Petri Nets with Parameterised Data
Modelling and Verification

Silvio Ghilardi[1], Alessandro Gianola[2,3], Marco Montali[2],
and Andrey Rivkin[2(✉)]

[1] Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy
`silvio.ghilardi@unimi.it`
[2] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy
`{gianola,montali,rivkin}@inf.unibz.it`
[3] CSE Department, University of California San Diego (UCSD),
San Diego, CA, USA
`agianola@eng.ucsd.edu`

Abstract. During the last decade, various approaches have been put forward to integrate business processes with different types of data. Each of these approaches reflects specific demands in the whole process-data integration spectrum. One particularly important point is the capability of these approaches to flexibly accommodate processes with multiple cases that need to co-evolve. In this work, we introduce and study an extension of coloured Petri nets, called catalog-nets, providing two key features to capture this type of processes. On the one hand, net transitions are equipped with guards that simultaneously inspect the content of tokens and query facts stored in a read-only, persistent database. On the other hand, such transitions can inject data into tokens by extracting relevant values from the database or by generating genuinely fresh ones. We systematically encode catalog-nets into one of the reference frameworks for the (parameterised) verification of data and processes. We show that fresh-value injection is a particularly complex feature to handle, and discuss strategies to tame it. Finally, we discuss how catalog-nets relate to well-known formalisms in this area.

1 Introduction

The integration of control flow and data has become one of the most prominently investigated topics in BPM [25]. Taking into account data when working with processes is crucial to properly understand which courses of execution are allowed [11], to account for decisions [5], and to explicitly accommodate business policies and constraints [13]. Hence, considering how a process manipulates underlying volatile and persistent data, and how such data influence the possible courses of execution within the process, is central to understand and improve how organisations, and their underlying information systems, operate throughout the entire BPM lifecycle: from modelling and verification [10,18] to enactment [19,21] and mining [2]. Each of such approaches reflects specific demands in the whole process-data integration spectrum. One key point is the capability

of these approaches to accommodate processes with multiple co-evolving case objects [4,14]. Several modelling paradigms have adopted to tackle this and other important features: data-/artifact-centric approaches [10,18], declarative languages based on temporal constraints [4], and imperative, Petri net-based notations [14,22,24].

With an interest in (formal) modelling and verification, in this paper we concentrate on the latter stream, taking advantage from the long-standing tradition of adopting Petri nets as the main backbone to formalise processes expressed in front-end notations such as BPMN, EPCs, and UML activity diagrams. In particular, we investigate for the first time the combination of two different, key requirements in the modelling and analysis of data-aware processes. On the one hand, we support the creation of fresh (case) objects during the execution of the process, and the ability to model their (co-)evolution using guards and updates. Examples of such objects are orders and their orderlines in an order-to-cash process. On the other hand, we handle read-only, persistent data that can be accessed and injected in the objects manipulated by the process. Examples of read-only data are the catalog of product types and the list of customers in an order-to-cash process. Importantly, read-only data have to be considered in a *parameterised* way. This means that the overall process is expected to operate as desired in a robust way, irrespectively of the actual configuration of such data.

While the first requirement is commonly tackled by the most recent and sophisticated approaches for integrating data within Petri nets [14,22,24], the latter has been extensively investigated in the data-centric spectrum [9,12], but only recently ported to more conventional, imperative processes with the simplifying assumptions that the process control-flow is block-structured (and thus 1-bounded in the Petri net sense) [7].

In this work, we reconcile these two themes in an extension of coloured Petri nets (CPNs) called *catalog-nets* (CLog-nets). On the one hand, in CLog-net transitions are equipped with guards that simultaneously inspect the content of tokens and query facts stored in a read-only, persistent database. On the other hand, such transitions can inject data into tokens by extracting relevant values from the database or by generating genuinely fresh ones. We systematically encode CLog-nets into the most recent version of MCMT [1,16], one of the few model checkers natively supporting the (parameterised) verification of data and processes [6,8,9]. We show that fresh-value injection is a particularly complex feature to handle, and discuss strategies to tame it. We then stress that, thanks to this encoding, a relevant fragment of the model can be readily verified using MCMT, and that verification of the whole model is within reach with a minor implementation effort. Finally, we discuss how catalog nets provide a unifying approach for some of the most sophisticated formalisms in this area, highlighting differences and commonalities.

2   The CLog-net Formal Model

Conceptually, a CLog-net integrates two key components. The first is a read-only persistent data storage, called *catalog*, to account for read-only, parameterised

data. The second is a variant of CPN, called ν-CPN [23], to model the process backbone. Places carry tuples of data objects and can be used to represent: *(i)* states of (interrelated) case objects, *(ii)* read-write relations, *(iii)* read-only relations whose extension is fixed (and consequently not subject to parameterisation), *(iv)* resources. As in [14, 23, 24], the net employs ν-variables (first studied in the context of ν-PNs [26]) to inject fresh data (such as object identifiers). A distinguishing feature of CLog-nets is that transitions can have guards that inspect and retrieve data objects from the read-only, catalog.

Catalog. We consider a *type domain* $\mathfrak{D}$ as a finite set of pairwise disjoint data types accounting for the different types of objects in the domain. Each type $\mathcal{D} \in \mathfrak{D}$ comes with its own (possibly infinite) *value domain* $\Delta_{\mathcal{D}}$, and with an equality operator $=_{\mathcal{D}}$. When clear from the context, we simplify the notation and use $=$ in place of $=_{\mathcal{D}}$. $R(a_1 : \mathcal{D}_1, \ldots, a_n : \mathcal{D}_n)$ is a $\mathfrak{D}$-*typed relation schema*, where R is a relation name and $a_i : \mathcal{D}_i$ indicates the i-th attribute of R together with its data type. When no ambiguity arises, we omit relation attributes and/or their data types. A $\mathfrak{D}$-*typed catalog (schema)* $\mathcal{R}_{\mathfrak{D}}$ is a finite set of $\mathfrak{D}$-typed relation schemas. A $\mathfrak{D}$-*typed catalog instance Cat* over $\mathcal{R}_{\mathfrak{D}}$ is a finite set of facts $R(\mathsf{o_1}, \ldots, \mathsf{o_n})$, where $R \in \mathcal{R}_{\mathfrak{D}}$ and $\mathsf{o_i} \in \Delta_{\mathcal{D}_i}$, for $i \in \{1, \ldots, n\}$.

We adopt two types of *constraints* in the catalog relations. First, we assume the first attribute of every relation $R \in \mathcal{R}_{\mathfrak{D}}$ to be its *primary key*, denoted as $\mathrm{PK}(R)$. Also, a type of such attribute should be different from the types of other primary key attributes. Then, for any $R, S \in \mathcal{R}_{\mathfrak{D}}$, $R.a \to S.id$ defines that the projection $R.a$ is a *foreign key* referencing $S.id$, where $\mathrm{PK}(S) = id$, $\mathrm{PK}(R) \neq a$ and $\mathtt{type}(id) = \mathtt{type}(a)$. While the given setting with constraints may seem a bit restrictive, it is the one adopted in the most sophisticated settings where parameterisation of read-only data is tackled [9, 12].

Example 1. Consider a simple catalog of an order-to-delivery scenario, containing two relation schemas. Relation schema $ProdCat(p : \mathtt{ProdType})$ indicates the product types (e.g., vegetables, furniture) available in the organisation catalogue of products. Relation schema $Comp(c : \mathtt{CId}, p : \mathtt{ProdType}, t : \mathtt{TruckType})$ captures the compatibility between products and truck types used to deliver orders; e.g. one may specify that vegetables are compatible only with types of trucks that have a refrigerator. $\triangleleft$

Catalog Queries. We fix a countably infinite set $\mathcal{V}_{\mathfrak{D}}$ of typed variables with a *variable typing function* $\mathtt{type} : \mathcal{V}_{\mathfrak{D}} \to \mathfrak{D}$. Such function can be easily extended to account for sets, tuples and multisets of variables as well as constants. As query language we opt for the union of conjunctive queries with inequalities and atomic negations that can be specified in terms of first-order (FO) logic extended with types. This corresponds to widely investigated SQL select-project-join queries with filters, and unions thereof.

A *conjunctive query (CQ) with atomic negation* Q over $\mathcal{R}_{\mathfrak{D}}$ has the form

$$Q ::= \varphi \mid R(x_1, \ldots, x_n) \mid \neg R(x_1, \ldots, x_n) \mid Q_1 \wedge Q_2 \mid \exists x.Q,$$

where *(i)* $R(\mathcal{D}_1, \ldots, \mathcal{D}_n) \in \mathcal{R}_{\mathfrak{D}}$, $x \in \mathcal{V}_{\mathfrak{D}}$ and each x_i is either a variable of type $\mathcal{D}_i$ or a constant from $\Delta_{\mathcal{D}_i}$; *(ii)* $\varphi ::= y_1 = y_2 \,|\, \neg\varphi \,|\, \varphi \wedge \varphi \,|\, \top$ is a *condition*, s.t. y_i is either a variable of type $\mathcal{D}$ or a constant from $\Delta_{\mathcal{D}}$. $\mathtt{CQ}_{\mathfrak{D}}^-$ denotes the set of all such conjunctive queries, and *Free(Q)* the set of all free variables (i.e., those not occurring in the scope of quantifiers) of query Q. $\mathcal{C}_{\mathfrak{D}}$ denotes the set of all possible conditions, *Vars(Q)* the set of all variables in Q, and *Const(Q)* the set of all constants in Q. Finally, $\mathtt{UCQ}_{\mathfrak{D}}^-$ denotes the set off all *unions of conjunctive queries* over $\mathcal{R}_{\mathfrak{D}}$. Each query $Q \in \mathtt{UCQ}_{\mathfrak{D}}^-$ has the form $Q = \bigwedge_{i=1}^n Q_i$, with $Q_i \in \mathtt{CQ}_{\mathfrak{D}}^-$.

A *substitution* for a set $X = \{x_1, \ldots, x_n\}$ of typed variables is a function $\theta : X \to \Delta_{\mathfrak{D}}$, such that $\theta(x) \in \Delta_{\mathtt{type}(x)}$ for every $x \in X$. An empty substitution is denoted as $\langle\rangle$. A *substitution* θ *for a query Q*, denoted as $Q\theta$, is a substitution for variables in *Free(Q)*. An *answer to a query Q* in a catalog instance *Cat* is a set of substitutions $ans(Q, Cat) = \{\theta : Free(Q) \to Val(Cat) \mid Cat, \models \theta\, Q\}$, where *Val(Cat)* denotes the set of all constants occurring in *Cat* and $\models$ denotes standard FO entailment.

Example 2. Consider the catalog of Example 1. Query *ProdCat(p)* retrieves the product types p present in the catalog, whereas given a product type value veg, query $\exists c.\, Comp(c, \mathsf{veg}, t)$ returns the truck types t compatible with veg.   ◁

CLog-nets. We first fix some standard notions related to *multisets*. Given a set A, the *set of multisets* over A, written $A^{\oplus}$, is the set of mappings of the form $m : A \to \mathbb{N}$. Given a multiset $S \in A^{\oplus}$ and an element $a \in A$, $S(a) \in \mathbb{N}$ denotes the number of times a appears in S. We write $a^n \in S$ if $S(a) = n$. The support of S is the set of elements that appear in S at least once: $supp(S) = \{a \in A \mid S(a) > 0\}$. We also consider the usual operations on multisets. Given $S_1, S_2 \in A^{\oplus}$: *(i)* $S_1 \subseteq S_2$ (resp., $S_1 \subset S_2$) if $S_1(a) \leq S_2(a)$ (resp., $S_1(a) < S_2(a)$) for each $a \in A$; *(ii)* $S_1 + S_2 = \{a^n \mid a \in A \text{ and } n = S_1(a) + S_2(a)\}$; *(iii)* if $S_1 \subseteq S_2$, $S_2 - S_1 = \{a^n \mid a \in A \text{ and } n = S_2(a) - S_1(a)\}$; *(iv)* given a number $k \in \mathbb{N}$, $k \cdot S_1 = \{a^{kn} \mid a^n \in S_1\}$; *(v)* $|m| = \sum_{a \in A} m(a)$. A multiset over A is called empty (denoted as $\emptyset^{\oplus}$) iff $\emptyset^{\oplus}(a) = 0$ for every $a \in A$.

We now define CLog-nets, extending ν-CPNs [23] with the ability of querying a read-only catalog. As in CPNs, each CLog-net place has a color type, which corresponds to a data type or to the cartesian product of multiple data types from $\mathfrak{D}$. Tokens in places are referenced via *inscriptions* – tuples of variables and constants. We denote by Ω_A the set of all possible inscriptions over a set A and, with slight abuse of notation, use *Vars(ω)* (resp., *Const(ω)*) to denote the set of variables (resp., constants) of $\omega \in \Omega_A$. To account for fresh external inputs, we employ the well-known mechanism of ν-Petri nets [26] and introduce a countably infinite set $\Upsilon_{\mathfrak{D}}$ of $\mathfrak{D}$-typed *fresh variables*, where for every $\nu \in \Upsilon_{\mathfrak{D}}$, we have that $\Delta_{\mathtt{type}(\nu)}$ is countably infinite (this provides an unlimited supply of fresh values). We fix a countably infinite set of $\mathfrak{D}$-typed variable $\mathcal{X}_{\mathfrak{D}} = \mathcal{V}_{\mathfrak{D}} \uplus \Upsilon_{\mathfrak{D}}$ as the disjoint union of "normal" ($\mathcal{V}_{\mathfrak{D}}$) and fresh ($\Upsilon_{\mathfrak{D}}$) variables.

Definition 1. *A $\mathfrak{D}$-typed CLog-net $\mathcal{N}$ over a catalog schema $\mathcal{R}_{\mathfrak{D}}$ is a tuple* $(\mathfrak{D}, \mathcal{R}_{\mathfrak{D}}, P, T, F_{in}, F_{out}, \mathtt{color}, \mathtt{guard})$, *where:*

1. P and T are finite sets of places and transitions, s.t. $P \cap T = \emptyset$;
2. $\texttt{color} : P \to \wp(\mathfrak{D})$ is a place typing function;
3. $F_{in} : P \times T \to \Omega^{\oplus}_{\mathcal{V}_{\mathfrak{D}}}$ is an input flow, s.t. $\texttt{type}(F_{in}(p,t)) = \texttt{color}(p)$ for every $(p,t) \in P \times T$;
4. $F_{out} : T \times P \to \Omega^{\oplus}_{\mathcal{X}_{\mathfrak{D}} \cup \Delta_{\mathfrak{D}}}$ is an output flow, s.t. $\texttt{type}(F_{out}(t,p)) = \texttt{color}(p)$ for every $(t,p) \in T \times P$;
5. $\texttt{guard} : T \to \{Q \wedge \varphi \mid Q \in \texttt{UCQ}^{-}_{\mathfrak{D}}, \varphi \in \mathcal{C}_{\mathfrak{D}}\}$ is a partial guard assignment function, s.t., for every $\texttt{guard}(t) = Q \wedge \varphi$ and $t \in T$, the following holds:
   (a) $Vars(\varphi) \subseteq InVars(t)$, where $InVars(t) = \cup_{p \in P} Vars(F_{in}(p,t))$;
   (b) $OutVars(t) \setminus (InVars(t) \cup \Upsilon_{\mathfrak{D}}) \subseteq Free(Q)$ and $Free(Q) \cap Vars(t) = \emptyset$, where $OutVars(t) = \cup_{p \in P} Vars(F_{out}(t,p))$ and $Vars(t) = InVars(t) \cup OutVars(t)$. ◁

Here, the role of guards is twofold. On the one hand, similarly, for example, to CPNs, guards are used to impose *conditions* (using φ) on tokens flowing through the net. On the other hand, a guard of transition t may also *query* (using Q) the catalog in order to propagate some data into the net. The acquired data may be still filtered by using $InVars(t)$. Note that in condition *(b)* of the guard definition we specify that there are some variables (excluding the fresh ones) in the outgoing arc inscriptions that do not appear in $InVars(t)$ and that are used by Q to insert data from the catalog. Moreover, it is required that all free variables of Q must coincide with the variables of inscriptions on outgoing and incoming arcs of a transition it is assigned to. In what follows, we shall define arc inscriptions as $k \cdot \omega$, where $k \in \mathbb{N}$ and $\omega \in \Omega_A$ (for some set A).

Semantics. The execution semantics of a CLog-net is similar to the one of CPNs. Thus, as a first step we introduce the standard notion of net marking. Formally, a *marking* of a CLog-net $N = (\mathfrak{D}, \mathcal{R}_{\mathfrak{D}}, P, T, F_{in}, F_{out}, \texttt{color}, \texttt{guard})$ is a function $m : P \to \Omega^{\oplus}_{\mathfrak{D}}$, so that $m(p) \in \Delta^{\oplus}_{\texttt{color}(p)}$ for every $p \in P$. We write $\langle N, m, Cat \rangle$ to denote CLog-net N marked with m, and equipped with a read-only catalog instance Cat over $\mathcal{R}_{\mathfrak{D}}$.

The firing of a transition t in a marking is defined w.r.t. a so-called *binding* for t defined as $\sigma : Vars(t) \to \Delta_{\mathfrak{D}}$. Note that, when applied to (multisets of) tuples, σ is applied to every variable singularly. For example, given $\sigma = \{x \mapsto 1, y \mapsto \texttt{a}\}$, its application to a multiset of tuples $\omega = \{\langle x, y \rangle^2, \langle x, \texttt{b} \rangle\}$ results in $\sigma(\omega) = \{\langle 1, \texttt{a} \rangle^2, \langle x, \texttt{b} \rangle\}$.

Definition 2. *A transition $t \in T$ is enabled in a marking m and a fixed catalog instance Cat, written $m[t\rangle_{Cat}$, if there exists binding σ satisfying the following: (i) $\sigma(F_{in}(p,t)) \subseteq m(p)$, for every $p \in P$; $\sigma(\texttt{guard}(t))$ is true; (ii) $\sigma(x) \notin Val(m) \cup Val(Cat)$, for every $x \in \Upsilon_{\mathfrak{D}} \cap OutVars(t)$;[1] (iii) $\sigma(x) \in ans(Q, Cat)$ for $x \in OutVars(t) \setminus (\Upsilon_{\mathfrak{D}} \cup InVars(t)) \cap Vars(Q)$ and query Q from $\texttt{guard}(t)$.* ◁

---

[1] Here, with slight abuse of notation, we define by $Val(m)$ the set of all values appearing in m.

Essentially, a transition is enabled with a binding σ if the binding selects data objects carried by tokens from the input places and the read-only catalog instance, so that the data they carry make the guard attached to the transition true.

When a transition t is enabled, it may fire. Next we define what are the effects of firing a transition with some binding σ.

Definition 3. *Let $\langle N, m, Cat \rangle$ be a marked CLog-net, and $t \in T$ a transition enabled in m and Cat with some binding σ. Then, t may* fire *producing a new marking m', with $m'(p) = m(p) - \sigma(F_{in}(p, t)) + \sigma(F_{out}(t, p))$ for every $p \in P$. We denote this as $m[t\rangle_{Cat} m'$ and assume that the definition is inductively extended to sequences $\tau \in T^*$.* ◁

For $\langle N, m_0, Cat \rangle$ we use $\mathcal{M}(N) = \{m \mid \exists \tau \in T. m_0[\tau\rangle_{Cat} m\}$ to denote the set of all markings of N reachable from its initial marking m_0.

We close with an example that illustrates all the main features of CLog-nets.

Given $b \in \mathbb{N}$, a marked CLog-net $\langle N, m_0, Cat \rangle$ is called *bounded with bound b* if $|m(p)| \leq b$, for every marking $m \in \mathcal{M}(N)$ and every place $p \in P_c$. Unboundedness in CLog-nets can arise due to various reasons: classical unbounded generation of tokens, but also uncontrollable emission of fresh values with ν-variables or replication of data values from the catalog via queries in transition guards. Notice that Definition 3 does not involve the catalog, which is in fact fixed throughout the execution.

Execution Semantics. The execution semantics of a marked CLog-net $\langle N, m_0, Cat \rangle$ is defined in terms of a possibly infinite-state transition system in which states are labeled by reachable markings and each arc (or transition) corresponds to the firing of a transition in N with a given binding. The transition system captures all possible executions of the net, by interpreting concurrency as interleaving. Due to space limitations, the formal definition of how this transition system is induced can be found in [15].

As pointed out before, we are interested in analysing a CLog-net irrespectively of the actual content of the catalog. Hence, in the following when we mention a (catalog-parameterised) marked net $\langle N, m_0 \rangle$ without specifying how the catalog is instantiated, we actually implicitly mean the *infinite set of marked nets $\langle N, m_0, Cat \rangle$* for every possible instance Cat defined over the catalog schema of N.

Example 3. Starting from the catalog in Example 1, Fig. 1 shows a simple, yet sophisticated example of CLog-net capturing the following order-to-delivery process. Orders can be created by executing the new order transition, which uses a ν-variable to generate a fresh order identifier. A so-created, working order can be populated with items, whose type is selected from those available in the catalog relation *ProdCat*. Each item then carries its product type and owning order. When an order contains at least one item, it can be paid. Items added to an order can be removed or loaded in a compatible truck. The set of available trucks, indicating their plate numbers and types, is contained in a dedicated *pool*
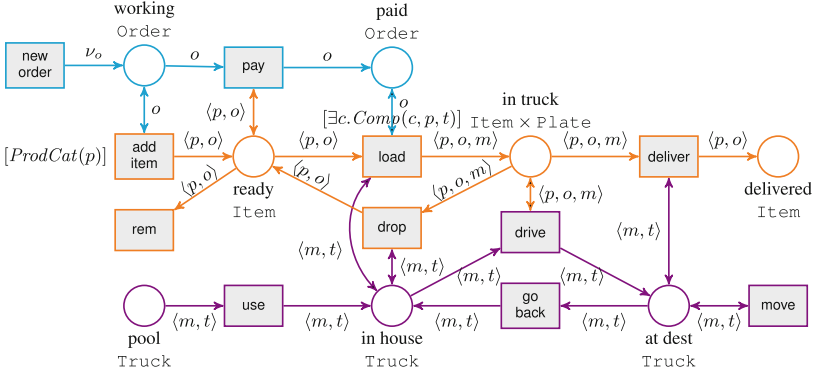
Fig. 1. A CLog-net (its catalog is in Example 1). In the picture, `Item` and `Truck` are compact representations for `ProdType × Order` and `Plate × TruckType` respectively. The top blue part refers to orders, the central orange part to items, and the bottom violet part to delivery trucks.

place. Trucks can be borrowed from the pool and placed in house. An item can be loaded into a truck if its owning order has been paid, the truck is in house, and the truck type and product type of the item are compatible according to the *Comp* relation in the catalog. Items (possibly from different orders) can be loaded in a truck, and while the truck is in house, they can be dropped, which makes them ready to be loaded again. A truck can be driven for delivery if it contains at least one loaded item. Once the truck is at its destination, some items may be delivered (this is simply modelled non-deterministically). The truck can then either move, or go back in house. ◁

Example 3 shows various key aspects related to modelling data-aware processes with multiple case objects using CLog-nets. First of all, whenever an object is involved in a many-to-one relation from the "many" side, it then becomes responsible of carrying the object to which it is related. This can be clearly seen in the example, where each item carries a reference to its owning order and, once loaded into a truck, a reference to the truck plate number. Secondly, the three object types involved in the example show three different modelling patterns for their creation. Unboundedly many orders can be genuinely created using a ν-variable to generate their (fresh) identifiers. The (finite) set of trucks available in the domain is instead fixed in the initial marking, by populating the *pool* place. The CLog-net shows that such trucks are used as resources that can change state but are never destroyed nor created. Finally, the case of items is particularly interesting. Items in fact can be arbitrarily created and destroyed. However, their creation is not modelled using an explicit ν-variable, but is instead simply obtained by the add item transition with the usual token-creation mechanism, in which the product type is taken from the catalog using the query assigned to add item. Thanks to the multiset semantics of Petri nets, it is still possible to create multiple items having the same product type and owning order. However, it is

not possible to track the evolution of a specific item, since there is no explicit identifier carried by item tokens. This is not a limitation in this example, since items are not referenced by other objects present in the net (which is instead the case for orders and trucks). All in all, this shows that ν-variables are only necessary when the CLog-net needs to handle the arbitrary creation of objects that are referenced by other objects.

3   From CLog-nets to MCMT

We now report on the encoding of CLog-nets into the verification language supported by the MCMT model checker, showing that the various modelling constructs of CLog-nets have a direct counterpart in MCMT, and in turn enabling formal analysis.

MCMT is founded on the theory of *array-based systems* [1,16], an umbrella term used to refer to *infinite-state transition systems* specified using a declarative, logic-based formalism by which arrays are manipulated via logical updates. An array-based system is represented using a multi-sorted theory with two kinds of sorts: one for the indexes of arrays, and the other for the elements stored therein. Since the content of an array changes over time, it is referred to by a function variable, whose interpretation in a state is that of a total function mapping indexes to elements (applying the function to an index denotes the classical *read* array operation). We adopt here the module of MCMT called "database-driven mode", which supports the representation of read-only databases.

Specifically, we show how to encode a CLog-net $\langle N, m_0 \rangle$, where $N = (\mathfrak{D}, \mathcal{R}_{\mathfrak{D}}, P, T, F_{in}, F_{out}, \texttt{color}, \texttt{guard})$ into (data-driven) MCMT specification. The translation is split into two phases. First, we tackle the type domain and catalog. Then, we present a step-wise encoding of the CLog-net places and transitions into arrays.

Data and Schema Translation. We start by describing how to translate static data-related components. Let $\mathfrak{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_{n_d}\}$. Each data type $\mathcal{D}_i$ is encoded in MCMT with declaration :smt (define-type Di). For each declared type $\mathcal{D}$ MCMT implicitly generates a special NULL constant indicating an empty/undefined value of $\mathcal{D}$.

To represent the catalog relations of $\mathcal{R}_{\mathfrak{D}} = \{R_1, \ldots, R_{n_r}\}$ in MCMT, we proceed as follows. Recall that in catalog every relation schema has $n + 1$ typed attributes among which some may be foreign keys referencing other relations, its first attribute is a primary key, and, finally, primary keys of different relation schemas have different types. With these conditions at hand, we adopt the functional characterisation of read-only databases studied in [9]. For every relation $R_i(id, A_1, \ldots, A_n)$ with $\text{PK}(R) = \{id\}$, we introduce unary functions that correctly reference each attribute of R_i using its primary key. More specifically, for every A_j $(j = 1, \ldots, n)$ we create a function $f_{R_i, A_j} : \Delta_{\texttt{type}(id)} \to \Delta_{\texttt{type}A_j}$. If A_j is referencing an identifier of some other relation S (i.e., $R_i.A_j \to S.id$), then f_{R_i, A_j} represents the foreign key referencing to S. Note that in this case the types of A_j and $S.id$ should coincide. In MCMT, assuming that $\texttt{D\_Ri.id} = \texttt{type}(id)$

and $D\_Aj = \texttt{type}(A_j)$, this is captured using statement `:smt (define Ri_Aj ::(-> D_Ri.id D_Aj))`.

All the constants appearing in the net specification must be properly defined. Let $C = \{v_1, \ldots, v_{n_c}\}$ be the set of all constants appearing in N. C is defined as $\bigcup_{t \in T} Const(\texttt{guard}(t)) \cup supp(m_0) \cup \bigcup_{t \in T, p \in P} Const(F_{out}(t, p))$. Then, every constant $v_i \in C$ of type $\mathcal{D}$ is declared in MCMT as `:smt (define vi ::D)`.

The code section needed to make MCMT aware of the fact that these elements have been declared to describe a read-only database schema is depicted in Listing 1.1 (notice that the last declaration is required when using MCMT in the database-driven mode).

Listing 1.1.

```
:db_driven
:db_sorts D1,...,Dnd
:db_functions
    R1_A1,...,Rnr_Ak
:db_constants v1,...,vnc
:db_relations   //leave empty
```

Listing 1.2.

```
:initial
:var x
:cnj init_p1
    ...
    init_pn
```

Places. Given that, during the net execution, every place may store unboundedly many tokens, we need to ensure a potentially infinite provision of values to places p using unbounded arrays. To this end, every place $p \in P$ with $\texttt{color}(p) = \mathcal{D}_1 \times \ldots \times \mathcal{D}_k$ is going to be represented as a combination of arrays $p_1, \ldots, p_k$, where a special index type $\mathbf{P_{ind}}$ (disjoint from all other types) with domain $\Delta_{P_{ind}}$ is used as the array index sort and $\mathcal{D}_1, \ldots, \mathcal{D}_k$ account for the respective target sorts of the arrays.[2] In MCMT, this is declared as `:local p_1 D1 ... :local p_k Dk`. Then, intuitively, we associate to the j-th token $(v_1, \ldots, v_k) \in m(p)$ an element $j \in \Delta_{P_{ind}}$ and a tuple $(j, p_1[j], \ldots, p_k[j])$, where $p_1[j] = v_1, \ldots, p_k[j] = v_k$. Here, j is an *"implicit identifier"* of this tuple in $m(p)$. Using this intuition and assuming that there are in total n control places, we represent the initial marking m_0 in two steps (a direct declaration is not possible due to the language restrictions of MCMT). First, we symbolically declare that all places are by default empty using the MCMT initialisation statement from Listing 1.2. There, `cnj` represents a conjunction of atomic equations that, for ease of reading, we organized in blocks, where each $init\_p_i$ specifies for place $p_i \in P$ with $\texttt{color}(p_i) = \mathcal{D}_1 \times \ldots \times \mathcal{D}_k$ that it contains no tokens. This is done by explicitly "nullifying" all component of each possible token in p_i, written in MCMT as `(= pi_1[x] NULL_D1)(= pi_2[x] NULL_D2)...(= pi_k[x] NULL_DK)`. The initial marking is then injected with a dedicated MCMT code that populates the place arrays, initialised as empty, with records representing tokens therein. Due to the lack of space, this code is provided in [15].

Transition Enablement and Firing. We now show how to check for transition enablement and compute the effect of a transition firing in MCMT. To this

---

[2] MCMT has only one index sort, but, as shown in [15], there is no loss of generality in doing that.
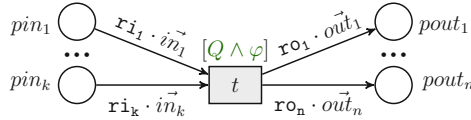
Fig. 2. A generic CLog-net transition ($\mathtt{ri_j}$ and $\mathtt{ro_j}$ are natural numbers)

end, we consider the generic, prototypical CLog-net transition $t \in T$ depicted in Fig. 2. The enablement of this transition is subject to the following conditions: *(FC1)* there is a binding σ that correctly matches tokens in the places to the corresponding inscriptions on the input arcs (i.e., each place pin_i provides enough tokens required by a corresponding inscription $F(pin_i, t) = \vec{in_i}$), and that computes new and possibly *fresh* values that are pairwise distinct from each other as well as from all other values in the marking; *(FC2)* the guard $\mathtt{guard}(t)$ is satisfied under the selected binding. In MCMT, t is captured with a transition statement consisting of a guard G and an update U as in Listing 1.3.

<table>
<tr><td>

Listing 1.3.
```
: transition
: var  x , x1 ,..., xK , y1 ,..., yN
: var  j
: guard  G
...  U  ...
```

</td><td>

Listing 1.4.
```
: numcases  NC
...
: case  (= j  i)
: val  v₁,ᵢ
...
: val  vₖ,ᵢ
...
```

</td></tr>
</table>

Here every $\mathtt{x}$ (resp., $\mathtt{y}$) represents an existentially quantified index variable corresponding to variables in the incoming inscriptions (resp., outgoing inscriptions), $\mathtt{K} = \sum_{j \in \{1,\ldots,k\}} \mathtt{ri}_j$, $\mathtt{N} = \sum_{j \in \{1,\ldots,n\}} \mathtt{ro}_j$ and $\mathtt{j}$ is a universally quantified variable, that will be used for computing bindings of ν-variables and updates. In the following we are going to elaborate on the construction of the MCMT transition statement. We start by discussing the structure of G which in MCMT is represented as a conjunction of atoms or negated atoms and, intuitively, addresses all the conditions stated above.

First, to construct a binding that meets condition *(FC1)*, we need to make sure that every place contains enough of tokens that match a corresponding arc inscription. Using the array-based representation, for every place pin_i with $F_{in}(pin_i, t) = \mathtt{ri}_i \cdot \vec{in_i}$ and $|\mathtt{color}(pin_i)| = k$, we can check this with a formula

$$\psi_{pin_i} := \exists \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{ri}_i}. \bigwedge_{\substack{j_1, j_2 \in \{\mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{ri}_j}\}, j_1 \neq j_2, \\ l \in \{1, \ldots, k\}}} pin_{i,l}[j_1] = pin_{i,l}[j_2] \wedge \bigwedge_{l \in \{1, \ldots, k\}} pin_{i,l}[\mathtt{x}_1] \neq \mathtt{NULL\_D}_l$$

Given that variables representing existentially quantified index variables are already defined, in MCMT this is encoded as conjunctions of atoms (= pini_l[j_1] pini_l[j_2]) and atoms not(= pini_l[x1] NULL_Dl), where NULL_Dl is a special null constant of type of elements stored in pini_l. All such conjunctions, for all input places of t, should be appended to G.

We now define the condition that selects proper indexes in the output places so as to fill them with the tokens generated upon transition firing. To this end, we need to make sure that all the q declared arrays a_w of the system[3] (including the arrays $pout_i$ corresponding to the output places of t) contain no values in the slots marked by y index variables. This is represented using a formula

$$\psi_{pout_i} := \exists \mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{ri}_i} . \bigwedge_{j \in \{\mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{ri}_j}\}, w \in \{1, \ldots q\}} a_w[j] = \mathtt{NULL\_D}_w,$$

which is encoded in MCMT similarly to the case of ψ_{pin_i}.

Moreover, when constructing a binding, we have to take into account the case of arc inscriptions causing implicit "joins" between the net marking and data retrieved from the catalog. This happens when there are some variables in the input flow that coincide with variables of Q, i.e., $Vars(F_{in}(pin_j, t)) \cap Vars(Q) \neq \emptyset$. For ease of presentation, denote the set of such variables as $\mathbf{s} = \{s_1, \ldots, s_r\}$ and introduce a function π that returns the position of a variable in a tuple or relation. E.g., $\pi(\langle x, y, z \rangle, y) = 2$, and $\pi(R, B) = 3$ in $R(id, A, B, E)$. Then, for every relation R in Q we generate a formula

$$\psi_R := \bigwedge_{j \in \{1, \ldots, k\}, s \in (\mathbf{s} \cap Vars(R))} pin_{j, \pi(\vec{in_j}, s)}[\mathtt{x}] = f_{R, A_{\pi(R,s)}}(id)$$

This formula guarantees that values provided by a constructed binding respect the aforementioned case for some index x (that has to coincide with one of the index variables from ψ_{pin_j}) and identifier id. In MCMT this is encoded as a conjunction of atoms (= (R_Ai id) pinj_l[x]), where i = $\pi(R, s)$ and l = $\pi(\vec{in_j}, s)$. As in the previous case, all such formulas are appended to G.

We now incorporate the encoding of condition *(FC2)*. Every variable z of Q with type(z) = D has to be declared in MCMT as :eevar z D. We call an *extended guard* a guard $Q^e \wedge \varphi^e$ in which every relation R has been substituted with its functional counterpart and every variable z in φ has been substituted with a "reference" to a corresponding array pin_j that z uses as a value provider for its bindings. More specifically, every relation $R/n + 1$ that appears in Q as $R(id, z_1, \ldots, z_n)$ is be replaced by conjunction $id \neq \mathtt{NULL\_D} \wedge f_{R, A_1}(id) = z_1 \wedge \ldots \wedge f_{R, A_n}(id) = z_n$, where D = type($id$). In MCMT, this is written as (not (= id NULL_D)) $expr_1$ ... $expr_n$ (id should be declared using :eevar as well). Here, every $expr_i$ corresponds to an atomic equality from above and is specified in MCMT in three different ways based on the nature of z_i. Let us assume that z_i has

---

[3] This is a technicality of MCMT, as explained in [15], since MCMT has only one index sort.

been declared before as `:eevar z_i D`. If z_i appears in a corresponding incoming transition inscription, then $expr_i$ is defined as `(= (R_Ai id) pin_j[x])` `(= z_i pin_j[x])`, where `i`-th attribute of R coincides with the `j`-th variable in the inscription $F_{in}(pin, t)$. If z_i is a variable bound by an existential quantifier in Q, then $expr_i$ in MCMT is going to look as `(= (R_Ai id) zi)`. Finally, if z_i is a variable in an outgoing inscription used for propagating data from the catalog (as discussed in condition *(1)*), then $expr_i$ is simply defined with the following statement: `(= (R_Ai id) z_i)`, where `Di` is the type of z_i.

Variables in φ are substituted with their array counterparts. In particular, every variable $z \in Vars(\varphi)$ is substituted with `pinj_i[x]`, where $i = \pi(\vec{in_j}, z)$. Given that φ is represented as a conjunction of variables, its representation in MCMT together with the aforementioned substitution is trivial. To finish the construction of G, we append to it the MCMT version of $Q^e \wedge \varphi^e$.

We come back to condition *(FC1)* and show how bindings are generated for ν-variables of the output flow of t. In MCMT we use a special universal guard `:uguard` (to be inserted right after the `:guard` entry) that, for every variable $\nu \in \Upsilon_{\mathfrak{D}} \cap (OutVars(t) \setminus Vars(\vec{out_j}))$ previously declared using `:eevar nu D`, and for arrays $p_1, \ldots, p_k$ with target sort D, consists of expression `(not(=nu p_1[j]))...(not(=nu p_k[j]))` for all p. This encodes "local" freshness for ν-variables, which suffice for our goal.

After a binding has been generated and the guard of t has been checked, a new marking is generated by assigning corresponding tokens to the outgoing places and by removing tokens from the incoming ones. Note that, while the tokens are populated by assigning their values to respective arrays, the token deletion happens by nullifying (i.e., assigning special `NULL` constants) entries in the arrays of the input places. All these operations are specified in the special update part of the transition statement U and are captured in MCMT as depicted in Listing 1.4. There, the transition runs through NC cases. All the following cases go over the indexes `y1,...,yN` that correspond to tokens that have to be added to places. More specifically, for every place $pout \in P$ such that $|\texttt{color}(pout)| = k$, we add an i-th token to it by putting a value $v_{r,i}$ in i-th place of every r-th component array of *pout*. This $v_{r,i}$ can either be a ν-variable `nu` from the universal guard, or a value coming from a place *pin* specified as `pin[xm]` (from some `x` input index variable) or a value from some of the relations specified as `(R_Ai id)`. Note that `id` should be also declared as `:eevar id D_Ri.id`, where $type(id) = \texttt{D\_Ri.id}$. Every `:val v` statement follows the order in which all the local and global variables have been defined, and, for array variables a and every every case `(= j i)`, such statement stands for a simple assignment $a[i] := \texttt{v}$.

Implementation Status. The provided translation is fully compliant with the concrete specification language MCMT. The current implementation has however a limitation on the number of supported index variables in each MCMT transition statement. Specifically, two existentially quantified and one universally quantified variables are currently supported. This has to be taken into account if one wants to run the model checker on the result produced by translating a CLognet, and possibly requires to rewrite the net (if possible) into one that does not

exceed the supported number of index variables. What can be actually rewritten (and how) is beyond the scope of this paper.

In addition, notice that this limitation is not dictated by algorithmic nor theoretical limitations, but is a mere characteristic of the current implementation, and comes from the fact that the wide range of systems verified so far with MCMT never required to simultaneously quantify on many array indexes. There is an ongoing implementation effort for a new version of MCMT that supports arbitrarily many quantified index variables, and consequently concrete model checking of the full CLog-net model is within reach. Currently, we do not have a software prototype that encodes the translation, but this section indicates exactly how this should be implemented.

4   Parameterised Verification

Thanks to the encoding of CLog-nets into (the database-driven module of) MCMT, we can handle the parameterised verification of safety properties over CLog-nets, and study crucial properties such as soundness, completeness, and termination by relating CLog-nets with the foundational framework underlying such an MCMT module [8,9].

This amounts to verifying whether it is true that *all* the reachable states of a marked CLog-net satisfy a desired condition, *independently from the content of the catalog*. As customary in this setting, this form of verification is tackled in a converse way, by formulating an *unsafe condition*, and by checking whether there exists an instance of the catalog such that the CLog-net can evolve the initial marking to a state where the unsafe condition holds. Technically, given a property ψ capturing an unsafe condition and a marked CLog-net $\langle N, m_0 \rangle$, we say that $\langle N, m_0 \rangle$ is *unsafe* w.r.t. ψ if there exists a catalog instance Cat for N such that the marked CLog-net with fixed catalog $\langle N, m_0, Cat \rangle$ can reach a configuration where ψ holds.

With a slight abuse of notation, we interchangeably use the term CLog-net to denote the input net or its MCMT encoding. We start by defining (unsafety) properties, in a way that again guarantees a direct encoding into the MCMT model checker. For space limitations, we refer to the translation of properties over CLog-nets in [15].

Definition 4. *A* property *over CLog-net N is a formula of the form $\exists \vec{y}.\psi(\vec{y})$, where $\psi(\vec{y})$ is a quantifier-free query that additionally contains atomic predicates $[p \geq c]$ and $[p(x_1, \ldots, x_n) \geq c]$, where p is a place name from N, $c \in \mathbb{N}$, and $Vars(\psi) = Y_P$, with Y_P being the set of variables appearing in the atomic predicates $[p(x_1, \ldots, x_n) \geq c]$.* ◁

Here, $[p \geq c]$ specifies that in place p there are at least c tokens. Similarly, $[p(x_1, \ldots, x_n) \geq c]$ indicates that in place p there are at least c tokens carrying the tuple $\langle x_1, \ldots, x_n \rangle$ of data objects. A property may also mention relations from the catalog, provided that all variables used therein also appear in atoms that inspect places.

This can be seen as a language to express *data-aware coverability properties* of a CLog-net, possibly relating tokens with the content of the catalog. Focusing on covered markings as opposed as fully-specified reachable markings is customary in data-aware Petri nets or, more in general, well-structured transition systems (such as ν-PNs [26]).

Example 4. Consider the CLog-net of Example 3, with an initial marking that populates the *pool* place with available trucks. Property $\exists p, o.[delivered(p, o) \geq 1] \wedge [working(o) \geq 1]$ captures the undesired situation where a delivery occurs for an item that belongs to a working (i.e., not yet paid) order. This can never happen, irrespectively of the content of the net catalog: items can be delivered only if they have been loaded in a compatible truck, which is possible only if the order of the loaded item is *paid*.   ◁

In the remainder of the section, we focus on the key properties of soundness and completeness of the backward reachability procedure encoded in MCMT, which can be used to handle the parameterised verification problem for CLog-nets defined above.[4] We call this procedure BReach, and in our context we assume it takes as input a marked CLog-net and an (undesired) property ψ, returning UNSAFE if there exists an instance of the catalog so that the net can evolve from the initial marking to a configuration that satisfies ψ, and SAFE otherwise. For details on the procedure itself, refer to [9,16]. We characterise the (meta-)properties of this procedure as follows.

Definition 5. *Given a marked CLog-net $\langle N, m_0 \rangle$ and a property ψ, BReach is: (i) sound if, whenever it terminates, it produces a correct answer; (ii) partially sound if a SAFE result it returns is always correct; (iii) complete (w.r.t. unsafety) if, whenever $\langle N, m_0 \rangle$ is UNSAFE with respect to ψ, then BReach detects it and returns UNSAFE.*   ◁

In general, BReach is not guaranteed to terminate (which is not surprising given the expressiveness of the framework and the type of parameterised verification tackled).

As we have seen in Sect. 3, the encoding of fresh variables requires to employ a limited form of universal quantification. This feature goes beyond the foundational framework for (data-driven) MCMT [9], which in fact does not explicitly consider fresh data injection. It is known from previous works (see, e.g., [3]) that when universal quantification over the indexes of an array is employed, BReach cannot guarantee that all the indexes are considered, leading to potentially spurious situations in which some indexes are simply "disregarded" when exploring the state space. This may wrongly classify a SAFE case as being UNSAFE, due to spurious exploration of the state space, similarly to what happens in lossy systems. By combining [9] and [3], we then obtain:

Theorem 1. BReach *is partially sound and complete for marked CLog-nets.*

                                                      ◁

---

[4] *Backward reachability* is not *marking reachability*. We consider reachability of a configuration satisfying a property that captures the covering of a data-aware marking.

Fortunately, MCMT is equipped with techniques [3] for debugging the returned result, and tame partial soundness. In fact, MCMT warns when the produced result is provably correct, or *may* have been produced due to a spurious state-space exploration.

A key point is then how to tame partial soundness towards recovering full soundness and completeness We obtain this by either assuming that the CLog-net of interest does not employ at all fresh variables, or is bounded.

Conservative CLog-nets are CLog-nets that do not employ ν-variables in arc inscriptions. It turns out that such nets are fully compatible with the foundational framework in [9], and consequently inherit all the properties established there. In particular, we obtain that BREACH is a semi-decision procedure.

Theorem 2. BREACH *is sound and complete for marked, conservative CLog-nets.*   ◁

One may wonder whether studying conservative nets is meaningful. We argue in favour of this by considering modelling techniques to "remove" fresh variables present in the net. The first technique is to ensure that ν-variables are used only when necessary. As we have extensively discussed at the end of Sect. 2, this is the case only for objects that are referenced by other objects. This happens when an object type participates on the "one" side of a many-to-one relationship, or for one of the two end points of a one-to-one relationship. The second technique is to limit the scope of verification by singling out only one (or a bunch of) "prototypical" object(s) of a given type. This is, e.g., what happens when checking soundness of workflow nets, where only the evolution of a single case from the input to the output place is studied.

Example 5. We can turn the CLog-net of Example 3 into a conservative one by removing the new order transition, and by ensuring that in the initial marking one or more order tokens are inserted into the *working* place. This allows one to verify how these orders co-evolve in the net. A detected issue carries over the general setting where orders can be arbitrarily created.   ◁

A third technique is to remove the part of the CLog-net with the fresh objects creation, assuming instead that such objects are all "pre-created" and then listed in a read-only, catalog relation. This is more powerful than the first technique from above: now verification considers all possible configurations of such objects as described by the catalog schema. In fact, using this technique on Example 3 we can turn the CLog-net into a conservative CLog-net that mimics exactly the behaviour of the original one.

Example 6. We make the CLog-net from Example 3 conservative in a way that reconstructs the original, arbitrary order creation. To do so we extend the catalog with a unary relation schema *CrOrder* accounting for (pre-)created orders. Then, we modify the new order transition: we substitute the ν-variable ν_o with a normal variable o, and we link this variable to the catalog, by adding as a guard a query *CrOrder*(o). This modified new order transition extracts an

order from the catalog and making it *working*. Since in the original CLog-net the creation of orders is unconstrained, it is irrelevant for verification if all the orders involved in an execution are created on-the-fly, or all created at the very beginning. Paired with the fact that the modified CLog-net is analysed for all possible catalog instances, i.e., all possible sets of pre-created orders, this tells us that the original and modified nets capture the same relevant behaviours. ◁

Bounded CLog-nets. An orthogonal approach is to study what happens if the CLog-net of interest is bounded (for a given bound). In this case, we can "compile away" fresh-object creation by introducing a place that contains, in the initial marking, enough provision of pre-defined objects. This effectively transforms the CLog-net into a conservative one, and so Theorem 2 applies. If we consider a boudned CLog-net and its catalog is acyclic (i.e., its foreign keys cannot form referential cycles where a table directly or indirectly refers to itself), then it is possible to show using the results from [9] that verifying safety of conservative CLog-nets becomes decidable.

Several modelling strategies can be adopted to turn an unbounded CLog-net into a bounded one. We illustrate two strategies in the context of our running example.

Example 7. Consider again the CLog-net of Example 3. It has two sources of unboundedness: the creation of orders, and the addition of items to working orders. The first can be tackled by introducing suitable resource places. E.g., we can impose that each order is controlled by a manager and can be created only when there is an idle manager not working on any other order. This makes the overall amount of orders unbounded over time, but bounded in each marking by the number of resources. Items creation can be bounded by imposing, conceptually, that each order cannot contain more than a maximum number of items. This amounts to impose a maximum multiplicity on the "many" side of each one-to-many relation implicitly present in the CLog-net.   ◁

5   Comparison to Other Models

We comment on how the CLog-nets relate to the most recent data-aware Petri net-based models, arguing that they provide an interesting mix of their main features.

DB-nets. CLog-nets in their full generality match with an expressive fragment of the DB-net model [22]. DB-nets combine a control-flow component based on CPNs with fresh value injection a là ν-PNs with an underlying read-write persistent storage consisting of a relational database with full-fledged constraints. Special "view" places in the net are used to inspect the content of the underlying database, while transitions are equipped with database update operations.

In CLog-nets, the catalog accounts for a persistent storage solely used in a "read-only" modality, thus making the concept of view places rather unnecessary. More specifically, given that the persistent storage can never be changed but only

queried for extracting data relevant for running cases, the queries from view places in DB-nets have been relocated to transition guards of CLog-nets. While CLog-nets do not come with an explicit, updatable persistent storage, they can still *employ places and suitably defined subnets to capture read-write relations and their manipulation.* In particular, as shown in [23], read-write relations queried using $\mathrm{UCQ}_\mathfrak{O}^-$ queries can be directly encoded with special places and transitions at the net level. The same applies to CLog-nets.

While verification of DB-nets has only been studied in the bounded case, CLog-nets are formally analysed here without imposing boundedness, and parametrically w.r.t. read-only relations. In addition, the MCMT encoding provided here constitutes the first attempt to make this type of nets practically verifiable.

PNIDs. The net component of our CLog-nets model is equivalent to the formalism of Petri nets with identifiers (PNIDs [17]) without inhibitor arcs. Interestingly, PNIDs without inhibitor arcs form the formal basis of the *Information Systems Modelling Language* (ISML) defined in [24]. In ISML, PNIDs are paired with special CRUD operations to define how relevant facts are manipulated. Such relevant facts are structured according to a conceptual data model specified in ORM, which imposes structural, first-order constraints over such facts. This sophistication only permits to formally analyse the resulting formalism by bounding the PNID markings and the number of objects and facts relating them. The main focus of ISML is in fact more on modelling and enactment. CLog-nets can be hence seen as a natural "verification" counterpart of ISML, where the data component is structured relationally and does not come with the sophisticated constraints of ORM, but where parameterised verification is practically possible.

Proclets. CLog-nets can be seen as a sort of *explicit data* version of (a relevant fragment of) Proclets [14]. Proclets handle multiple objects by separating their respective subnets, and by implicitly retaining their mutual one-to-one and one-to-many relations through the notion of correlation set. In Fig. 1, that would require to separate the subnets of orders, items, and trucks, relating them with two special one-to-many channels indicating that multiple items belong to the same order and loaded in the same truck.

A correlation set is established when one or multiple objects $o_1, \ldots, o_n$ are co-created, all being related to the same object o of a different type (cf. the creation of multiple items for the same order in our running example). In Proclets, this correlation set is implicitly reconstructed by inspecting the concurrent histories of such different objects. Correlation sets are then used to formalise two sophisticated forms of synchronisation. In the *equal* synchronisation, o flows through a transition t_1 while, simultaneously, *all* objects $o_1, \ldots, o_n$ flow through another transition t_2. In the *subset* synchronisation, the same happens but only requiring a subset of $o_1, \ldots, o_n$ to synchronise.

Interestingly, CLog-nets can encode correlation sets and the subset synchronisation semantics. A correlation set is explicitly maintained in the net by imposing that the tokens carrying $o_1, \ldots, o_n$ also carry a reference to o. This is what happens for items in our running example: they explicitly carry a reference to the

order they belong to. Subset synchronisation is encoded via a properly crafted subnet. Intuitively, this subnet works as follows. First, a lock place is inserted in the CLog-net so as to indicate when the net is operating in a normal mode or is instead executing a synchronisation phase. When the lock is taken, some objects in $o_1, \ldots, o_n$ are nondeterministically picked and moved through their transition t_2. The lock is then released, simultaneously moving o through its transition t_1. Thanks to this approach, a Proclet with subset synchronisation points can be encoded into a corresponding CLog-net, providing for the first time a practical approach to verification. This does not carry over Proclets with equal synchronisation, which would allow us to capture, in our running example, sophisticated mechanisms like ensuring that when a truck moves to its destination, *all* items contained therein are delivered. Equal synchronisation can only be captured in CLog-nets by introducing a data-aware variant of wholeplace operation, which we aim to study in the future.

6 Conclusions

We have brought forward an integrated model of processes and data founded on CPN that balances between modelling power and the possibility of carrying sophisticated forms of verification parameterised on read-only, immutable relational data. We have approached the problem of verification not only foundationally, but also showing a direct encoding into MCMT, one of the most well-established model checkers for the verification of infinite-state dynamic systems. We have also shown that this model directly relates to some of the most sophisticate models studied in this spectrum, attempting at unifying their features in a single approach. Given that MCMT is based on Satisfiability Modulo Theories (SMT), our approach naturally lends itself to be extended with numerical data types and arithmetics. We also want to study the impact of introducing wholeplace operations, essential to capture the most sophisticated syhncronization semantics defined for Proclets [14]. At the same time, we are currently defining a benchmark for data-aware processes, systematically translating the artifact systems benchmark defined in [20] into corresponding imperative data-aware formalisms, including CLog-nets.

References

1. MCMT: Model checker modulo theories. http://users.mat.unimi.it/users/ghilardi/mcmt/. Accessed 15 June 2020
2. Aalst, W.M.P.: Object-centric process mining: dealing with divergence and convergence in event data. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 3–25. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_1

3. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. J. Satisf. Boolean Model. Comput. **8**(1/2), 29–61 (2012)

4. Artale, A., Kovtunova, A., Montali, M., van der Aalst, W.M.P.: Modeling and reasoning over declarative data-aware processes with object-centric behavioral constraints. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) BPM 2019. LNCS, vol. 11675, pp. 139–156. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26619-6_11

5. Batoulis, K., Haarmann, S., Weske, M.: Various notions of soundness for decision-aware business processes. In: Mayr, H.C., Guizzardi, G., Ma, H., Pastor, O. (eds.) ER 2017. LNCS, vol. 10650, pp. 403–418. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69904-2_31

6. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Verification of data-aware processes via array-based systems (extended version). Technical report arXiv:1806.11459 (2018)

7. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Formal modeling and SMT-based parameterized verification of data-aware BPMN. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) BPM 2019. LNCS, vol. 11675, pp. 157–175. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26619-6_12

8. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: From model completeness to verification of data aware processes. In: Lutz, C., Sattler, U., Tinelli, C., Turhan, A.-Y., Wolter, F. (eds.) Description Logic, Theory Combination, and All That. LNCS, vol. 11560, pp. 212–239. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22102-7_10

9. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: SMT-based verification of data-aware processes: a model-theoretic approach. Math. Struct. Comput. Sci. **30**(3), 271–313 (2020)

10. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: a database theory perspective. In: Proceedings of PODS, pp. 1–12. ACM (2013)

11. De Masellis, R., Di Francescomarino, C., Ghidini, C., Montali, M., Tessaris, S.: Add data into business process verification: bridging the gap between theory and practice. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of AAAI, pp. 1091–1099 (2017)

12. Deutsch, A., Li, Y., Vianu, V.: Verification of hierarchical artifact systems. In: Proceedings of PODS, pp. 179–194. ACM (2016)

13. Dumas, M.: On the convergence of data and process engineering. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 19–26. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23737-9_2

14. Fahland, D.: Describing behavior of processes with many-to-many interactions. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_1

15. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Petri nets with parameterised data: modelling and verification (extended version). Technical report arXiv:2006.06630 (2020)

16. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4), 1–46 (2010)

17. van Hee, K.M., Sidorova, N., Voorhoeve, M., van der Werf, J.M.E.M.: Generation of database transactions with petri nets. Fundamenta Informaticae **93**(1–3), 171–184 (2009)

18. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88873-4_17

19. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: fundamental requirements and their support in existing approaches. Int. J. Inf. Syst. Model. Des. **2**(2), 19–46 (2011)

20. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: a practical verifier for artifact systems. PVLDB **11**(3), 283–296 (2017)

21. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40176-3_14

22. Montali, M., Rivkin, A.: DB-Nets: on the marriage of colored petri nets and relational databases. Trans. Petri Nets Other Models Concurr. **28**(4), 91–118 (2017)

23. Montali, M., Rivkin, A.: From DB-nets to coloured petri nets with priorities. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 449–469. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_24

24. Polyvyanyy, A., van der Werf, J.M.E.M., Overbeek, S., Brouwers, R.: Information systems modeling: language, verification, and tool support. In: Giorgini, P., Weber, B. (eds.) CAiSE 2019. LNCS, vol. 11483, pp. 194–212. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21290-2_13

25. Reichert, M.: Process and data: two sides of the same coin? In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 2–19. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_2

26. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability and complexity of petri nets with unordered data. Theoret. Comput. Sci. **412**(34), 4439–4451 (2011)