})$ if it is possible to replace a subquery $\Phi'(\mathbf{x'})$ of $\Phi(\mathbf{x})$ with $\Psi(\mathbf{x'})$, such that the resulting query is still case-navigational w.r.t. $\mathcal{D}$. This notion is useful to construct add and delete effects starting from a case-navigational query.

With all these notions at hand, we now transport the notion of case-navigation to condition-action rules and action effects accordingly. A case-centric DCDS $\langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ is *case-navigational* if, given $C = \textsc{case-rel}_{\mathcal{D}}$:

- $\mathcal{D}$ is navigation-supporting;
- ϱ_{norm} is constituted by condition action rules of the form $\Phi(c, \mathbf{x}) \mapsto \textsc{act}(c, \mathbf{x})$, where query $C(c) \land \Phi(c, \mathbf{x})$ is case-navigational w.r.t. $\mathcal{D}$.
- For every action $\textsc{act}(\mathbf{p})$ in $\mathcal{A}_{norm}$ and each effect $e \in \textsc{effect}(\textsc{act})$, e has the form

$$\bigvee_{i \in \{1, \ldots, n\}} \Phi_i(\mathsf{c}, \mathbf{p}, \mathbf{x}) \rightsquigarrow \textbf{add } A \textbf{ del } D$$

where, by considering parameters $\mathbf{p}$ as constants, we have that:

1. for each $i \in \{1, \ldots, n\}$, query $C(\mathsf{c}) \wedge \Phi_i(\mathsf{c}, \mathbf{p}, \mathbf{x})$ is case-navigational w.r.t. $\mathcal{D}$.[5]
2. For each $i \in \{1, \ldots, n\}$, and for each fact F in A, F is navigationally embeddable into $C(\mathsf{c}) \wedge \Phi_i(\mathsf{c}, \mathbf{p}, \mathbf{x})$, by considering the service calls in F as existentially quantified variables.
3. For each $i \in \{1, \ldots, n\}$, and for each fact F in D, we have that F is navigationally embeddable into $C(\mathsf{c}) \wedge \Phi_i(\mathsf{c}, \mathbf{p}, \mathbf{x})$.

The requirements on action effects ensure that the manipulation of data through add and delete facts does not work over parts of the database that are unrelated (in the sense of Section 3.2) to the case-navigational query in the body of the effect.

Example 4. The archery training DCDS can easily be turned into an equivalent form that makes it case-navigational. Consider for example the condition-action rule for the ADD-APPR action:

$$Group(g) \wedge State(g, \mathtt{in}) \\ \wedge \exists n, b, c.MerryM(m, n, b, c, \mathtt{null}) \mapsto \text{ADD-APPR}(m, g)$$

This query is already case-navigational: it navigates from the *Group* to the *State* relation, while *MerryM* can be freely queried, being cardinality-immutable. The fact that *MerryM* is cardinality-immutable also implies that the effect used to specify action ADD-APPR is also already in a case-navigational form. As for the other actions, they are already in case-navigational form, or can be directly made case-navigational by adding $Group(g)$ in front of the employed queries, which already use group g as entry point. ■

In general, case-navigationality guarantees that queries access the data in a well-disciplined way, always starting from a case identifier and retrieving tuples that directly or indirectly refer to it. In this light, the case identifier can be seen as a sort of "provenance identifier". This also tells us that when a process is not case-navigational because it contains relations that are disconnected from the case relation, a possible strategy to turn it into a case-navigational process is to augment such relations with a (non-null) "provenance column", in turn pointing via a foreing key to the case relation. Given one such relations, say, R, this modification ensures that every tuple of R now refers to one and only one case identifier, and that the provenance column can be used as the join column to access R in case-navigational way.

Unfortunately, being navigational does not suffice for decidability of soundness. In fact, we argue that the possibility of having unboundedly many tuples (indirectly) referring to a single case identifier, is a source of undecidability.

Theorem 2. *Checking data-aware soundness of case-navigational DCDSs is undecidable.*

[5] $C(\mathsf{c})$ is added just for compatibility with the definition, but it is ensured by construction, since parameter c points to C.

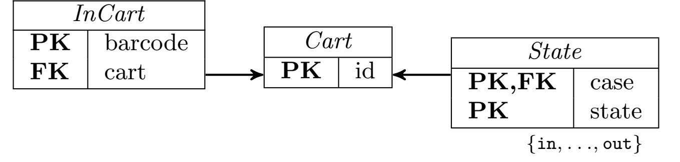


Fig. 9: A case-centric data model for user carts

Proof. Consider a case-centric DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ modeling a process that manipulates items in a cart. The schema and constraints of $\mathcal{D}$ are shown in Figure 9. Each case represent a user cart ($\text{CASE-REL}_{\mathcal{D}} = Cart$), and each cart is filled with products equipped with a barcode. The key aspect of $\mathcal{D}$ is that, for a given cart, *unboundedly many* products can be added to the cart. This features makes it possible to use $\mathcal{D}$ to realize a case-centric DCDS that models the behaviour of shoppers who do not have a limit on the number of products they intend to buy. In this light, $\mathcal{P}$ offers to shoppers the possibility of adding and removing items to/from the cart.

Specifically, a shopper who owns cart $\mathtt{c}$ can add a new product of type $\mathtt{t}$ to $\mathtt{c}$ using the following process fragment:

$$Cart(c) \mapsto \text{INS-PROD}(c)$$
$$\text{INS-PROD}(\mathsf{c}): \\ \{\mathsf{true} \rightsquigarrow \textbf{add } \{InCart(\textbf{newCode}_{InCart[barcode]}(), \mathsf{c})\}\}$$

Notice the usage of the $\textbf{newCode}_{InCart[barcode]}$ service call, which guarantees that the added product has a fresh barcode.

On the other hand, a shopper can also change her mind about a product that is currently in the cart. The following process fragment supports the shopper in the removal of a product from the cart:

$$Cart(c) \wedge InCart(p, c) \mapsto \text{REM-PROD}(p, c)$$
$$\text{REM-PROD}(\mathsf{p}, \mathsf{c}) : \{InCart(\mathsf{p}, \mathsf{c}) \rightsquigarrow \textbf{del } \{InCart(\mathsf{p}, \mathsf{c})\}\}$$

Given these two process fragments, it is quite easy to see that a shopper can straightforwardly simulate a counter machine, where:

- the value of the counter is represented by the number of items in the cart;
- the two actions INS-PROD and REM-PROD add and remove a tuple to/from the *InCart* relation, and thus they respectively encode the increment and decrement of the counter;
- testing whether the counter simulated by cart $\mathtt{c}$ is zero amounts to querying whether the extension of *InCart* is empty for $\mathtt{c}$: $IsEmpty(\mathsf{c}) = \neg\exists p.InCart(p, \mathsf{c})$. □

4.3 Case-Width-Bounded DCDSs

The source of undecidability in Theorem 2 relies on the possibility of having unboundedly many tuples re-

ferring (directly or indirectly) to a single case identifier. We therefore need to limit this source of unboundedness. To do so, we introduce a class of case-centric DCDSs that are *case-width-bounded*, by leveraging cardinality constraints in combination with key constraints (i.e., cardinality-bounded foreign key constraints in the sense of Section 2.1). Interestingly, when paired with the case-navigational requirement introduced before, this approach guarantees that, starting from a case identifier, each (backward) foreign key navigation has only boundedly many branches to follow.

Technically, a data component $\mathcal{D}$ is *case-width-bounded* if:

- $\mathcal{D}$ is case-navigational;
- each foreign key constraint $A[R] \longrightarrow B[S]$, where A is *not* cardinality-immutable, is paired with a corresponding cardinality constraint $\mathrm{CARD}(A[R], n..m)$, so that together they form a cardinality-bounded foreign key constraint $A[R] \xrightarrow{n..m} B[S]$.

Observe that for foreign keys pointing to a cardinality-immutable relation, no cardinality constraint need to be enforced, since the number of tuples in the source relation of the foreign key is anyway bounded by the number of tuples in the target relation, which does not change during the execution.

Correspondingly, we say that a case-centric DCDS is *case-width-bounded* if it is case-navigational and works over a data component that is case-width-bounded.

Example 5. The archery training schema of Figure 1 is case-width-bounded, since foreign keys are either paired with a corresponding cardinality constraint, or they point to a cardinality immutable relation. Specifically, foreign keys without corresponding cardinality constraints point to relations *MerryM* and *Trust*, which are both cardinality-immutable. Instead, foreign keys pointing to cardinality-mutable relations all have a numeric upper bound: every group has exactly one state, and at most two weekly slots. ∎

It is important to observe that, to ensure case-width-boundedness, also the standard foreign key from *State* to the case relation (cf. Figure 3) needs to be properly associated to a cardinality constraint. This is possible only when the control-flow component of the process gives raise to boundedly-many control states for the same case, which is typical in reality. In workflow net terms, this implies that the injection of a single "case token" into the net gives raise to a *bounded* marked net, which controls the maximum degree of concurrency induced by the case token.

We continue our (un)decidability tour by showing that being case-width-bounded is still not sufficient to ensure decidability, even when the width-bound is 1. The new source of undecidability is related to the possibility of recurring over cyclic foreign keys, in such a way that each navigation step has a bounded (actually, a single) branch, but, as a whole, it can reach unboundedly many values along a chain.

Theorem 3. *Checking data-aware soundness over case-width-bounded DCDSs is undecidable.*

Proof. To show undecidability, we take inspiration from the well-known *Snake*© videogame. In our variant, we consider a snake that, at the beginning, consists of just a snake head. Every time the snake finds a piece of food, it grows of one body part. Every time the snake hits a wall, it looses its tail body part (unless it only has its head - in this case nothing happens). We do not explicitly account for the position of the snake on the screen, nor for the actions used to move the snake, as they are not needed in the proof.

To formalize this behavior, we rely on the data component shown in Figure 10. In this data component, each case denotes (the head of) a snake: $\textsc{case-rel}_{\mathcal{D}} = \mathit{SnakeH}$. The *BodyPart* relation keeps track of the body parts of a snake. Each body part keeps a reference to the corresponding snake's head, and (optionally) a reference to its next part, that is, the part that comes next towards the head. The special control value `null` is used in the *next* column of the first body part, i.e., the body part that has the head as next.

It is easy to see that such a data component is case-width-bounded: each cardinality-bounded foreign key models a one-to-one relation.

The growth of snake `c` due to eating some food is captured by the following case-navigational action:

$$\textsc{eat-food}(\mathtt{c}) : \left\{\begin{array}{l} \neg\exists f, n.\mathit{BodyPart}(f,\mathtt{c},n) \rightsquigarrow \\ \qquad \mathbf{add}\{\mathit{BodyPart}(\mathbf{newBP}_{\mathit{BodyPart}[id]}(), \mathtt{c}, \mathtt{null})\} \\ \mathit{BodyPart}(f_o,\mathtt{c},\mathtt{null}) \rightsquigarrow \mathbf{del}\{\mathit{BodyPart}(f_o,\mathtt{c},\mathtt{null})\} \\ \qquad \mathbf{add}\left\{\begin{array}{l}\mathit{BodyPart}(\mathbf{newBP}_{\mathit{BodyPart}[id]}(), \mathtt{c}, \mathtt{null}), \\ \mathit{BodyPart}(f_o, \mathtt{c}, \mathbf{newBP}_{\mathit{BodyPart}[id]}())\end{array}\right\} \end{array}\right\}$$

The first effect deals with the case where the snake is currently without any body part (i.e., it just has a head). In this case, the first body part is created by calling the $\mathbf{newBP}_{\mathit{BodyPart}[id]}()$ service call so as to generate a fresh identifier for it.

The second effect deals instead with the case where snake `c` has already at least one body part. The query selects the first body part of `c`, namely the one that has `null` as next. The identifier of such a body part matches with variable f_o. The head of the effect acts then as follows:

- A new first body part is added, using the $\mathbf{newBP}_{\mathit{BodyPart}[id]}()$ service call to generate a fresh identifier for it.
- The old first body part is updated, making it the second one; this has the indirect effect of shifting all the other body parts one position forward.

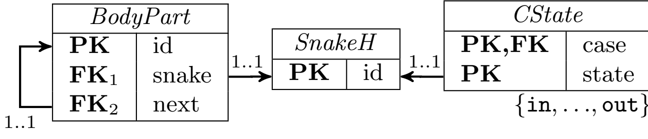


Fig. 10: A case-width-bounded data component modeling snakes

Symmetrically, the shrinking of snake c due to a wall hit is modeled by the following case-navigational action:

$$\text{HIT-WALL}(\mathtt{c}):\left\{\begin{array}{l} BodyPart(f_o,\mathtt{c},\mathtt{null}) \rightsquigarrow \textbf{del}\ \{BodyPart(f_o,\mathtt{c},\mathtt{null})\} \\ BodyPart(f_o,\mathtt{c},\mathtt{null}) \\ \wedge\ BodyPart(f_n,\mathtt{c},f_o) \rightsquigarrow \textbf{del}\ \{BodyPart(f_n,\mathtt{c},f_o)\} \\ \qquad\qquad\qquad\qquad \textbf{add}\{BodyPart(f_n,\mathtt{c},\mathtt{null}))\} \end{array}\right\}$$

The action deletes the current first body part, and at the same time updates the current second body part turning it into the first one. This has the indirect effect of shifting all the other body parts one position backward.

Given these two process fragments, it is easy to see that this DCDS can simulate a counter machine, where:

– The value of the counter is represented by the length (i.e., the number of body parts) of the snake.
– The two actions EAT-FOOD and HIT-WALL are respectively guaranteed to add and remove a body part to/from the snake (which means incrementing and decrementing the counter).
– Testing whether the counter encoded by snake c is zero amounts to querying whether there is no body part for c:

$$\neg\exists b, n. BodyPart(b, \mathtt{c}, n)$$

This concludes the proof. □

4.4 Case-Depth-Bounded DCDSs

The source of undecidability in Theorem 3 relies on the possibility of building unbounded chains of tuples, connected to each other via foreign keys. Therefore, the next necessary restriction is to "bound" the length of such chains. There are several ways of doing this. Following our methodological flavor, we choose a general approach that consists in checking whether the DCDS data component employs foreign keys safely.

Specifically, unbounded chains can be created by complying with the case-navigational property (that is, by navigating foreign keys) only if the foreign keys collectively form cyclic dependencies. This is the case of Figure 10, where the *BodyPart* relation cyclically depends on itself. In general, cycles may be formed by composing several foreign keys together.

We therefore characterize a class of data components guaranteeing the absence of such cycles. Technically, we say that a data component $\mathcal{D}$ is *case-depth-bounded* if:

– $\mathcal{D}$ is case-navigational.
– Whenever $\mathcal{D}$ contains a set of foreign key constraints of the form
  – $\text{CASE-REL}_{\mathcal{D}}[c] \hookleftarrow R_1[A_1] \hookleftarrow \ldots \hookleftarrow R_n[A_n] \hookleftarrow S_1[B_1]$
  – $S_1[B_1] \hookleftarrow \ldots \hookleftarrow S_m[B_m] \hookleftarrow S_1[B_1]$

  then at least one relation S_i (with $i \in \{1, \ldots, m\}$) is cardinality-immutable.

Intuitively, the mentioned set of foreign key constraints expresses that it is possible to navigate backward from the case relation, in such a way that the same relation is visited twice. If it is the case, then it is required that one of the relations involved in the cycle is cardinality-immutable. This indirectly rules out the possibility of having cycles that involve cardinality-mutable relations only.

In the following, we say that a tuple (indirectly or directly) *refers to* a case identifier if the tuple can be retrieved using a case-navigational query starting from that case identifier. This means that there is a chain of tuples going from the case identifier to the target tuple, such that every tuple in the chain refers to the previous one via a foreign key.

The key property guaranteed by a case-depth-bounded data component is that it is never possible to write a case-navigational query that visits the same relation twice along a path, unless the query visits a cardinality-immutable relation in between. This, in turn, implies the following interesting intermediate result.

Lemma 1. *No case-navigational DCDS working over a case-depth-bounded data component can create a chain of tuples of unbounded length, where the source of the chain is a Case tuple, and each other tuple refers to the previous one via a foreign key.*

Proof. Suppose, by absurdum, that there is a DCDS that can create such a chain. Since the chain is unbounded but a data component has a fixed schema consisting of finitely many relations, the chain must contain unboundedly many tuples for some relations. Due to the case-navigational assumption, the DCDS can create such unboundedly many tuples only via navigational effects. This, in turn, can be done only if there is a cyclic composition of foreign key constraints in the DCDS data component. Let $S_1, \ldots, S_m$ be the relations involved in such a cycle. Then the chain must cyclically contain a sequence of m fresh tuples for each S_i (with $i \in \{1, \ldots, m\}$), repeated unboundedly many times. This, in turn, implies that each such relation has unboundedly many tuples along the chain. However, from the definition of case-depth-bounded data component, at least one out of these relations must be cardinality-immutable. The extension of cardinality-immutable relations is fixed and cannot be increased, therefore it is impossible to have unboundedly many tuples for a cardinality-immutable relation along the chain. This contradicts the hypothesis. □

An effective way to detect cyclic chains of foreign keys and rule them out is to see the schema and constraints of a data component $\mathcal{D}$ as a *FK-graph* $G_\mathcal{D}$, that is, a tuple $\langle N, n_0, E\rangle$, where:

- N is a set of nodes, which contains a node for each relation in the database schema of $\mathcal{D}$.
- $n_0 \in N$ is the node corresponding to the case relation $\textsc{case-rel}_\mathcal{D}$.
- E is a set of edges, each of which has the form $\langle R_1, R_2, id\rangle$, where $R_1, R_2 \in N$ and id is a unique identifier. Formally, E is the minimal set satisfying the following condition: for each foreign key constraint $R_i[A_i] \longrightarrow R_j[B_j]$ of $\mathcal{D}$, E contains a dedicated tuple $\langle R_i, R_j, id\rangle$, where id is a fresh identifier.

We use the standard notation $R_i \longrightarrow R_j$ if there exists an edge identifier id such that $\langle R_i, R_j, id\rangle \in E$, and $\stackrel{+}{\longrightarrow}$ to denote that a node can reach another node via a path in the graph.

With this notion at hand, we have that a data component $\mathcal{D}$ is case-depth-bounded if in $G_\mathcal{D} = \langle N, n_0, E\rangle$ there is no node $n \in N$ such that: *(i)* $n_0 \stackrel{+}{\longrightarrow} n$, and *(ii)* $n \stackrel{+}{\longrightarrow} n$ without ever passing through a cardinality-immutable relation. This property can be checked in NLogSpace in the size of the graph (i.e., in the size of the data component $\mathcal{D}$), since it consists of a sequence of two reachability checks.

Figure 11 provides examples of foreign-key graphs. It is easy to see that the data component used in the proof of Theorem 3 is not case-depth-bounded (cf. Figure 11b).

4.5 Case-Bounded DCDSs

We now consider the interaction between width- and depth-boundedness. Recall that both are necessary towards decidability, as soundness turned out to be undecidable for width-bounded but depth-unbounded systems (cf. Theorem 3), as well as for width-unbounded and depth-bounded systems (cf. Theorem 2).

In particular, we introduce the class of *case-bounded* DCDSs, i.e., DCDSs that are simultaneously width-bounded and depth-bounded. For this class of DCDSs, the following interesting intermediate result holds.

Lemma 2. *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P}\rangle$ be a case-bounded DCDS with $\Upsilon_\mathcal{S} = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow\rangle$. Then there exists an overall bound $b \in \mathbb{N}$ such that, for every state $s \in \Sigma$ and every case-tuple $\textsc{case-rel}_\mathcal{D}(\mathtt{c}) \in db(s)$, the number of tuples referring to $\mathtt{c}$ is bounded by b.*

Proof. Let $\mathcal{D} = \langle \Delta, \mathcal{R}, \mathcal{C}, \mathcal{I}_0\rangle$. Being case-bounded, $\mathcal{S}$ is case-width-bounded and case-depth-bounded. Thanks to case-width-boundedness, a DCDS can create at most $k \cdot W$ tuples directly referring to a given tuple, where: *(i)* k is the number of foreign keys in the schema (which bounds the number of foreign keys that point to the same relation), and *(ii)* W is the maximum upper bound of all the cardinality constraints in $\mathcal{D}$.

Thanks to case-depth-boundedness, there is also a bound on the length of the chains of tuples that can be created by $\mathcal{S}$ starting from a case identifier and using navigational effects (cf. Lemma 1. Now let T the maximum number of tuples of a cardinality-immutable relation in $\mathcal{I}_0$. Since this number remains constant along any evolution of $\mathcal{S}$, the number of tuples that can be chained is bounded by $T \cdot D$, where D is the length of the longest simple path in $G_\mathcal{D}$. This upper bound is obtained by iterating T times through a loop that consists of D relations - this cannot be exceeded, since the number of tuples present in any cardinality-immutable relation corresponds, by definition, to T.

Putting everything together, we obtain that the number of tuples that can refer to a single case identifier is bounded by $b = (k \cdot W)^{(D \cdot T)+1}$. □

Unfortunately, despite the boundedness result of Lemma 2, unbounded chains can still be obtained by creating subtle interactions among different cases.

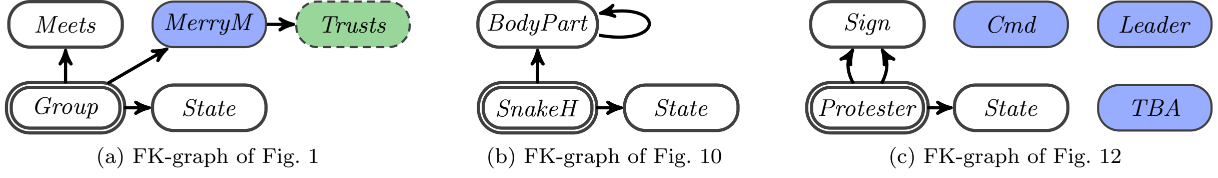
Theorem 4. *Checking soundness over case-bounded DCDSs is undecidable.*

Proof. The proof takes inspiration from the proof of Theorem 4.6 in [11], and is more involved than the other undecidability proofs in this work, since it is not true anymore that each single case can simulate a 2-counter machine, but cases must collectively cooperate to do so.

To illustrate the proof, we use the metaphor of a pacific protest and its demonstrators, and construct a DCDS that manages a protest by simulating a 2-counter machine. As shown in Figure 12, the data layer for this setting maintains data about protesters (where each protester is a case) and about protest signs that protesters held together. Each protester can hold from zero to two signs (one per hand), and each protest sign can be held by one or two protesters (one with her right hand, the other with her left hand). Each sign tracks the (unique) message contained in the sign, together with the identifiers of the protesters who respectively stand on the left and on the right of the sign (using `null` if one is missing). Two cardinality-immutable relations are used to keep track of common data for all protesters: the id of the leader, and the current "command" to be collectively executed. The leader table contains a single tuple, initialized to some random value and then overridden as soon as the first protester is created (i.e., the first protester becomes the leader of the protest). It is easy to see that this data component is both width- and depth-bounded.

The manipulation of the two counters corresponds, in our protest metaphor, to the organization of the protest with two chains of protesters on the left and on the right of the leader, each constituted by an alternation of protesters and signs.

The joint command relation contains a code representing the current command to be executed - in the



(a) FK-graph of Fig. 1 (b) FK-graph of Fig. 10 (c) FK-graph of Fig. 12

Fig. 11: Examples of FK-graphs; double-line nodes correspond to the case relation, blue nodes with thin line to cardinality-bounded relations, and green nodes with thin, dashed line to read-only relations

context of the 2-counter machine reduction, this corresponds to the program counter.

When the first protester is created, a dedicated action of the DCDS is used to introduce two signs into the system, making sure that the first sign has the leader on the right and nobody, i.e., `null`, on the left, whereas the second has the leader on the left, and nobody on the right. The action also marks that the first protester is actually a leader, and that it is currently alone, i.e., the last of the queue on the left and on the right. The two initial signs hold by the leader represent the zero elements of the two counters. In the following, we focus on the manipulation of the second counter (related to the right-chain starting from the leader). The manipulation of the first counter is managed symmetrically.

To test whether the second counter is zero, we can in fact use the following case-navigational query[6]:

$$\mathit{Leader}(l) \wedge \mathit{Protester}(l) \wedge \exists m.\mathit{Sign}(m, l, \mathtt{null})$$

The increment of the second counter corresponds to the addition of a protester to the chain. The fact that the chain is meant to be extended is signalled by a specific value `ext` in the *Cmd* relation.

To be added, a protester must be un-supplied with signs. The addition consists of three steps. To indirectly synch, the different protesters involved in the addition mark the *phase* column of the *TBA* (to-be-added) relation with three different constants `0`, `1`, and `2`. The last constant `2` represents that the addition protocol was executed in the past, and that the *id* column contains old data.

In the first step, a protester with no signs nondeterministically offers to be added to the chain, provided that nobody else already did (i.e., the *phase* column of *TBA* has value `2`). This is done by suitably querying and updating the cardinality-immutable *TBA* (to-be-added) relation. In particular, if the second component of *TBA* is `true`, then it means that nobody offered for the current addition.

$$\mathit{Protester}(p) \wedge \neg(\exists m, p_2.\mathit{Sign}(m, p_2, p) \vee \mathit{Sign}(m, p, p_2)) \wedge \mathit{Cmd}(\mathtt{ext}) \wedge (\exists p_2.\mathit{TBA}(p_2, \mathtt{2})) \mapsto \textsc{offer}(p)$$

$$\textsc{offer}(\mathsf{p}) : \left\{ \begin{array}{l} \exists p_2.\mathit{TBA}(p_2, \mathtt{2}) \rightsquigarrow \mathbf{del}\{\mathit{TBA}(p_2, \mathtt{2})\} \\ \qquad\qquad\qquad\quad \mathbf{add}\{\mathit{TBA}(\mathsf{p}, \mathtt{0})\} \end{array} \right\}$$

[6] Recall that *Leader* can be freely queried, being cardinality-immutable.

The second step is executed by the rightmost protester of the chain, and consists in making the protester in the *id* column of the *TBA* relation the new rightmost protester.

By noting that the rightmost protester can be univocally identified by checking that she holds a sign with her right hand, but there is nobody standing on the right of the same sign, the formalization of this process fragment is then as follows:

$$\mathit{Protester}(p) \wedge (\exists m.\mathit{Sign}(m, p, \mathtt{null})) \wedge \mathit{Cmd}(\mathtt{ext}) \wedge \mathit{TBA}(n, \mathtt{0}) \mapsto \textsc{ins}(p, n)$$

$$\textsc{ins}(\mathsf{p}, \mathsf{n}) : \left\{ \begin{array}{rl} \mathit{Sign}(m, \mathsf{p}, \mathtt{null}) \rightsquigarrow & \mathbf{del}\{\mathit{Sign}(m, \mathsf{p}, \mathtt{null})\} \\ & \mathbf{add}\{\mathit{Sign}(m, \mathsf{p}, \mathsf{n})\} \\ \mathit{TBA}(\mathsf{p}, \mathtt{0}) \rightsquigarrow & \mathbf{del}\{\mathit{TBA}(\mathsf{p}, \mathtt{0})\} \\ & \mathbf{add}\{\mathit{TBA}(\mathsf{p}, \mathtt{1})\} \end{array} \right\}$$

The last step of the addition is then again under the responsibility of the just inserted protester, who now needs to prepare a new sign to be placed on her own right. This has the effect of incrementing by 1 the number of signs on the right of the leader. At the same time, the protester also updates the *TBA* relation and the *Cmd* relation (the latter depending on the specific strategy adopted by the protest, i.e., by the specific DCDS at hand).

$$\mathit{Protester}(p) \wedge \mathit{Cmd}(\mathtt{ext}) \wedge \mathit{TBA}(p, \mathtt{1}) \mapsto \textsc{prep}(p)$$

$$\textsc{prep}(\mathsf{p}) : \left\{ \begin{array}{rl} \mathsf{true} \rightsquigarrow & \mathbf{add}\{\mathit{Sign}(\mathbf{newMsg}_{\mathit{Sign}[msg]}(), \mathsf{p}, \mathtt{null})\} \\ \mathit{TBA}(\mathsf{p}, \mathtt{1}) \rightsquigarrow & \mathbf{del}\{\mathit{TBA}(\mathsf{p}, \mathtt{1})\}\ \mathbf{add}\{\mathit{TBA}(\mathsf{p}, \mathtt{2})\} \\ \mathit{Cmd}(\mathtt{ext}) \rightsquigarrow & \mathbf{del}\{\mathit{Cmd}(\mathtt{ext})\}\ \mathbf{add}\{\mathit{Cmd}(\ldots)\} \end{array} \right\}$$

The decrement of the second counter is much easier. It corresponds to the contraction of the chain on the right of the leader, i.e., to the removal of the rightmost protester from the chain. This is signalled by the presence of the `contr` constant inside the cardinality-immutable *Cmd* relation. The removal just consists in affirming that the sign currently standing at the left of the rightmost protester will not have anybody on its right, and that the rightmost protester will loose the sign standing on her own right (this will make her eligible for a future insertion in the chain).

$$\mathit{Protester}(p) \wedge \exists m, l.(\mathit{Sign}(m, l, p) \wedge l \neq \mathtt{null}) \wedge \exists m.(\mathit{Sign}(m, p, \mathtt{null}) \wedge \mathit{Cmd}(\mathtt{contr}) \mapsto \textsc{rem}(p)$$

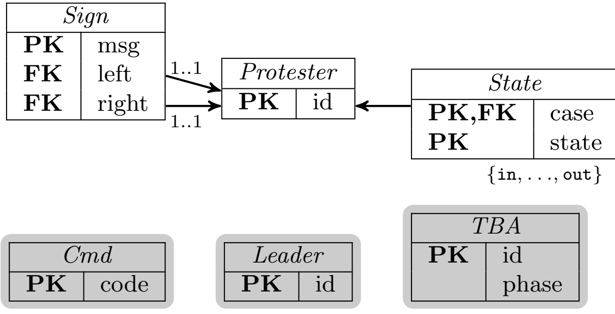


Fig. 12: A case-bounded data model for protesters and protest signs

$$\text{REM}(\mathsf{p}) : \left\{ \begin{array}{rl} \mathit{Sign}(m,l,\mathsf{p}) \rightsquigarrow & \mathbf{del}\{\mathit{Sign}(m,l,\mathsf{p})\} \\ & \mathbf{add}\{\mathit{Sign}(m,l,\texttt{null})\} \\ \mathit{Sign}(m,\mathsf{p},\texttt{null}) \rightsquigarrow & \mathbf{del}\{\mathit{Sign}(m,\mathsf{p},\texttt{null})\} \\ \mathit{Cmd}(\texttt{ext}) \rightsquigarrow & \mathbf{del}\{\mathit{Cmd}(\texttt{contr})\} \\ & \mathbf{add}\{\mathit{Cmd}(\ldots)\} \end{array} \right\}$$

It is worth noting that, thanks to the condition-action rule for REM, it is never possible to apply it on the leader.

We can now see the evolution of the protest as a 2-counter machine, where the value of the first (resp., second) counter is represented by the number of signs present on the left (resp., right) of the leader, minus 1. The zero element of the first (resp., second) counter is actually represented by the sign on the left (resp., right) by the leader. The evolution relies on the fact that unboundedly many protesters (representing cases) are nondeterministically created, and that the addition and removal into/from the chain are made possible by indirect synchronizations working over cardinality-immutable relations. The input 2-counter machine can be modeled by properly deciding how to update the content of the *Cmd* relation, so as to mimic the program of the 2-counter machine. The halting of such machine is reduced to soundness of the corresponding DCDS by just ensuring that when the halting state is reached, all protesters move into the `out` state.

4.6 Case-Isolated DCDSs

The undecidability result of Theorem 4 is due to the lack of isolation across cases: when a single relation can be achieved navigationally from two distinct cases, such relation can be used to implicitly transfer information among cases, and in turn coordinate their evolution.

We consequently need to limit this interaction. We do so by introducing a suitable notion of isolation, which guarantees that the data associated to some case are not touched by other cases. Technically, we say that a data component $\mathcal{D}$ is *case-isolated* if:

- $\mathcal{D}$ is case-navigational.
- Whenever $\mathcal{D}$ contains a set of foreign key constraints of the form
  - $\text{CASE-REL}_{\mathcal{D}}[c] \longleftarrow R^1_1[A^1_1] \longleftarrow \ldots \longleftarrow R^1_n[A^1_n] \longleftarrow R[A]$
  - $\text{CASE-REL}_{\mathcal{D}}[c] \longleftarrow R^2_1[A^2_1] \longleftarrow \ldots \longleftarrow R^2_k[A^2_k] \longleftarrow R[A]$

  such that $R^1_i[A^1_i] \neq R^2_j[A^2_j]$ for some $i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,k\}$, then R is a read-only relation.

Intuitively, the mentioned set of foreign key constraints expresses that it is possible to find two distinct backward foreign-keys navigation paths that start from the case relation and reach the same relation. If it is the case, then it is required that this latter relation is read-only (and consequently cannot be used to transfer information from one case to the other). This indirectly rules out the possibility of having multiple paths pointing to the same read-write relation.

Ad for case-depth-boundedness, we can reformulate case-isolation of a data component $\mathcal{D}$ as a property over its corresponding FK-graph $G_{\mathcal{D}}$. In particular, we have that $\mathcal{D}$ is case-isolated if, whenever in $G_{\mathcal{D}}$ a node n can be reached from n_0 along two different paths (that is, paths that differ for at least one node), then n corresponds to a read-only relation. It is easy to see that the data component used in the proof of Theorem 4 is not case-isolated (cf. Figure 11c).

Case-isolation implies the following interesting intermediate result.

Lemma 3. *Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a case-isolated DCDS with $\Upsilon_{\mathcal{S}} = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$. Then, for every state $s \in \Sigma$ and every relation R in $\mathcal{D}$ that is not read-only, each R-tuple in $db(s)$ refers to at most one case identifier.*

Proof. Suppose, by absurdum, that there is a state $s \in \Sigma$, case identifiers $\mathtt{c}_1$ and $\mathtt{c}_2$, and a non-read-only, n-ary relation R such that:

- $\{\text{CASE-REL}_{\mathcal{D}}(\mathtt{c}_1), \text{CASE-REL}_{\mathcal{D}}(\mathtt{c}_2)\} \subseteq db(s)$;
- there exists a tuple $\mathbf{v}$ of n values in Δ, for which $R(\mathbf{v}) \in db(s)$;
- $R(\mathbf{v})$ refers to both $\mathtt{c}_1$ and $\mathtt{c}_2$.

Since, by definition, a case-centric DCDS has no case in its initial database, then s must have been produced after the application of a sequence of actions, during which: *(i)* $\mathtt{c}_1$ and $\mathtt{c}_2$ have been created, and *(ii)* $R(\mathbf{v})$ has been inserted or updated making it referring to $\mathtt{c}_1$ and $\mathtt{c}_2$. Since by hypothesis $\mathcal{S}$ is case-isolated, hence also case-navigational, the latter condition implies the existence of two *different* chains of tuples, created by $\mathcal{S}$ using navigational effects, of the form:

$$R(\mathbf{v}), R^i_1(\mathbf{v}^i_1), \ldots, R^i_n(\mathbf{v}^i_n), \text{CASE-REL}_{\mathcal{D}}(\mathtt{c}_i)$$

for $i \in \{1,2\}$, where each tuple points to the next tuple via a foreign key. The existence of two different chains of this form, contructed by the DCDS using navigational effects, implies that $\mathcal{D}$ contains two different sets of foreign key constraints of the form:

$$R[A] \longrightarrow R^i_1[A^i_1] \longrightarrow \ldots \longrightarrow R^i_n[A^i_n] \longrightarrow \text{CASE-REL}_{\mathcal{D}}[id]$$

for $i \in \{1,2\}$. However, this in turn implies that in the FK-graph $G_{\mathcal{D}}$ there are two paths leading from R to $\text{CASE-REL}_{\mathcal{D}}$, which contradicts the hypothesis that $\mathcal{S}$ is case-bounded, hence also case-isolated. $\square$

4.7 Decidability, at Last

As we have shown in the previous sections, case-isolation and case-boundedness do not suffice alone for achieving decidability of soundness. In fact, all the proofs of the undecidability theorems in this work (with the exception of Theorem 4) employ a DCDS that is case-isolated but not case-bounded, whereas the one of Theorem 4) employs a DCDS that is case-bounded but not case-isolated.

In fact, the key towards decidability of data-aware soundness is the combination of case-isolation and case-boundedness.

Theorem 5. *Checking soundness of case-isolated, case-bounded DCDSs is decidable, and reducible to conventional model checking of propositional μ-calculus.*

Proof. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a case-centric DCDS that is both case-isolated and case-bounded. Thanks to case-isolation, $\mathcal{S}$ evolves each case (and related tuples) in a way that is completely independent from that of the other cases – cf. Lemma 2. As a consequence, the soundness of a single case is not affected by the evolution of the other cases. Since by definition a case-centric DCDS starts with an empty extension for its case relation, every case of $\mathcal{S}$ is injected into the system through the execution of the NEW-CASE action, which has a `true` precondition and treats every case homogeneously: it simply puts every case into the initial, `in` state. This, in turn, implies that *all cases are sound if and only if one case is sound.* We can then replace the usual condition-action rule

$$\mathsf{true} \mapsto \text{NEW-CASE}()$$

with the following rule, which ensures that only one single case is injected into the system:

$$\neg\text{CASE-REL}_{\mathcal{D}}(_) \mapsto \text{NEW-CASE}()$$

Let $\widehat{\mathcal{S}}$ be DCDS obtained from $\mathcal{S}$ with such a replacement. Recalling that the (data-aware) soundness property can be formalized in $\mu\mathcal{L}_P$ as Φ_{sound} (cf. Section 3.3), we obtain that $\Upsilon_{\mathcal{S}} \models \Phi_{sound}$ if and only if $\Upsilon_{\widehat{\mathcal{S}}} \models \Phi_{sound}$. We hence consider the model checking problem $\Upsilon_{\widehat{\mathcal{S}}} \models \Phi_{sound}$.

Thanks to the modified version of the NEW-CASE condition-action rule, in $\Upsilon_{\widehat{\mathcal{S}}}$ relation $\text{CASE-REL}_{\mathcal{D}}$ is guaranteed to contain at most one tuple. The claim is then obtained by observing that:
- since $\mathcal{S}$ is case-bounded by hypothesis, we can apply Lemma 2, consequently obtaining that $\widehat{\mathcal{S}}$ is state-bounded in the sense of [2].
- By [2], we know that verification of $\mu\mathcal{L}_P$ properties over state-bounded DCDSs is decidable and reducible to conventional model checking techniques, hence so is deciding whether $\Upsilon_{\widehat{\mathcal{S}}} \models \Phi_{sound}$. □

Notably, the Robin Hood archery training DCDS illustrated in Section 2.3 is both case-isolated and case-bounded. Hence, Theorem 5 guarantees that soundness can be checked over it.

By inspecting the proof of Theorem 5, we can make two important observations. The first one is related to the generality and robustness of the result. In our setting, we are considering the specific property of soundness, but since decidability is obtained by a reduction to finite-state model checking with the technique in [2], it actually holds more in general for all temporal properties that:
1. can be expressed in $\mu\mathcal{L}_P$;
2. obey to case isolation, that is, they can be verified without considering the interplay between different cases.

The second observation concerns the computational complexity of the problem. The abstraction technique in [2] provides us an immediate upper bound for checking soundness of case-isolated, case-bounded DCDSs. This upper ubound is ExpTime in the size of the initial DCDS[7], by considering the arity of the relations bounded by a constant. This complexity result carries over to arbitrary $\mu\mathcal{L}_P$ properties of the aforementioned form. As for soundness, the complexity can actually be tightened to PSpace, by observing that soundness itself can be expressed in the fragment of first-order CTL with persistent quantification (which can be seen as a proper fragment of $\mu\mathcal{L}_P$). This, in turn, makes it possible to perform verification by constructing the (abstract) transition system on-the-fly, leveraging the well-known techniques for CTL [5].

We close our investigation with three general observations on our decidability result. First of all, it is interesting to notice that, by restricting the attention to read-write relations only, the combination of case-isolation and case-boundedness requires that FK-graphs have the shape of a tree, rooted in the case relation.[8] This, in turn, implies that case-navigational queries are tree-shaped (with a negative filter). This way of structuring data (and corresponding queries) resembles document-centric processes, where each process case works over a dedicated business object, separately from the other cases. In our framework, such a business object is stored in a nested-tuple relational data structure, a solution that is adopted by concrete data-centric system, such as BizArtifact[9].

Second, we stress the novelty of the decidability result of Theorem 5: while all the key decidability results for DCDSs and similar frameworks have been provided, so far, under the hypothesis of state boundedness, Theorem 5 holds for a class of DCDSs in which relations are

[7] We ignore the contribution of the countably infinite data domain.

[8] Read-only and cardinality-immutable relations constitute an exception to this.

[9] `http://sourceforge.net/projects/bizartifact/`

unbounded. Decidability is obtained thanks to the fact that, despite unboundedness, such relations are populated by composing an unbounded number of bounded data slices, each originating from a single case identifier. This reflects the methodological guideline of horizontal partitioning, in the sense discussed in Section 3.2.

Finally, the proof of Theorem 5 shows that case-bounded, case-isolated DCDSs can be correctly verified by shrinking their unboundedly many cases into a single, prototypical case. This can be seen as the data-aware counterpart of soundness in workflow nets [29], which is checked by introducing a single, prototypical token into the input state of the process.

5 Conclusion

We have investigated the classical notion of case-centric processes, but in a rich, data-aware setting, where we fully take into account how the process control-flow operates over a full-fledged, relational database with constraints. We have also reformulated the standard correctness notion of soundness accordingly. To ground our study, we have considered Data-Centric Dynamic Systems (DCDSs) as a reference framework.

The main result of the paper is that checking whether a case-centric DCDS is sound can be decided, under the hypothesis that each case evolves independently from the other, and that each single case cannot generate unboundedly many data directly or indirectly referring to it. Notably, decidability is obtained by showing how to find a cutoff on the number of process instances that must be subject to the soundness test, having the guarantee that soundness transfers to the entire system, where not limit is imposed on the number of process instances that can be injected.

It is important to notice that this class of DCDSs is characterized syntactically, and can be naturally paired with a set of methodological guidelines supporting the modeler during the design phase. In this light, the results presented here depart from those in [6, 2, 4]; these rely on the semantic properties of run- and state-boundedness, which are highly undecidable to check, and for which only syntactic, sufficient conditions can be provided [3]. It is also important to observe that, by recasting the results in [11] in our setting, one can show that all the modeling restritions we require towards decidability are indeed tight, i.e., soundness turns out to be undecidable if we relax any of them.

To the best of our knowledge, the only other work that considers verification of unbounded, data-aware processes is [21]. Differently from this work, however, [21] aims at providing a general tight decidability result by just controlling the number and arity of relations. The result is theoretically interesting but of very limited practical applicability, as decidability is provided for systems working over a single, unary relation only.

We plan to continue this line of research along two directions. First of all, we want to merge this investigation with that of [11], on the one hand to generalize our decidability result to the case of navigational $\mu\mathcal{L}_P$ properties, and on the other hand to go beyond case-centric processes, and consider rich artifact-centric systems in which dynamic entities co-evolve by establishing mutual many-to-many relations in a controlled way. Second, we want to study the possibility of practically implementing the presented techniques, in particular by attacking the exponentiality in the data that comes with data-aware dynamic systems through modularization techniques, taking inspiration from horizontal and vertical partitioning as introduced here.

Acknowledgements

The authors are partially supported by the EU project *Optique* (FP7-IP-318338), and by the UNIBZ internal projects *KENDO* and *OnProm*.

References

1. P. A. Abdulla, M. F. Atig, A. Kara, and O. Rezine. Verification of dynamic register automata. In *Proc. of the 44th Int. Conf. On Foundation of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 29 of *LIPIcs*, pages 653–665. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
2. B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, 2013.
3. B. Bagheri Hariri, D. Calvanese, A. Deutsch, and M. Montali. State-boundedness for decidability of verification in data-aware dynamic systems. In *14th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*. AAAI Press, 2014.
4. B. Bagheri Hariri, D. Calvanese, M. Montali, G. De Giacomo, R. De Masellis, and P. Felli. Description logic Knowledge and Action Bases. *Journal of Artificial Intelligence Research*, 2013.
5. C. Baier, J.-P. Katoen, and K. Guldstrand Larsen. *Principles of Model Checking*. The MIT Press, 2008.
6. F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, 2012.
7. M. Bojanczyk, L. Braud, B. Klin, and S. Lasota. Towards nominal computation. In *39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. ACM Press, 2012.
8. M. Bojanczyk, L. Segoufin, and S. Torunczyk. Verification of database-driven systems via amalgamation. In *32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, 2013.

9. D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data-aware process analysis: A database theory perspective. In *32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. ACM Press, 2013.
10. D. Calvanese, G. De Giacomo, M. Montali, and F. Patrizi. Verification and synthesis in description logic based dynamic systems. In *7th Int. Conf. on Web Reasoning and Rule Systems (RR)*, volume 7994 of *LNCS*. Springer, 2013.
11. D. Calvanese, M. Montali, E. Montserrat, and E. Teniente. Verifiable UML artifact-centric business process models. In *23rd ACM Int. Conf. on Information and Knowledge Management (CIKM)*, 2014. To appear.
12. A. Chandra and D. Harel. Computable queries for relational database systems. *J. Comput. Syst. Sci.*, 21, 1980.
13. M. Dumas. On the convergence of data and process engineering. In *15th Int. Conf. on Advances in Databases and Information Systems (ADBIS)*, volume 6909 of *LNCS*, pages 19–26. Springer, 2011.
14. M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers. *Fundamentals of Business Process Management.* Springer, 2013.
15. D. Fahland, M. de Leoni, B. F. van Dongen, and W. M. P. van der Aalst. Behavioral conformance of artifact-centric process models. In *14th Int. Conf. on Business Information Systems (BIS)*, volume 87 of *LNCS*. Springer, 2011.
16. D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data and Knowledge Engineering*, 70(5), 2011.
17. M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5), 2002.
18. R. Hüchting, R. Majumdar, and R. Meyer. A theory of name boundedness. In *Proc. of the 24th Int. Conf. On Concurrency Theory (CONCUR)*, volume 8052 of *LNCS*, pages 182–196. Springer, 2013.
19. R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *On the Move Confederated Int. Conf. (OTM)*, volume 5332 of *LNCS*. Springer, 2008.
20. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *22nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, pages 1–14, 2003.
21. A. Lomuscio and J. Michaliszyn. Model checking unbounded artifact-centric systems. In *14th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*. AAAI Press, 2014.
22. R. Meyer. On boundedness in depth in the pi-calculus. In *Proc. of the 5th IFIP Int. Conf. On Theoretical Computer Science (TCS)*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.
23. M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., 1967.
24. M. Montali, D. Calvanese, and G. De Giacomo. Verification of data-aware commitment-based multiagent system. In *13th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 2014.
25. U. Montanari and M. Pistore. History-dependent automata: An introduction. In *5th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-Moby)*, volume 3465 of *LNCS*. Springer, 2005.
26. R. Needham. *Distributed Systems*, chapter Names. Addison Wesley Publ. Co., 1989.
27. F. Rosa-Velardo and D. de Frutos-Escrig. Decidability and complexity of petri nets with unordered data. *Lecture Notes in Theor. Comp. Science*, 412(34), 2011.
28. D. Solomakhin, M. Montali, S. Tessaris, and R. De Masellis. Verification of artifact-centric systems: Decidability and modeling issues. In *Proc. of the 11th Int. Conf. on Service Oriented Computing (ICSOC)*, 2013.
29. W. M. P. van der Aalst. Verification of workflow nets. In *18th Int. Conf. on Application and Theory of Petri Nets (ICATPN)*, volume 1248 of *LNCS*, pages 407–426. Springer, 1997.
30. W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Asp. Comput.*, 23(3), 2011.
31. V. Vianu. Automatic verification of database-driven systems: a new frontier. In *12th Int. Conf. on Database Theory (ICDT)*, 2009.
32. M. Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer, 2007.