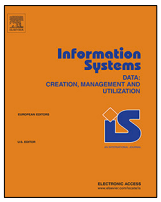
Information Systems 107 (2022) 102011

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Petri net-based object-centric processes with read-only data ☆

Silvio Ghilardi [a], Alessandro Gianola [b], Marco Montali [b], Andrey Rivkin [b,*]

[a] Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy
[b] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy

ARTICLE INFO

Article history:
Received 1 February 2021
Received in revised form 19 December 2021
Accepted 16 February 2022
Available online 25 February 2022
Recommended by Manfred Reichert

Keywords:
Higher-level net models
Relationships between Petri nets and other approaches
Verification

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 case objects that need to co-evolve. In this work, we introduce and study an extension of coloured Petri nets, called catalogue and object-aware nets (COA-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 demonstrate that this class of nets can be used for representing various multi-case modelling scenarios involving objects with one-to-many correlations. We then study a parameterised verification problem of COA-nets and show how to systematically encode them into one of the reference frameworks for attacking that kind of problem in the context of infinite-state systems with data. We demonstrate that different fragments of COA-nets can have different expressive power that can affect the decidability of not only the verification problem at hand, but also the standard problem of place nonemptiness checking. Finally, we discuss how COA-nets relate to well-known formalisms in the area of multi-case and data-aware process modelling and analysis.

1. Introduction

The integration of control flow and data has become one of the most prominently investigated topics in BPM [1]. Taking into account data when working with processes is crucial to properly understand which courses of execution are allowed [2], to account for decisions [3], and to explicitly accommodate business policies and constraints [4]. 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 [5,6] to enactment [7,8] and mining [9]. 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 [10,11]. Several modelling paradigms have adopted to tackle this and other important features: data-/artefact-centric approaches [5,6], declarative languages based on temporal constraints [11], and imperative, Petri net-based notations [10,12,13]. 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 catalogue 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

☆ This work has been partially supported by the UNIBZ projects VERBA, DUB, and QUEST.

* Corresponding author.
E-mail addresses: silvio.ghilardi@unimi.it (S. Ghilardi), gianola@inf.unibz.it (A. Gianola), montali@inf.unibz.it (M. Montali), rivkin@inf.unibz.it (A. Rivkin).

https://doi.org/10.1016/j.is.2022.102011
0306-4379/

Petri nets [10,12,13], the latter has been extensively investigated in the data-centric spectrum [14,15], but only recently imported 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) [16].

By relying on a preliminary version of this work [17], we reconcile these two themes in an extension of coloured Petri nets (CPNs) called *catalogue and object-aware nets* (COA-nets). On the one hand, in COA-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 provide a set of modelling guidelines to highlight these features in a practical spectrum. We then study the parameterised verification problem for COA-nets, in which one can check correctness of various properties for any possible catalogue instance. To attack this problem, we systematically encode COA-nets into the most recent version of MCMT [18,19], one of the few model checkers natively supporting the (parameterised) verification of infinite-state systems [20–24], especially those involving both data and processes [15,25,26]. Moreover, we demonstrate an application of the aforementioned encoding to a simple COA-net and show how to address currently existing limitations of the MCMT tool without losing the expressive power of verified models. We also investigate in detail the expressive power of COA-nets and show its relation to the feasibility and decidability of the verification problem. We then stress that, thanks to the encoding into MCMT, a relevant fragment of the model can be readily verified using MCMT, and that verification of the whole model is within reach. Finally, we discuss how COA-nets provide a unifying approach for some of the most sophisticated formalisms in this area, highlighting differences and commonalities.

2. The COA-net formal model

In this section, we present key concepts and notions used for defining catalogue-nets. Conceptually, a COA-net integrates two key components. The first is a read-only persistent data storage, called *catalogue*, to account for read-only, parameterised data. The second is a variant of CPN, called ν-CPN [27], to model the process backbone. Places in ν-CPNs 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 [10,13,27], the net employs ν-variables (first studied in the context of ν-PNs [28]) to inject fresh data (such as object identifiers). A distinguishing feature of COA-nets is that transitions can have guards that inspect and retrieve data objects from the read-only catalogue. At the end of the section we discuss in more detail all the previously mentioned characteristics using a COA-net in Fig. 6.

Data types. We consider a *type set* $\mathfrak{D}$ as a finite set of pairwise disjoint types accounting for the different kinds of objects in the domain of interest. We partition this finite set of types in two disjoint subsets: the subset of *id sorts* $\mathfrak{D}_{id}$ and the subset of *value sorts* $\mathfrak{D}_{val}$ (following the nomenclature of [15]). Conceptually, a type in $\mathfrak{D}_{id}$ is called id sort and accounts for *identifiers* of different kinds of objects, while a type in $\mathfrak{D}_{val}$ is called value sort and accounts for *value data types* such as strings, numbers. Each type $\mathcal{D} \in \mathfrak{D}$ comes with its own *domain* $\Delta_{\mathcal{D}}$, and with an equality operator $=_{\mathcal{D}}$: in line with [15], we require that when $\mathcal{D} \in \mathfrak{D}_{id}$, the domain $\Delta_{\mathcal{D}}$ is *finite*, whereas when $\mathcal{D} \in \mathfrak{D}_{val}$, the domain $\Delta_{\mathcal{D}}$ is possibly infinite. Domains of different types are pairwise disjoint. When clear from the context, we simplify the notation of the equality operator $=_{\mathcal{D}}$ and only use $=$. We assume that each type $\mathcal{D} \in \mathfrak{D}$ comes with a special constant $\mathtt{undef}_{\mathcal{D}} \in \mathfrak{D}$ to denote an undefined value in that domain. Here are a few examples of types: value sorts such as strings $\mathtt{string} = \langle \mathbb{S}, =_s \rangle$ and integers $\mathtt{int} = \langle \mathbb{Z}, =_{int} \rangle$, and id sorts like orders $\mathtt{Order} = \langle \mathbb{O}, =_o \rangle$ and product types $\mathtt{ProdType} = \langle \mathbb{P}, =_p \rangle$ used in some e-commerce application. Our definition of data types is essentially used to symbolically distinguish different domains. One could see it as an extension of the pure names concept adopted in [28,29] that allows to distinguish different "kinds" of names.

Catalogue. $R(a_1 : \mathcal{D}_1, \ldots, a_n : \mathcal{D}_n)$ is a $\mathfrak{D}$-*typed relation schema*, where R is a relation name and $a_i : \mathcal{D}_i$ indicates the ith 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 catalogue (schema)* $\mathcal{R}_{\mathfrak{D}}$ is a finite set of $\mathfrak{D}$-typed relation schemas. A $\mathfrak{D}$-*typed catalogue instance Cat over* $\mathcal{R}_{\mathfrak{D}}$ is a finite set of facts $R(\mathtt{o}_1, \ldots, \mathtt{o}_n)$, where $R \in \mathcal{R}_{\mathfrak{D}}$ and $\mathtt{o}_i \in \Delta_{\mathcal{D}_i}$, for $i \in \{1, \ldots, n\}$.

We adopt some natural *constraints* in the catalogue relations. First, we assume the first attribute of every relation $R \in \mathcal{R}_{\mathfrak{D}}$ to be its *primary key*, denoted as $\textsc{pk}(R)$: the type of a primary key needs to be an id sort. Also, the type of such an attribute should be different from the types of other primary key attributes. Then, for any $R, S \in \mathcal{R}_{\mathfrak{D}}$, $R.a \rightarrow S.id$ defines that the projection $R.a$ is a *foreign key* referencing $S.id$, where $\textsc{pk}(S) = id$, $\textsc{pk}(R) \neq a$ and $\mathcal{D} = \mathcal{D}'$, for $id : \mathcal{D}$ and $a : \mathcal{D}'$. We also assume that every id sort $\mathcal{D} \in \mathfrak{D}_{id}$ determines the primary key of some n-ary $\mathfrak{D}$-typed catalogue relation $R_{\mathcal{D}}$, in the sense that every element $s_1 \in \Delta_{\mathcal{D}} \setminus \{\mathtt{undef}_{\mathcal{D}}\}$ is the first component of some tuple $(s_1, \ldots, s_n)$ such that the fact $R_{\mathcal{D}}(s_1, \ldots, s_n)$ is in the catalogue instance Cat. 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 [14,15].

Example 1. Consider a simple catalogue 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. ◁

Catalogue queries. We fix a countably infinite set $\mathcal{V}_{\mathfrak{D}}$ of typed variables with a *variable typing function* $\mathtt{type} : \mathcal{V}_{\mathfrak{D}} \rightarrow \mathfrak{D}$. 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)* $\varphi ::= y_1 = y_2 \mid y_1 \neq y_2 \mid \varphi \wedge \varphi \mid \top$ is a *condition*, s.t. y_i is either a variable of type $\mathcal{D}$ or a constant from $\Delta_{\mathcal{D}}$. $\mathtt{CQ}^{\neg}_{\mathfrak{D}}$ 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, $\mathtt{UCQ}^{\neg}_{\mathfrak{D}}$ denotes the set off all *unions of conjunctive queries* over $\mathcal{R}_{\mathfrak{D}}$. Each query $Q \in \mathtt{UCQ}^{\neg}_{\mathfrak{D}}$ has the form $Q = \bigvee_{i=1}^{n} Q_i$, with $Q_i \in \mathtt{CQ}^{\neg}_{\mathfrak{D}}$.

A *substitution* for a set $X = \{x_1, \ldots, x_n\}$ of typed variables is a function $\theta : X \rightarrow \Delta_{\mathfrak{D}}$, 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 catalogue instance Cat is a set of substitutions $ans(Q, Cat) = \{\theta : Free(Q) \to Val(Cat) \mid Cat, \theta \models Q\}$, where $Val(Cat)$ denotes the set of all constants occurring in Cat and $\models$ denotes standard FO entailment.

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

COA-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 $S(a) = n$. 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 $a \in A$. In what follows, with slight abuse of notation we assume that functions `type`, *Vars* and *Const* are extended to account for sets, tuples and multisets of variables and constants. For example, $Vars(\{x, 1, \mathtt{a}, y, z\}) = \{x, y, z\}$ and $Const(\{x, 1, \mathtt{a}, y, z\}) = \{1, \mathtt{a}\}$.

We now define COA-nets, extending ν-CPNs [27] with the ability of querying a read-only catalogue. As in CPNs, each COA-net place has a colour 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. To account for fresh external inputs, we employ the well-known mechanism of ν-Petri nets [28] 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 its domain, which is finite if the type of ν is an id sort, otherwise it is (possibly) infinite (in the last case it 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 COA-net $\mathcal{N}$ over a catalogue 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. $\mathtt{color} : P \to \mathcal{K}_{\mathfrak{D}}$ is a place typing function, where $\mathcal{K}_{\mathfrak{D}}$ is a set of all possible cartesian products $\mathcal{D}_1 \times \cdots \times \mathcal{D}_m$, s.t. $\mathcal{D}_i \in \mathfrak{D}$, for each $i = 1, \ldots, m$;
3. $F_{in} : P \times T \to \Omega^{\oplus}_{\mathcal{V}_{\mathfrak{D}}}$ is an input flow, s.t. $\mathtt{type}(F_{in}(p, t)) = \mathtt{color}(p)$ 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. $\mathtt{type}(F_{out}(t, p)) = \mathtt{color}(p)$ for every $(t, p) \in T \times P$;
5. $\mathtt{guard} : T \to \{Q \wedge \varphi \mid Q \in \mathtt{UCQ}^{\neg}_{\mathfrak{D}}, \varphi \in \mathcal{C}_{\mathfrak{D}}\}$ is a partial guard assignment function, s.t., for every $\mathtt{guard}(t) = Q \wedge \varphi$ 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 $Free(Q) \subseteq Vars(t)$, 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 catalogue 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, if there are some variables (excluding the fresh ones) in the outgoing arc inscriptions that do not appear in $InVars(t)$, then these are the free variables of Q. Such variables, in turn, are used in the net to propagate data from the catalogue via query answers. Moreover, it is required that all free variables of Q are exhaustively covered by the variables in the input and output arcs. This condition essentially forbids to have queries producing answers that are (partially) not going to be used in the net. As customary in high-level Petri nets, using the same variable in two different arc inscriptions amounts to checking the equality between the respective components of such inscriptions. For every transition $t \in T$, we also define ${}^\bullet t = \{p \in P \mid (p, t) \in \text{DOM}(F_{in})\}$ as a *pre-set* of t and $t^\bullet = \{p \in P \mid (t, p) \in \text{DOM}(F_{out})\}$ as a *post-set* of t.[1]

Semantics. The execution semantics of a COA-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 COA-net $N = (\mathfrak{D}, \mathcal{R}_{\mathfrak{D}}, P, T, F_{in}, F_{out}, \mathtt{color}, \mathtt{guard})$ is a function $m : P \to \Omega^{\oplus}_{\mathfrak{D}}$, so that $m(p) \in \Delta^{\oplus}_{\mathtt{color}(p)}$ for every $p \in P$. We write $\langle N, m, Cat\rangle$ to denote COA-net N marked with m, and equipped with a read-only catalogue 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 $\sigma = \{x \mapsto 1, y \mapsto \mathtt{a}\}$, its application to a multiset of tuples $\omega = \{\langle x, y\rangle^2, \langle x, \mathtt{b}\rangle\}$ results in $\sigma(\omega) = \{\langle 1, \mathtt{a}\rangle^2, \langle 1, \mathtt{b}\rangle\}$.

Next, we define when a transition can be called *enabled*. 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 catalogue instance, so that the data they carry make the guard attached to the transition true.

Definition 2. A transition $t \in T$ is *enabled* in a marking m and a fixed catalogue 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 $p \in P$; *(ii)* $\sigma(\mathtt{guard}(t))$ is true; *(iii)* for every fresh variable $\nu \in \Upsilon_{\mathfrak{D}} \cap OutVars(t)$, we have that $\sigma(\nu) \in \Delta_{\mathtt{type}(\nu)} \setminus Val(m)$;[2] *(iv)* $\sigma(x) = \theta(x)$, for $\theta \in ans(Q, Cat)$, $x \in (OutVars(t) \setminus InVars(t)) \cap Vars(Q)$ and query Q from $\mathtt{guard}(t)$.[3] ◁

In the definition, point *(iii)* constraints the possible bindings for fresh variables. If $\Delta_{\mathtt{type}(\nu)}$ of some fresh variable ν is the domain of an id sort (i.e., if the type of the variable ν is an id sort), it may be the case that the identifiers contained in the catalogue for that type are all present in the current marking; in this extreme case, *(iii)* indicates that the transition cannot fire. If instead $\Delta_{\mathtt{type}(\nu)}$ is the domain of a value sort, there is always a way to pick a suitable, fresh value for ν.

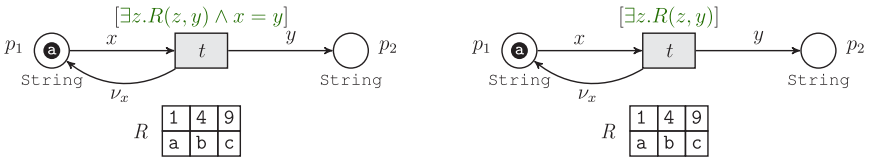
When a transition t is enabled, it may fire. Next we define what are the effects of firing a transition with some binding σ.

Definition 3. Let $\langle N, m, Cat\rangle$ be a marked COA-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 $\tau \in T^*$. ◁

[1] DOM(f) denotes a domain of function f.

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

[3] This condition stipulates that the binding needs to "agree" on the result of Q when it comes to the variables of Q that are used in the output inscriptions of t, and have not been already bound by the variables in the input inscriptions of the same transition.



(a) A bounded marked COA-net

(b) Unbounded variant of the net in Figure 1a

Fig. 1. Boundedness in COA-nets with a fixed catalogue instance (fixing pairs in binary relation R).

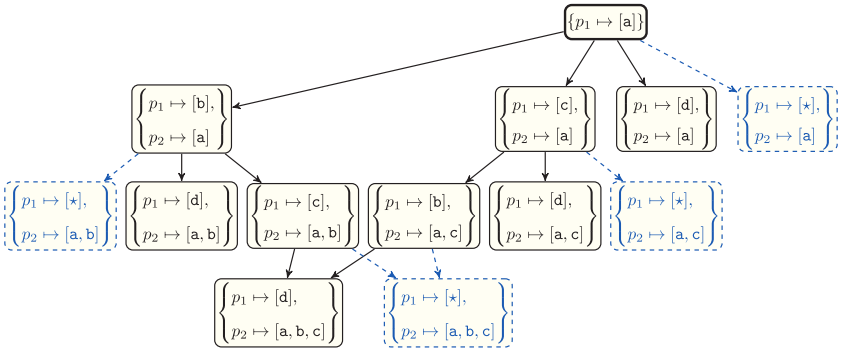


Fig. 2. Transition system capturing the execution semantics of the marked net from Fig. 1(a). Symbol $\star$ denotes a string value different from a, b, c, d (there exist infinitely many such $\star$).

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.

Definition 4. Given $b \in \mathbb{N}$, place p in a marked COA-net $\langle N, m_0, Cat\rangle$ is called *b-bounded* if $|m(p)| \leq b$, for every marking $m \in \mathcal{M}(N)$. The same net is called *bounded with bound b* if every place $p \in P$ is b-bounded. ◁

Unboundedness in COA-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 catalogue via queries in transition guards.

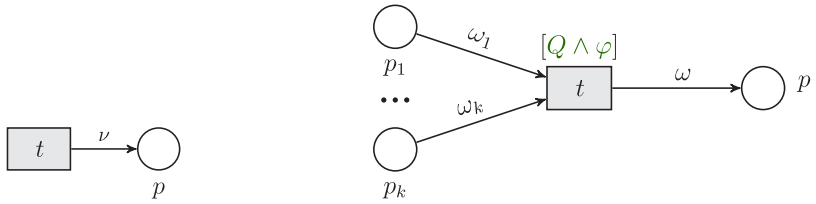
Example 3. Fig. 1 demonstrates two almost identical marked COA-nets, with a catalogue fixing pairs in a binary relation R whose first component defines the primary key for R, and the second component is a string attribute. The net represented in Fig. 1(a) is bounded. This can be seen considering that the guard of t extracts strings stored in the second component of R and compares them to those in place p_1, consequently allowing t to fire only if those values coincide. Moreover, after each firing, the token stored in p_1 is consumed, bringing back to p_1 a token carries a "locally fresh" string value generated using ν_x. Being locally fresh, such a value must be different from any string value present p_1 and p_2. This means that, sooner or later, a string not contained in the second component of R (i.e., different from a, b, and c) will need to be selected when binding ν_x. When this happens, t will not fire anymore. All in all, this means that the marked net will never assign more than one token to p_1, and no more than three tokens to p_2 (where three is indeed the number of distinct string values contained in the second component of R). Differently from the marked net discusses so far, its variant shown Fig. 1(b) is unbounded. In this case, the loop relating p_1 and t preserves the bound of one token for p_1, no restrictions are imposed on $\texttt{guard}(t)$ on the token consumed from p_1. This, in turn, indicates that t can fire unboundedly many times, inflating p_2 with unboundedly many tokens, each carrying one of the three string values a, b, and c extracted from the catalogue. This shows an example of unboundedness arising even if the number of values simultaneously present in the current marking stays bounded. ◁

Execution semantics. The execution semantics of a marked COA-net $\langle N, m_0, Cat\rangle$ is defined in terms of a possibly infinite-state transition system in which states are labelled 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. Technically, let $\langle N, m_0, Cat\rangle$ be a marked COA-net with catalogue instance Cat. Then its execution semantics is captured by transition system $\Lambda_N = (S, s_0, \rightarrow)$, where:

- S is a possibly infinite set of markings over N;
- $\rightarrow \subseteq S \times T \times S$ is a T-labelled transition relation between pairs of markings;
- S and $\rightarrow$ are defined by simultaneous induction as the smallest sets satisfying the following conditions: *(i)* $m_0 \in S$; *(ii)* given $m \in S$, for every transition $t \in T$, binding σ and marking m' over N, if $m[t\rangle_{Cat} m'$, then $m' \in S$ and $m \xrightarrow{t} m'$.

As pointed out before, we are interested in analysing a COA-net irrespectively of the actual content of the catalogue. Hence, in the following when we mention a (catalogue-parameterised) marked net $\langle N, m_0\rangle$ without specifying how the catalogue 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 catalogue schema of N.

Example 4. Fig. 2 shows the transition system capturing the execution semantics of the marked net from Fig. 1(a), whose initial marking m_0 assigns to p_1 one token carrying the string value a. For ease of reading, we omit the catalogue instance and arc labels. Notice that, although our net is bounded, its transition



(a) Object generating emitter (b) Catalog-based object constructor

Fig. 3. Object creation patterns. In Fig. 3(b), $Free(Q) \neq \emptyset$ and $Vars(\omega) \subseteq Free(Q)$.

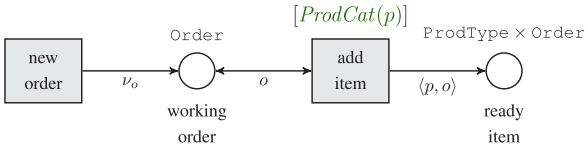


Fig. 4. An example of using both patterns from Fig. 3.

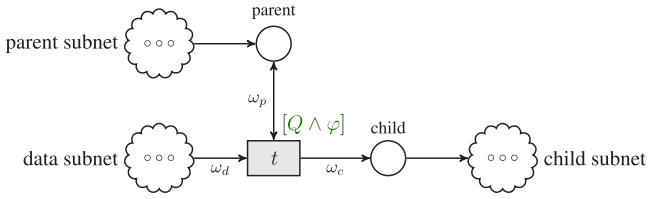


Fig. 5. A pattern for creating objects involved in a one-to-many relationship. Here, ω_c contains only variables taken from $Vars(\omega_d)$, $Vars(\omega_p)$ and $Free(Q)$.

system contains infinitely many states due to the presence of the ν-variable ν_x that, in every firing, can be bound to any string value that is not present in current net marking.

2.1. Modelling capabilities

Given that COA-net models can conceptually address two seemingly disjoint dimensions, one may wonder about the modelling capabilities of the COA-net formalism and, in particular, more recurring *modelling patterns* that could be of relevance for domain experts that would like to use COA-nets for representing data- and process-aware information systems. While there are already works that tackle pattern-based modelling approaches using various classes of Petri nets (see, for example, [30, 31]), we focus on specifying patterns that cover generation and management of *multiple* case objects.

When choosing to opt for a setting in which management of case objects is no less important than correct representation of control flow, it is crucial to ensure that such setting lends enough of expressive power for creating, deleting or updating the case objects as well as takes into account their types and relations between each other. In COA-nets, objects are represented as typed tokens. Introducing a new object to a net, apart from using a traditional token generation mechanism, can be done by either using fresh variables or by materialising such object based on results of a query performed on top of a catalogue. The first case is rather limited only to *emitter* transitions and enforces the concept of identity over created objects (that is, every object created with this pattern is *identified* with a unique element taken from the domain of the related variable from $\Upsilon_{\mathfrak{D}}$). The related *object generating emitter* pattern is demonstrated in Fig. 3(a). As an example of its application in practice, one can think of an emitter transition that allows to generate (fresh) identifiers of unboundedly many orders in some order-to-delivery process.

The second *catalogue-based object constructor* pattern is shown in Fig. 3(b). Upon firing of transition t, it extracts some relevant data from the catalogue using query Q in the guard, and then binds extracted values to variables in ω. This allows to create objects that can potentially coalesce control flow data (taken from some of the incoming places of t) with information available in the persistent storage. A simple use case for this pattern can be a direct follow-up of the order creation example. Assuming that our net contains order identifiers, previously generated with emitter new order, and the catalogue is the one described in Example 1, we can use the former in order to generate new objects specifying an item (of a certain type) being assigned to an order. Fig. 4 shows that the creation of items is not modelled using an explicit ν-variable, but is instead simply obtained by the add item transition which enriches a selected order token with the product type taken from the catalogue using the query assigned to add item.

As a side remark, we briefly comment on the overall modelling power of the catalogue. In general, it is common to have processes that only alter a portion of all the data they have access to (think, for example, of product types, employees and carriers in an order-to-cash process), or processes that by design can only access some (possibly structured) data in a read-only mode. Given that data objects can also be related to one another, this naturally calls for introducing a static storage to keep track of such read-only relations. In our framework, these relations may be represented with special read-only places (as it is essentially done in [27]). However, using places for read-only relations would not naturally allow one to capture key and foreign key constraints, nor to distinguish the query language used to inspect them from that used for normal places.[4] And even if it were the case, then resulting model will be conceptually too confusing. This is the main reason for having a separate representation.

Let us now focus on patterns that help modelling management of relations or, better said, dependencies between case objects. In particular, we show how to address one-to-many relations using the language of COA-nets. We deliberately avoid the trivial one-to-one case as well as the case of many-to-many relations. The latter, however, can be easily represented by simply reifying the relation in a new separate object and producing a consequent pair of one-to-many relations, each of which keeping the new object on the "one" side. In the context of Petri nets, this approach is very much in line with the one recently proposed in [10].

Whenever we want to create an object that is meant to participate in a one-to-many relationship on the "many" side, it has to carry a data of a parent object it is related to. Although such data can be of any type, we suggest to bring its complexity to a bare minimum and use object identifiers whenever possible. Fig. 5 demonstrates a schematic overview of a generic COA-net for this case. As opposed to object generation patterns, it is crucial

[4] There are also technical results on verification clearly showing that there is a difference between the way read-only and read–write relations can be constrained and queried [15].

to induce the conceptual separation between different subnets that contribute to the creation of case objects and that, at the same time, show their lifecycles. Thus, we use three subnets, two of which show the lifecycle of the parent and child objects (respectively stored in *parent* and *child* places) taking part in the one-to-many relation, and another subnet that, on top of the query Q, is used as an independent data supplier with its own potentially complex workflow.[5] Notice that the last subnet is essentially a combination of object generating emitters and catalogue-based constructors. Although it is not explicitly shown in Fig. 5, the parent subnet considers the parent object lifecycle including the creation step (which can be realised using one of the patterns from Fig. 3, proviso that newly generated parent objects are duly equipped with unique identifiers as suggested above), whereas the creation of child objects is instrumented with transition t. Notice also that both the catalogue query and the data subnet can be avoided. However, that, in turn, requires the modeller to use ν-variables in ω_c. One may also wonder whether given the nature of the catalogue storage, dependencies between relations therein should be also taken into account and/or can be reflected in the net model. The answer on this matter becomes evident when looking into the nature of the queries. Strictly speaking, query answers do not carry over any constraints imposed on the source database. Thus, data values that the modeller gets in the net are just copies of those in the catalogue.

As an example of the pattern discussed above, one may use the one from Fig. 4. Here, each newly created item carries a reference to its owning order, and thus models the one-to-many relation between orders and items. 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 can be always changed by adding to the catalogue another relation $Item(c : \texttt{IId}, n : \texttt{Descr}, p : \texttt{ProdType})$ that stores information about items by capturing their identifiers, short content descriptions and product types, which, in turn provides information for adding identity to items in the net.

Lastly, we would like to comment on one more possible usage of complex objects in COA-nets. Quite often, processes require a presence of resources, both structured and unstructured, which are fixed in the process model domain. Our formalism lends its expressive power to account for both types. Resources should be represented as tokens assigned to dedicated places with the initial marking, and their amount initially present in the net should never change (that is, one shall never create new or destroy any of the already existing resources). A natural example of resources are shipping company employees involved in handling orders.

We conclude with an example that summarises all the main features of COA-nets.

Example 5. Starting from the catalogue in Example 1, Fig. 6 shows a simple, yet sophisticated example of COA-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 catalogue 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. Unpaid items added to a working order can be always removed, whereas paid ones can be loaded in a compatible truck. The set of available trucks, indicating their plate numbers and types, is contained in a dedicated *pool* 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 catalogue. 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 5 showcases various key aspects related to modelling data-aware processes with multiple case objects using COA-nets that have been discussed above. Using the pattern from Fig. 5, we demonstrate that 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 is the case for every item carrying a reference to its owning order and, once loaded into a truck, a reference to the truck plate number. One can also see that the above example manipulates three different object types that have been previously discussed in this section. Unboundedly many case objects representing orders can be genuinely created using the object generating emitter as it is demonstrated in Fig. 4. The (finite) set of trucks available in the domain is instead fixed in the *pool* place in Fig. 6 by the initial marking, and represents a pool of resources that can change state but are never destroyed nor created. Finally, objects representing items can be arbitrarily created and destroyed using the pattern from Fig. 5. Since items are in one-to-many relation with orders on the "many" side, their creation is not modelled using an explicit ν-variable, but is instead simply obtained by the add item transition which acquires product types from the catalogue using the query in $\texttt{guard}(\text{add item})$. This also demonstrates that, unlike orders and trucks, items do not require any identifiers as the scenario does not need either to track their lifecycle or to relate them to other child objects. Thus, one may also assert that ν-variables are only necessary when the COA-net needs to handle the arbitrary creation of objects that are referenced by other objects or when object identifiers are explicitly required in the modelling scenario.

We also use the above example to briefly illustrate one of the main (modelling) limitations of the COA-nets. By looking closely at transition rem, it is easy to notice that it can also delete items that have been already paid. In COA-nets, there is no workaround allowing to remove only unpaid items (obviously, only connecting transition rem to place paid will not suffice) as the formalism does not allow for universal quantification over objects stored in places. To support such quantification, one would need to introduce either relations with write access or whole-place operations.

3. From COA-nets to MCMT

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

MCMT is founded on the theory of *array-based systems* [18,19], 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.

[5] By independence we imply that the data coming from the data subnet are not meant to create any additional dependencies.

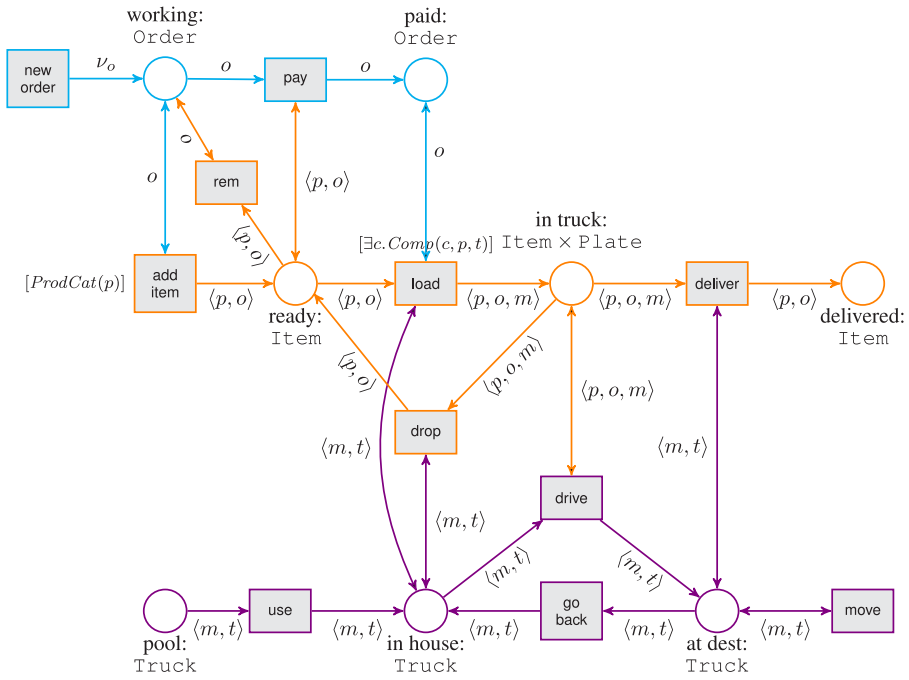


Fig. 6. A COA-net (its catalogue is in Example 1). In the picture, `Item` and `Truck` are compact representations for `ProdType` × `Order` and `Plate` × `TruckType` respectively. The top blue part refers to orders, the central orange part to items, and the bottom violet part to delivery trucks. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

3.1. Array-based systems: a summary

In general terms, an array-based system describes the evolution of array data structures of unbounded size. The logical representation of an array relies on a theory with two types of sorts: one for the array indexes, and the other for the elements stored in the array cells. Since the content of an array changes over time, it is represented by a *function* variable, called *array variable*, which defines for each index what is the value stored in the corresponding cell. In other terms, the interpretation of an array variable 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). Its interpretation changes when moving from one state to another, reflecting the intended manipulation of the array. Hence, an array-based system working over array a is defined through: *(i)* a state formula $I(a)$ describing the *initial configuration(s)* of a; *(ii)* a formula $\tau(a, a')$ describing the *transitions* that transforms the content of the array from a to a'. By suitably using logical operators, τ can express in a single formula a set of different updates over a. One of the most studied verification problems is that of *unsafety verification*: it checks whether the evolution induced by τ over a starting from a configuration in $I(a)$ eventually *reaches* an *unsafe* configuration described by a state formula $K(a)$.

In [15], array-based systems were extended towards an array-based version of the artefact-centric approach, considering in particular the sophisticated model in [32]. In the resulting formalism, called RAS, a *relational* artefact system accesses a read-only database with keys and foreign keys (cf. our definition of catalogue). In addition, the RAS operates over a set of evolving relations possibly containing unboundedly many updatable entries (cf. tokens in places of COA-nets). Fig. 7 gives an intuitive idea of how this type of system looks like: it is an easy example (from [16]) of a catalogue containing internal information maintained by a company about some job categories (the pink box) and a read–write evolving relation (formed of four columns) storing the information about job applications (e.g., a user who applies for a programmer position and her application is evaluated with a score like 85). The catalogue is treated as a rich, background theory, which can be considered as a more sophisticated version of the element sort in basic array systems. Each evolving relation is treated as a set of arrays, where each array accounts for one column of the corresponding evolving relation. A tuple in the relation is reconstructed by accessing all such arrays with the same index.

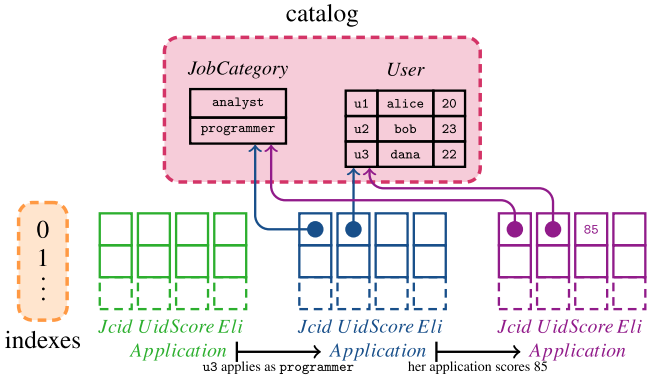


Fig. 7. Array-based representation of a RAS formalising the manipulation of a job application [16].

Verification of RAS is also tamed in [15], checking whether there exists an instance of the read-only database so that the RAS can reach an unsafe configuration. This is approached by extending the original symbolic *backward reachability* procedure for unsafety verification of array-based systems to the case of RAS. The procedure starts from the undesired states captured by $K(a)$, and iteratively computes so-called *preimages*, i.e., logical formulae symbolically describing those states that, through consecutive applications of τ, directly or indirectly reach configurations satisfying $K(a)$. A preimage formula may contain existentially quantified variables referring to data objects in the catalogue.

MCMT employs novel quantifier elimination techniques [33,34] to suitably remove such variables, and obtain a state formula that describes the predecessor states. A fixpoint check is delegated to a state-of-the-art SMT solver, so as to check whether the computed predecessors all coincide with already iteratively computed states. If no new state has been produced, the procedure stops by emitting *safe*. Otherwise, the preimage formula is conjoined with $I(a)$ and sent to the SMT solver to check for satisfiability: if so, then the computed preimage states intersect the initial ones, and the procedure stops by emitting *unsafe* as a verdict; if not, new preimages are iteratively computed and the steps above are repeated. In order to support the representation of read-only databases and the extension of backward reachability, one needs to adopt the module of MCMT called "database-driven mode', which is the one that will be used here too.

3.2. Encoding into MCMT

In this section, we show how to encode a COA-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 catalogue. Then, we present a step-wise encoding of the COA-net elements and the net semantics into the MCMT variant of array-based systems.

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 $\mathcal{D}$. We recall that MCMT does not only support user-defined data types, but also user-defined operations. However, in this paper we only consider the former and, as it has been already mentioned in Section 2, assume that every type has an equality operation defined for it. For more details on user defined operations please refer to the MCMT documentation (http://users.mat.unimi.it/users/ghilardi/mcmt/).

To represent the catalogue relations of $\mathcal{R}_{\mathfrak{D}} = \{R_1, \ldots, R_{n_r}\}$ in MCMT, we proceed as follows. Recall that in catalogue 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 [15]. 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., $R_i.A_jS.id$), 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 $\texttt{D\_Aj} = \texttt{type}(A_j)$, 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)
```

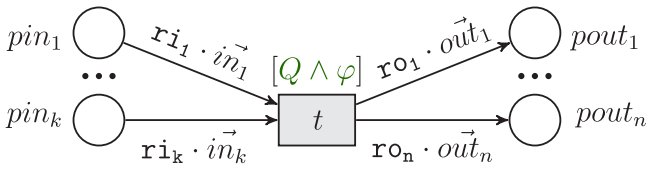


Fig. 8. A generic COA-net transition ($\texttt{ri}_j$ and $\texttt{ro}_j$ are natural numbers).

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 as follows (notice that the last declaration is required when using MCMT in the database-driven mode):

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

Here, without loss of generality we assume that n is the index of the last attribute of relation R_{n_r}.

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 $\texttt{color}(p) = \mathcal{D}_1 \times \cdots \times \mathcal{D}_k$ is going to be represented as a combination of arrays $p_1, \ldots, p_k$, where a special index type $\texttt{P}_{\texttt{ind}}$ (disjoint from all other types) with domain $\Delta_{\texttt{P}_{\texttt{ind}}}$ is used as the array index sort and $\mathcal{D}_1, \ldots, \mathcal{D}_k$ account for the respective target sorts of the arrays.[6] In MCMT, this is declared using local (array) variables as following:[7]

```
:local p_1 D1    ...    :local p_k Dk
```

Then, intuitively, we associate to the jth 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 following MCMT initialisation statement

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

Here, cnj represents a conjunction of atomic equations that, for ease of reading, we organised in blocks, where each $init\_p_i$ specifies for place $p_i \in P$ with $\texttt{color}(p_i) = \mathcal{D}_1 \times \cdots \times \mathcal{D}_k$ that it contains no tokens. This is done by explicitly "nullifying" all components 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. We come back to the second step of

[6] MCMT has only one index sort, but, as shown in [35], there is no loss of generality in doing that.

[7] :local declarations can be immediately followed by :global ones used for denoting global, non-array variables. We will utilise them later on when describing the initial marking encoding.

the initial marking encoding after having discussed the transition encoding.

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 end, we consider the generic, prototypical COA-net transition $t \in T$ depicted in Fig. 8. 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 $\texttt{guard}(t)$ is satisfied under the selected binding.

For simplicity, we assume here that $Q \in \mathtt{CQ}^{\neg}_{\mathfrak{D}}$ and it is in Prenex normal form. Whenever $Q \in \mathtt{UCQ}^{\neg}_{\mathfrak{D}}$, such that $Q = \bigvee_{i=1}^{n} Q_i$, with $Q_i \in \mathtt{CQ}^{\neg}_{\mathfrak{D}}$, one needs to create n-variants of transition t, and suitably modify input and output arcs (together with their inscriptions) as well as transition guards, each of which will contain only one Q_i.

In MCMT, t is captured with a transition statement consisting of a guard G and an update U as follows

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

Here every x (resp., y) represents an existentially quantified index variable corresponding to variables in the incoming inscriptions (resp., outgoing inscriptions), $\mathtt{K} = \sum_{j\in\{1,\dots,k\}} \mathtt{ri}_j$, $\mathtt{N} = \sum_{j\in\{1,\dots,n\}} \mathtt{ro}_j$ and 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 $k' = |\mathtt{color}(pin_i)|$, we can check this with a formula

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

Here, the first big conjunct is used to check that there are $\mathtt{ri}_i$ identical tokens available in the array-based representation of pin_i. Given that variables representing existentially quantified index variables are already defined, in MCMT this is encoded as conjunctions of atoms

```
(= pini_l[j1] pini_l[j2])
```

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,[8] (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

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

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 catalogue. 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, we denote the set of such variables as $\mathbf{s} = \{s_1, \dots, s_r\}$ 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 $R(id, A, B, E)$.[9] Then, for every relation R in Q we generate a formula

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

This formula guarantees that values provided by a constructed binding respect the aforementioned case for some index χ (that is one of the existentially quantified index variables from ψ_{pin_j} used to refer to variable s in $\vec{in}_j$ that, in turn, also appears in R) and identifier id.[10] In MCMT this is encoded as a conjunction of atoms

```
(= (R_Ai id) pinj_l[x])
```

where $\mathtt{i} = \pi(R, s)$, $\mathtt{l} = \pi(\vec{in}_j, s)$ and id is a special eevar variable (we discuss eevar variables in the next paragraph) with sort D_R_id. As in the previous case, all such formulae are appended to G.

We now incorporate the encoding of condition *(FC2)*. Every variable d of Q with $\mathtt{type}(d) = \mathtt{D}$ has to be declared in MCMT as an existential variable[11] as follows

```
:eevar d D
```

Notice that such variables appear in Q and are used for querying the catalogue. As argued in [15], whenever a querying over a relational database is required in an infinite-state setting, the management of such variables becomes one of the most crucial points. In the setting when array-based systems are used in the "formal back-end", in order to manage such d variables one needs to rely on suitable, advanced quantifier elimination techniques studied in [33,34,36,37].

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 d in φ has been substituted with a "reference" to a corresponding array pin_j that d uses as a value provider for its bindings. More specifically, every relation $R/n + 1$ that

[8] This is a technicality of MCMT, as explained in [35] since MCMT has only one index sort.

[9] For simplicity, we assume that each inscription $\vec{in}_j$ (resp. $\vec{out}_j$) in Fig. 8 does not have repeating variables.

[10] Notice that it suffices to select a single index variable χ, since other participating indexes are already inter-equated in ψ_{pin_j}.

[11] Existentially quantified variables are crucial for expressing queries and get eliminated in the backward reachability procedure used by MCMT whenever it verifies a property (see Section 4.2 for more details).

appears in Q as $R(id, d_1, \ldots, d_n)$ is be replaced by conjunction $id \neq \texttt{NULL\_D} \wedge f_{R,A_1}(id) = d_1 \wedge \ldots \wedge f_{R,A_n}(id) = d_n$, where $\texttt{D} = \texttt{type}(id)$. In MCMT, this is written as

```
(not (= id NULL_D)) expr_1 ... expr_n
```

In this case, it is required that `id` is 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 d_i. Let us assume that d_i has been declared before as

```
:eevar d_i D
```

If d_i appears in a corresponding incoming transition inscription, then $expr_i$ is defined in MCMT as

```
(= (R_Ai id) pin_j[x])(= d_i pin_j[x])
```

where `i`-th attribute of R coincides with the `j`-th variable in the inscription $F_{in}(pin, t)$. If d_i is a variable bound by an existential quantifier in Q, then $expr_i$ in MCMT is going to look as

```
(= (R_Ai id) d_i)
```

Finally, if d_i is a variable in an outgoing inscription used for propagating data from the catalogue (as discussed in condition *(1)*), then $expr_i$ is simply defined with the following statement

```
(= (R_Ai id) d_i)}
```

where `Di` is the type of d_i.

Variables in φ are substituted with their array counterparts. In particular, every variable $\vec{d} \in Vars(\varphi)$ is substituted with `pinj_i[x]`, where $\texttt{i} = \pi(in_j, d)$. Given that φ is represented as a conjunction of atoms, 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 each $\vec{j} \in \{1, \ldots, n\}$, and for every variable $\nu \in \Upsilon_{\mathfrak{D}} \cap (OutVars(t) \setminus Vars(out_j))$ previously declared using

```
:eevar nu D
```

and for arrays $p_1, \ldots, p_k$ with target sort `D`, consists of expression (for all p)

```
(not(=nu p_1[j]))...(not(=nu p_k[j]))
```

This encodes "local" freshness for ν-variables, which suffice for our goal. Notice that here `j` is a universally quantified index variable defined in the MCMT transition statement with `:var j`.

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 follows

```
:numcases NC
...
:case (= j i)
:val v_{1,i}
...
:val v_{k,i}
...
```

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 ith token to it by putting a value $v_{r,i}$ in ith place of every rth 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 $\texttt{type}(\texttt{id}) = \texttt{D\_Ri\_id}$. 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 `case (= ji)`, such statement stands for a simple assignment $a[i] := \texttt{v}$. For COA-nets, one also has to include a "default" case that essentially encodes a law of inertia for all the tokens that have not been consumed. This is done by repeating local variables in place of `v` in each `:val v` statement.[12] Global variables, instead, have same values in all the cases, including the default one.

Finally, let us come back to the initial marking encoding. A special MCMT transition is used to inject the initial marking into the MCMT array-based system. This MCMT transition populates the arrays representing places, all initialised as empty, with entries that correspond to the initial COA-net marking m_0.

This MCMT transition can be executed only if flag `init_fl`, denoting whether the initial marking assignment has taken place, is `TRUE`.[13] It works as follows:

```
:transition
:var i1,..., iM
:var j
:guard (=init_fl TRUE)
:numcases NCm0
...
:case (= j i)
:val v_{1,i}
...
:val v_{k,i}
...
:val FALSE
...
```

Note that the `init_fl` flag should be previously declared using the MCMT statement

```
:global  init_fl Bool
```

Same holds for the boolean constants `TRUE` and `FALSE`: they are declared using the respective statements

```
:smt (define TRUE ::Bool)
```

and

```
:smt (define FALSE ::Bool)
```

Then, the transition runs through `NCm0` cases. All the cases go over the indexes `i1`,..., `iM` that correspond to tokens that have to be added to places. More specifically, for every place $p \in P$ such that $m_0(p) \neq \emptyset^{\oplus}$ and $|\texttt{color}(p)| = k$, we add an ith token to it by putting constant $v_{r,i} \in C$ in ith place of every rth component array of p. Moreover, every case has to update `init_fl`, changing its value to `FALSE`.

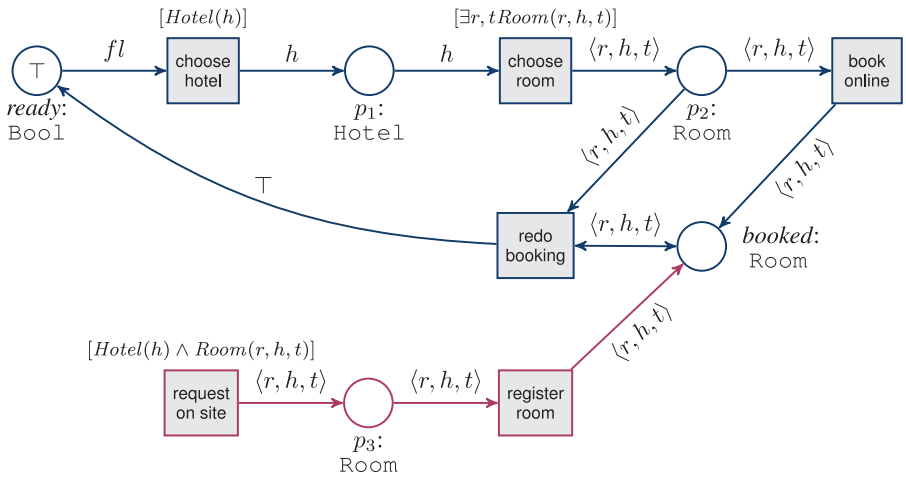


Fig. 9. A COA-net representing a hotel booking process with the initial marking containing only one token in place `ready`.

3.3. Encoding example

Currently, the encoding presented in the previous section does not have a prototype realising it. Thus, we demonstrate the feasibility of our approach by manually encoding a COA-net into MCMT [18].

Let us start with a COA-net N that represents a simple, faulty hotel booking scenario.[14] In Fig. 9, the net is split into two parts, one representing an on-line booking process followed by a customer and the other accounting for bookings done directly on site. In the first process, the customer consequently chooses a hotel and a room in it, and, if the chosen room is not available, can repeat the booking process. The second process instead plays a role of a "booking adversary": it allows to pick any room without verifying the room availability and can always be executed in parallel with the on-line booking process. Consequently, this makes N unable to prevent double bookings (since, as we have already mentioned in Section 2.1, there is no way to express universal quantification over places in COA-nets).

Both processes run on a catalogue composed of two relations $Hotel(h : \texttt{HName})$ and $Room(r : \texttt{RId}, h : \texttt{HName}, t : \texttt{String})$. The first relation stores information about all the hotels that can be potentially available for booking, whereas the second relation lists rooms (as well as their types) offered by such hotels.

We now illustrate a few MCMT code snippets acquired by manually translating the booking example. The translation process starts with defining all the static data components such as data types and relation signatures in MCMT:

```
:smt (define-type HName)
:smt (define-type RId)
:smt (define-type Bool)
:smt (define-type String)
:smt (define RoomHotel ::(-> RId HName))
:smt (define RoomType ::(-> RId String))
:smt (define TRUE ::Bool)
:smt (define FALSE ::Bool)
```

Here, `RoomHotel` and `RoomType` are functions used for the functional characterisation of the last two attributes of relation *Room*.

To make MCMT aware of the catalogue schema used in the example, we add the following block:

```
:db_driven
:db_sorts Hotel Room Bool String
:db_functions RoomType RoomHotel
:db_constants  TRUE FALSE
:db_relations
```

Now we need to encode the places of N. Observe that, while such places as p_3 and `booked` are clearly unbounded, `ready`, p_1 and p_2 will never carry more than one token. To this end, we demonstrate how bounded places can be treated in MCMT by simply resorting to globally declared variables (instead of arrays) for storing token components. Following the translation instructions from the previous section, one gets the following:

```
:local booked_1 RId
:local booked_2 HName
:local booked_3 String
:local p3_1 RId
:local p3_2 HName
:local p3_3 String
:global p1_1 HName
:global p2_1 Rid
:global p2_2 HName
:global p2_3 String
:global ready Bool
:global init_fl Bool
```

As mentioned above, all the variables needed for representing 1-bounded places are declared using the `:global` keyword. Given that place `ready` can be seen as a flag, we use `Bool`-typed variable `ready` to model it. Then, we write the initialisation statement, in which all the places are "nullified", together with all the `eevar` declarations needed later on for encoding transition preconditions:

```
:initial
:var x
:cnj (= init_fl TRUE) (= ready FALSE) (= p1_1
    NULL_HName)
     (= p2_1 NULL_RId) (= p2_2 NULL_HName) (= p2_3
         NULL_String)
     (= booked_1[x] NULL_RId) (= booked_2[x]
         NULL_HName)
     (= booked_3[x] NULL_String) (= p3_1[x]
         NULL_RId)
     (= p3_2[x] NULL_HName) (= p3_3[x] NULL_String)
:eevar e HName
:eevar d HName
:eevar f HName
:eevar g RId
:eevar l RId
:eevar m RId
:eevar n String
```

12 Notice that all local variables have to be indexed with `j`.

13 In case a transition is enriched with guard G, then G should contain a conjunct `(= init_fl FALSE)`.

14 As we will see later, this example encodes unwanted booking behaviour.

```
:eevar o String
:eevar p String
:eevar q Bool
```

These statements are immediately followed by an MCMT transition modelling the initial marking assignment:

```
:transition
:var j
:var x
:guard  (= init_fl TRUE) (= x x)
:numcases 2
:case (= x j)
:val booked_1[j]
:val booked_2[j]
:val booked_3[j]
:val p3_1[j]
:val p3_2[j]
:val p3_3[j]
:val p1_1
:val p2_1
:val p2_2
:val p2_3
:val TRUE
:val FALSE
:case
:val booked_1[j]
:val booked_2[j]
:val booked_3[j]
:val p3_1[j]
:val p3_2[j]
:val p3_3[j]
:val p1_1
:val p2_1
:val p2_2
:val p2_3
:val TRUE
:val FALSE
```

Let us now showcase a couple of the net transitions. For example, choose hotel is represented with the following code:

```
:transition
:var j
:guard  (= init_fl FALSE) (not (= e NULL_HName))
    (= ready TRUE)
:numcases 1
:case
:val booked_1[j]
:val booked_2[j]
:val booked_3[j]
:val p3_1[j]
:val p3_2[j]
:val p3_3[j]
:val e
:val p2_1
:val p2_2
:val p2_3
:val FALSE
:val init_fl
```

Notice that the update part here uses a single case as there is no need to run through array indexes of the updated places. The situation becomes different when we deal with unbounded places. For example, the book online transition is modelled as

```
:transition
:var j
:var x
:guard  (= init_fl FALSE) (not (= p2_1 NULL_RId))
   (not (= p2_2 NULL_HName)) (not (= p2_3
       NULL_String))
   (= booked_1[x] NULL_RId)   (= booked_2[x]
       NULL_HName)
   (= booked_3[x] NULL_String) (= p3_1[x] NULL_RId)
   (= p3_2[x] NULL_HName) (= p3_3[x] NULL_String)
:numcases 2
:case (= x j)
:val p2_1
:val p2_2
:val p2_3
:val p3_1[j]
:val p3_2[j]
:val p3_3[j]
:val p1_1
:val p2_1
:val p2_2
:val p2_3
:val ready
:val init_fl
:case
:val booked_1[j]
:val booked_2[j]
:val booked_3[j]
:val p3_1[j]
:val p3_2[j]
:val p3_3[j]
:val p1_1
:val p2_1
:val p2_2
:val p2_3
:val ready
:val init_fl
```

Here, following the new marking generation encoding described in Section 3.2, we model addition of a new token to place booked in `:case` (= x j) and preservation of all the "untouched" tokens in the (default) case below.

The interested reader can go over the complete example encoding here: https://tinyurl.com/5eyr59rm.

4. Parameterised unsafety checking and its formal properties

Thanks to the encoding of COA-nets into the *database-driven module* of MCMT, we can handle the parameterised verification of safety properties over COA-nets.

The purpose of this section is to formalise this verification problem, and to comment on its (meta-)properties, primarily considering algorithmic aspects arising from the encoding into MCMT such as soundness, completeness, and termination. We also discuss (un)decidability issues. In particular, we isolate an interesting class of COA-nets for which verification is decidable. At the same time, we consider data-aware Petri nets with the simplest form of correlation possible. We demonstrate that they are already Turing-powerful, in turn showing that undecidability is not caused by the presence of the catalogue nor by the sophisticated form of parameterised verification we consider.

4.1. Unsafety properties

We focus our attention on *safety properties*, that is, properties that have to globally hold in each state of the system. As customary, a safety property is verified in the converse, that is, by expressing a corresponding *unsafety property*, and by checking if there exists a reachable state of the system that satisfies it. If so, then the state and the sequence of transitions to reach that state from the initial one witness that the system is indeed not safe.

Definition 5. An *(unsafety) property* over COA-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 $[p(x_1, \ldots, x_n) \geq c]$. $\triangleleft$

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. Here, $x_1, \ldots, x_n$ act as a filter that selects the matching tokens in p. Additionally, the same variables may be used to

inspect different places as join operators, in turn expressing *co-reference* constraints on tokens present in the respective places. A property may also mention relations from the catalogue, 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 COA-net, possibly relating tokens with the content of the catalogue. 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 [28]).

Example 6. Consider the COA-net of Example 5, 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 is expressed by checking for the existence of two tokens that are respectively located in the *delivered* and *working* places, and that co-refer on the same order. ◁

Similarly to what shown in Section 3, also properties as of Definition 5 admit a direct encoding into the MCMT model checker, as we show next. We start by looking into single components of a property of the form $\exists\vec{x}.\psi$ and discuss how each component should be represented in MCMT. As mentioned in Definition 5, properties are quantifier-free queries that may contain relations in them proviso that such relations have their variables appearing in atomic place predicates. To this end, the translation of queries without place predicates is "subsumed" by the translation of extended guards presented in Section 3 (however, we are going to comment on its peculiarities later on).

We start by translating predicates. An atomic predicate $[p \geq c]$, where $\texttt{color}(p) = \mathcal{D}_1 \times \cdots \times \mathcal{D}_k$, is represented using the following MCMT statement:

```
(not (= p_1[z_1]  NULL_D1)) ... (not (= p_1[z_c]
    NULL_D1))
...
(not (= p_k[z_1]  NULL_Dk)) ... (not (= p_k[z_c]
    NULL_Dk))
```

Here, `z_i` are special existentially quantified index variables that can appear only in the property formula and do not need to be explicitly declared anywhere in the MCMT specification. The statement above formalises the fact that there are at least c tokens in place p (or, similarly, at least c tuples in arrays representing place p) such that each of them has all the components different from `NULL`.

Analogously, we deal with each atomic predicate of the form $[p(x_1, \ldots, x_n) \geq c]$ and translate it into the following MCMT statement:

```
(= p_1[z_1]  x_1)  ... (= p_n[z_1]  x_n)
...
(= p_1[z_c]  x_1)  ... (= p_n[z_c]  x_n)
```

This statement, as opposed to the statement formalising predicate $[p \geq c]$, does not need to check that token components are empty. Instead, it formalises the fact that there are at least c tokens in place p all identical to the token $(x_1, \ldots, x_n)$.

Let us now denote each predicate $[p_i \geq c_i]$ (resp., $[p_i(x_1, \ldots, x_n) \geq c_i]$) statement translation with `P_i` (resp., `P'_i`) and the total number of such translations as `m` (resp. `m'`). Then, the final translation of property $\exists\vec{x}.\psi$ is represented in MCMT as follows:

```
:u_cnj P_i1 P_i2 ... P_im P'_j1 P'_j2 P'_jm' G
```

Here, `G` is a statement containing the quantifier-free query translation, in which variables in every relation have to be substituted with suitable `p_j[z_l]` array components taken from some of the `P_i` and `P'_i` statements.

Example 7. Consider the COA-net from Section 3.3. It is easy to see that it may be possible to book a room that has already been booked or pre-registered (that is, a token with the room identifier has been placed in place `booked`). The property capturing this situation for the on-line component of the net can be written as $\exists r, h, t.[p_2(r, h, t) \geq 1] \wedge [\texttt{booked}(r, h, t) \geq 1]$. In MCMT, proviso that p_2 is 1-bounded and the predicate $[p_2(r, h, t) \geq 1]$ can be substituted with $[p_2(r, h, t) = 1]$, this property is specified as following:

```
:u_cnj (not (= p2_1 NULL_RId)) (not (= p2_2
    NULL_HName))
       (not (= p2_3 NULL_String)) (= p2_1
           booked_1[z1])
       (= p2_2 booked_2[z1]) (= p2_3 booked_3[z1])
```

Notice that this property encoding is optimised as it suffices to specify that the same token can be found both in p_2 and `booked`.

Similarly, for the on site part of the net the unsafety property is specified as $\exists r, h, t.[p_3(r, h, t) \geq 1] \wedge [\texttt{booked}(r, h, t) \geq 1]$. Its MCMT encoding is as follows:

```
:u_cnj (not (= p3_1[z1] NULL_RId)) (not (= p3_2[z1]
    NULL_HName))
       (not (= p3_3[z1] NULL_String)) (= p3_1[z1]
           booked_1[z2])
       (= p3_2[z1] booked_2[z2]) (= p3_3[z1]
           booked_3[z2])
```

4.2. Verification problem

To express the verification problem of interest in our setting, we follow the line of research extensively studied to formally analyse dynamic systems with read-only relational data [15,38], focusing our attention on (un)safety properties.

Technically, we frame our verification problem as checking whether it is true that *all* the reachable states of a marked COA-net satisfy a desired safety condition, *independently from the content of the catalogue*. This captures the fact that the system is *robustly* safe, in the sense that safety does not depend on a specific configuration of the read-only data. In the converse, we take an unsafety property defined as in Definition 5, and check whether there exists an instance of the catalogue such that the COA-net can evolve the initial marking to a state where that property holds. This is formally captured next.

Definition 6. Given a property ψ as in Definition 5, a marked COA-net $\langle N, m_0\rangle$ is *unsafe* w.r.t. ψ if there exists a catalogue instance *Cat* for N such that the marked fixed-catalogue COA-net $\langle N, m_0, Cat\rangle$ can reach a configuration where ψ holds. If this is not the case, then $\langle N, m_0\rangle$ is *safe* w.r.t. ψ. ◁

In the following, whenever we generically mention the term *verification*, we implicitly refer to the verification task of Definition 6.

Example 8. Consider again the COA-net of Example 5, and the property defined in Example 6. The COA-net is safe w.r.t. this property, as the property never holds in the executions of the COA-net, irrespectively of the content of the net catalogue. This is because an item can be delivered only if it has been previously loaded in a compatible truck; this is in turn possible only if the order to which the loaded item belongs is *paid*. ◁

Example 9. In Example 7, we have mentioned two properties that we conjecture to be unsafe. MCMT allows to check the corresponding COA-net against both of these properties at once: it suffices to mention them in the specification file and the tool will consider them as one disjunction. This disjunction is shown to be unsafe by MCMT in 0.26 s.[15] ◁

4.3. Soundness and completeness

We now consider the usage of MCMT to carry out the verification problem of Definition 6. With a slight abuse of terminology, we interchangeably use the term *COA-net* to denote the net under study or its MCMT encoding, and likewise for the term *property*.

Proofs of all results in this section are directly obtained by exploiting the close relationship between COA-nets and the foundational framework at the basis of the database-driven module of MCMT [15,26], implicitly established by the translation defined in Section 3.

Specifically, we consider the key (meta-)properties of soundness and completeness of the backward reachability procedure (briefly discussed in Section 3.1) encoded in MCMT for verifying COA-nets.[16] We call this procedure BREACH, and in our context we assume it takes as input a marked COA-net and an (undesired) property ψ, returning UNSAFE if there exists an instance of the catalogue so that the net can evolve from the initial marking to a configuration that satisfies ψ, and SAFE otherwise. Details on the procedure can be found in [15,19].

We formally characterise the (meta-)properties of BREACH as follows.

Definition 7. Given a marked COA-net $\langle N, m_0 \rangle$ and a property ψ, BREACH is:

- *sound* if, whenever it terminates, it produces a correct answer;
- *partially sound* if, whenever it returns SAFE, that output is always correct;
- *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. This is not surprising given the expressiveness of the framework and the type of parameterised verification tackled.

We start by studying BREACH on COA-nets in their full generality. As we have seen in Section 3, the encoding of fresh variables calls for a limited form of universal quantification. This feature goes beyond the features covered by the foundational framework underpinning (data-driven) MCMT [15], which in fact does not explicitly consider fresh data injection. It is known from previous works (see, e.g., [39]) that when transition formulae employ universal quantification over the indexes of an array, BREACH cannot guarantee that all the indexes are indeed considered. This can lead to potentially spurious situations in which some indexes are simply "disregarded" when exploring the state space. This spurious exploration of the state space, which is similar to what happens in lossy systems, may result in the wrong classification of a SAFE case as being UNSAFE. By combining [15,39], we then get the following result.

Theorem 1. BREACH *is partially sound and complete for marked COA-nets.* ◁

To mitigate the presence of potentially spurious results, MCMT is equipped with techniques for debugging the returned result [39]. In particular, MCMT explicitly returns, together with the unsafety verdict, a flag discriminating the case where the produced result is provably correct from the case where the result *may* have been mistakenly obtained due to a spurious exploration of the state space.

A key point is then how to tame partial soundness towards recovering full soundness and completeness. We obtain this for the two special classes of *conservative* and *bounded* COA-nets, described next.

4.4. Conservative COA-nets

Conservative COA-nets do not have the ability to inject fresh values into their tokens, but only to manipulate data objects mentioned in the initial marking or in the catalogue.

Definition 8. A COA-net is conservative if no ν-variable is used in its arc inscriptions. ◁

Conservative COA-nets form a subclass of the foundational framework in [15], and consequently enjoy all the properties established there. In particular, we inherit that BREACH is a semi-decision procedure.

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

One may wonder whether studying conservative nets is meaningful. We argue in favour of this by considering some modelling techniques to remove fresh variables from a net while preserving interesting properties.

Avoiding fresh variables. The first technique is to ensure that whenever there is an unnecessary usage of ν-variables, such ν-variables are removed. As we have extensively discussed at the end of Section 2, generating a fresh object and inserting it into a token is necessary only if that object can be subsequently used in other tokens as well, thus acting as a reference. This is the case if the object belongs to the "one" side of a many-to-one relationship (e.g., the case of an order that can contain many items), or whenever the object participates to a one-to-one relationship and belongs to the endpoint that has been chosen as reference.

Fixing objects. When the COA-net requires to capture references, ν-variables can still be removed if one limits the scope of verification, by considering a fixed set of "prototypical" objects of a given type instead of the more general case where such objects are created on-the-fly. This is for example what happens in soundness checking of workflow nets, where analysis is limited to a *single* case object and its evolution from the input to the output place of the net.

In the general case, moving from arbitrary creation to a fixed set of elements leads to an under-approximation of the original COA-net. In fact, unsafety carries from the modified to the original net, in the sense that if a property is unsafe for the modified net, then it is unsafe also for the original one. Contrariwise, a property may be judged as safe in the modified net while being unsafe on the original one.

Example 10. By inspecting the COA-net of Example 5, two observations are in place when it comes to the creation of tokens.

[15] The tool was run on a machine with macOS High Sierra 10.13.3, 2.3 GHz Intel Core i5 and 8 GB RAM.

[16] Recall that *Backward reachability* has nothing to do with *marking reachability*. In this work, we deal with reachability of a configuration that satisfies a property which, in turn, is implicitly achieved by covering a marking, which, in turn, assigns to places tokens that carry tuples of data (potentially retrieved from the read-only catalogue).

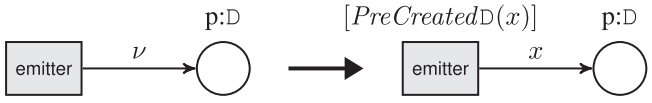


Fig. 10. Verification-preserving transformation of an emitter transition injecting fresh objects into a conservative, emitter transition that picks objects from a dedicated read-only catalogue relation.

- The creation of orders requires to use a ν-variable to generate fresh order identifiers, which later on are referenced by items so as to track which items belong to which orders.
- The creation of items does not need to appeal to ν-variables, since the only important aspect is to faithfully reconstruct the number of items that belong to the same order and share category, not to distinguish them. To do so, one can simply rely on the multiset semantics of Petri nets.

To remove the only ν-variable present in the net, we can remove the new order transition, and directly use the initial marking to insert one or more order tokens 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. ◁

Pre-creation of objects. A third technique to eliminate ν-variables is similar in spirit to the one just described, but yields a much more sophisticated setting. The idea is the remove ability of the net to create fresh objects, by assuming that all the objects of interest are created at the very beginning. Differently from the previous case, though, we do not explicitly insert these objects in the initial marking, but we assume instead that they are "pre-created" and listed in a dedicated, read-only catalogue relation, from which objects are picked and injected in the net. This is more powerful than focusing on a fixed set of objects, since now verification considers all possible configurations of this read-only relation, consequently checking safety where an arbitrary amount of objects is considered.

When the original net contains an emitter transition supporting the unconstrained creation of fresh objects of a given kind (such as in the case of orders for Example 5), we can employ this technique to turn the emitter transition into a conservative transition that preserves verification. The general idea on how to do so is shown in Fig. 10. The original net has an emitter transition that can at any point in time inject a fresh object of type D into a D-typed place p. The corresponding conservative net introduces a new, dedicated unary relation *PreCreated*D, storing the pre-created objects of type D, and modifies the original emitter transition by decorating it with a guard that fetches an object from that relation, and injects that object into p. This approach directly generalises to the case where tokens in place p contain tuples of objects is handled analogously.

We can intuitively see that this transformation preserves verification through the following line of reasoning. Since the original net supports arbitrary creation of objects, for every n there is a run of the net where exactly n objects are dynamically created and used. To check for unsafety, one such run has to be isolated to show that the system can reach an undesired configuration. Assume this run employs k objects generated through the emitter transition. When carrying out verification of the corresponding conservative net with pre-created objects, there must exist an instance of catalogue for which the same run as before is generated. Since relation *PreCreated*D is completely unconstrained, this is indeed possible, as there exists an instance of the catalogue that inserts exactly k objects in *PreCreated*D.

Example 11. Pre-creation of objects can be applied to the net of Example 5, transforming it into a conservative net for which BREACH is guaranteed to be sound and complete. ◁

4.5. Bounded COA-nets

An orthogonal approach with respect to the one studied in Section 4.4 is to analyse what happens if the COA-net of interest is *bounded*. Recall that, as defined in Section 2, a COA-net is bounded by b if every reachable marking does not assign more than b tokens to every place of the net. Infinitely many states can still be reached due to the fact that tokens may, along a run, carry infinitely many distinct objects.

In this case, we can straightforwardly "compile away" fresh variables by introducing a place that contains, in the initial marking, enough provision of predefined objects, consuming from that place whenever the net requires a fresh object. This effectively transforms the COA-net into a conservative one, and so Theorem 2 applies, leading to the following result.

Corollary 1. BREACH *is sound and complete for bounded, marked COA-nets.* ◁

In [14,15], decidability results are obtained when the schema of read-only relations is *acyclic*, that is, its foreign keys never form referential cycles where a relation directly or indirectly points to itself. In particular, by imposing acyclicity and boundedness, we can relate COA-nets to [15, Thm.4.2]. In fact, thanks to boundedness, the content of each place can be stored in a set of predefined variables (whose size would depend on the bound and the arity of the place). This, in turn, yields the following.

Theorem 3. *The verification problem defined in Definition 6 is decidable for acyclic, bounded, marked COA-nets.* ◁

Notice that this decidability result does not imply that BREACH terminates, as the algorithm and the encoding of COA-nets into data-driven MCMT are not specifically tailored to this particular case, and re-defining the encoding by using predefined variables would not be practical.

It is worth pointing out that Theorem 3 covers the case of OA-nets, that is, COA-nets without catalogue. Actually, since verifying OA-nets does not require to handle any form of parameterisation, one can relate bounded OA-nets to the notion of bounded, generic transition systems extensively studied in [40,41], yielding decidability of full model checking for a variety of first-order temporal logics.

Several modelling strategies can be adopted to turn an unbounded COA-net into a bounded one. We briefly illustrate here two, based on prior works.

The first strategy is about explicitly modelling resources that are responsible for the evolution of certain objects [42]. A bound on resources of a certain kind consequently bounds the number of such objects that can coexist in the same state.

The second strategy conceptually relates to multiplicities in data modelling. Whenever there are one-to-many relationships implicitly present in a net, one can ensure that every object participating to the "one" side relates to boundedly many objects on the "many" side by explicitly imposing an upper bound on the multiplicity attached to the "many" side [43].

We illustrate the application of such strategies in the context of our running example.

Example 12. The COA-net of Example 5 has two sources of unboundedness: the creation of orders, and the addition of items to working orders.

The creation of unboundedly many orders can be controlled by introducing a suitable resource place, for example a place containing idle managers, each of which gets responsibility over one and only one order at a time. We can then impose that each order is created only when there is an idle manager not working on any other order. More specifically, whenever an order is created and injected in the net, an idle manager is consumed and associated to the order. Whenever an order completely disappears from the net, its responsible manager is returned to the idle manager place. This makes the overall amount of orders unbounded over time, but bounded in each marking by the number of manager resources.

The addition of items to a working order can be bounded by imposing, conceptually, that each order cannot contain more than a maximum fixed number of items. ◁

4.6. Expressiveness of COA-nets and boundaries of decidability

After all the considerations done on soundness, completeness, and decidability, one may wonder whether the intrinsic difficulty in verifying COA-nets comes from the sophistication of the model and/or the fact that verification is parameterised w.r.t. the catalogue. In this section we show that, even by reducing the modelling capabilities to the acceptable minimum required to capture interesting object-centric processes, unbounded COA-nets remain Turing-powerful.

To do so, we consider OA-nets, where the catalogue is not present at all, and the very simple property of checking whether a designated place is nonempty (which is a particular case of covering).

This problem is decidable for the special case of OA-nets employing just a single type and unary places only. In fact, this setting coincides with that of ν-Petri nets [28], which have decidable coverability.

ν-Petri nets lack support of object reference, and in turn of one-to-many relationships among objects, since each token can only carry a single data object, not multiple ones. One may consequently wonder whether the decidability status changes by considering the minimal setting in which relationships are supported, namely OA-nets with no catalogue and equipped with binary places hosting tokens that carry pairs of data objects.

It turns out that OA-nets of this form are already Turing-powerful, and so make nonemptiness of places undecidable to check. The proof is a simplified version of the one described in [44], which is based on a reduction from the halting problem for Minsky 2-counter automata. A *Minsky n-counter automaton* is a tuple $\langle \Sigma, Q, q_I, n, \delta, F \rangle$, where:

- Σ is a finite *alphabet* of symbols;
- Q is a finite set of *locations*;
- $q_I \in Q$ is the *initial location*;
- n is the *number of counters*;
- $\delta \subseteq Q \times \Sigma \times L \times Q$ is a labelled transition relation capturing the *control configuration updates* of the automaton, where L is the *instruction set* $\{\texttt{inc}, \texttt{dec}, \texttt{ifz}\} \times \{1, \ldots, n\}$;
- $F \subseteq Q$ is the set of *accepting locations*.

It is well-known that, for a Minsky 2-counter automaton that starts from the initial location with both counters set to 0, checking whether an accepting location can be reached is undecidable.

The idea of the proof is to represent a counter using a set of tuples in a binary relation, and the three counter operations of increment, decrement, and test for zero as operations manipulating those tuples, following the intuition shown in Fig. 11.

Technically, we get the following result.

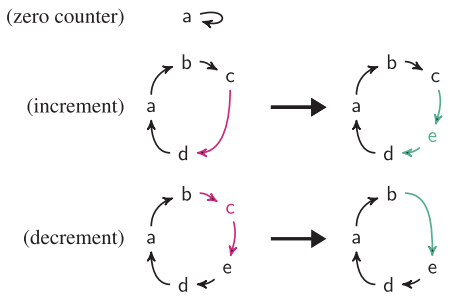


Fig. 11. Simulation of counter operations using rings constructed using a binary relation. A zero counter is represented as a self-loop; increment is simulated by introducing a new (fresh) element in ring, suitably updating the links so as to insert such a new element in some position of the ring; decrement is simulated by removing an element from the ring, suitably updating its attached links.

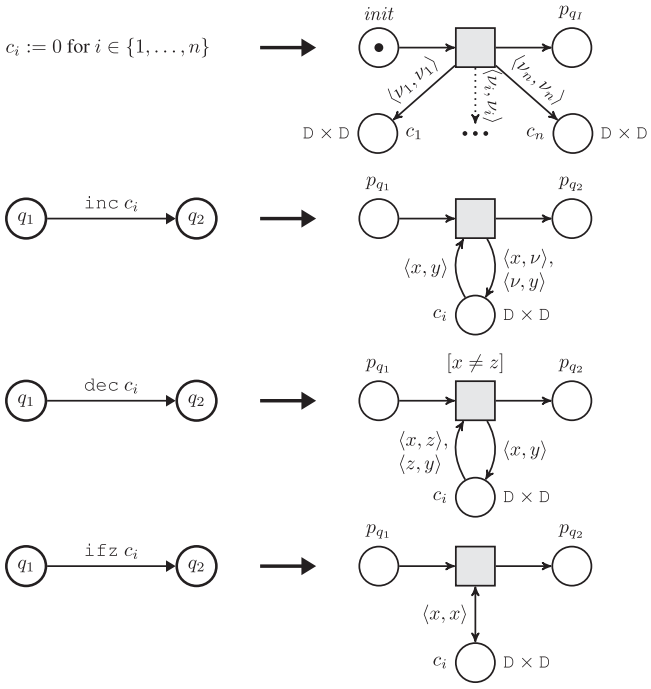


Fig. 12. Simulation of a Minksy counter machine via OA-nets.

Theorem 4. *Verifying place nonemptiness over OA-nets where only two places carry pairs of data objects, and the other places are 1-bounded, is undecidable.* ◁

Proof. Fig. 12 shows how OA-nets can simulate Minsky counter machines. Locations of the counter machines are represented as 1-bounded places carrying a black token, whereas counters are represented as places carrying pairs of data objects from some data domain D. Indeed, to represent a ring we use tokens that store *pairs* of consequent data objects in the ring (that is, the edges in Fig. 11). To form a proper ring, we need to guarantee some invariants on the data carried by these tokens. Specifically, a counter c with value n is represented by marking the corresponding counter place p_c with $n + 1$ tokens that carry pairs of data objects and that obey to the following two conditions:

(connectedness) Each data object appearing in the marking of p_c appears exactly twice in it, once in the first component of a token, and once in the second component of (possibly the same) token; this guarantees that each data object in the ring has exactly one successor and one predecessor.

(single ring) Every two data objects a and b appearing in the marking m of p_c are "connected", in the sense that there exists a multiset $m' \subseteq m(p_c)$ containing k elements (where $k \geq 1$) that form a chain $\langle \mathsf{d}_1, \mathsf{d}_2 \rangle \ \langle \mathsf{d}_2, \ldots \rangle \ \langle \ldots, \mathsf{d}_k \rangle \ \langle \mathsf{d}_k, \mathsf{d}_{k+1} \rangle$, such that $\mathsf{d}_1 = \mathsf{a}$ and $\mathsf{d}_{k+1} = \mathsf{b}$; this guarantees that the marking cannot contain multiple rings.

These two conditions imply that the marking of p_c employs exactly $n + 1$ distinct data objects when marking the place. We call a marking of this form an n-ring for p_c.

The initialisation of counter c to 0 is captured by injecting into p_c a token carrying the same data object in its first and second component (reconstructing the self-loop intuition of Fig. 11). This satisfies by construction the two ring-conditions above, resulting in a 0-ring for p_c.

We then proceed by induction. Assume that we have a counter place p_c marked with an n-ring. Increment for c in the Minsky machine is simulated by moving a black token according to the location update, and by manipulating the counter place as follows: one token carrying $\langle x, y \rangle$ is consumed from p_c, and two tokens respectively carrying $\langle x, \nu \rangle$ and $\langle \nu, y \rangle$ are inserted in p_c. Using ν guarantees that the inserted element does not match with any of the already existing elements in the ring. This produces a new marking that is an $n + 1$-ring for p_c (see the intuition in Fig. 11).

Decrement for c in the Minsky machine is simulated by moving a black token according to the update in location, ensuring that in that case the n-ring for p_c contains at least two data objects, that is, $n > 0$. This is done by consuming a token $\langle x, z \rangle$ with x different than z, and another token $\langle z, y \rangle$ that *joins* with the first one by using z as first component. Checking that x differs from z is needed to ensure that the transition cannot fire in the case where the counter is 0, which in turn means that the ring marking p_c is a self-loop where x matches z. This means that such an n-ring must have been obtained by previous increments, which in turn means that also y is guaranteed to be different from z. Decrement is then tackled by removing these two tokens, and injecting back a token that carries $\langle x, y \rangle$, thus conceptually directly linking x to y (see, again, the intuition in Fig. 11). This has the effect of removing the data object bound to z from p_c, making the new marking an $n - 1$-ring for p_c.

The proof concludes by noticing that:

- when the input machine has two counters, the resulting OA-net has the form mentioned in the statement of the theorem;
- reaching an accepting location in the input machine translates into checking that the corresponding OA-net can reach a marking where one of the corresponding accepting places carries a token, without imposing any constraint on the tokens stored in the counter places, which is indeed a coverability test. □

4.7. Addressing verification limitations

The translation provided in Section 3 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 means that, intuitively, in order to obtain a "functional" translation, one needs to use a net model in which each transition can manipulate at most two unbounded places and possibly arbitrary many bounded places, as the latter can be represented either as variables or arrays with already defined, constant indexes.

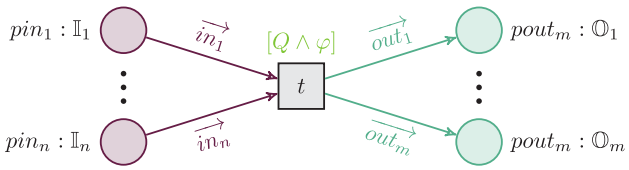


Fig. 13. A generic COA-net transition with one inscription per arc. Without loss of generality, we assume that variables in input inscriptions never repeat, and that the intended matches are expressed as equality conditions in the guard φ.

Notice that the aforementioned 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. In the following, we show how to overcome this limitation by transforming an arbitrary COA-net into a net for which an MCMT representation in which the supported number of index variables is not exceeded.

Theorem 5. *Let $\langle N, m_0 \rangle$ be a marked COA-net $N = (\mathfrak{D}, \mathcal{R}_{\mathfrak{D}}, P, T, F_{in}, F_{out}, \mathtt{color}, \mathtt{guard})$ with initial marking m_0 and $\exists \vec{y}.\psi(\vec{y})$ be a property over N. Then, there exists a COA-net $N' = (\mathfrak{D}', \mathcal{R}_{\mathfrak{D}}, P', T', F'_{in}, F'_{out}, \mathtt{color}', \mathtt{guard}')$ with initial marking m'_0 such that:*

(1) $|({}^\bullet t \cup t^\bullet) \cap P| \leq 2$, for every $t \in T'$;
(2) every $p \in (P' \setminus P)$ is 1-bounded;
(3) the result of BREACH *for $\exists \vec{y}.\psi(\vec{y})$ over $\langle N, m_0 \rangle$ coincides with the result of* BREACH *for $\exists \vec{y}.\psi(\vec{y}) \wedge \mathtt{LOCK} = \mathtt{TRUE}$ over $\langle N', m'_0 \rangle$.* ◁

Proof. We start by discussing the construction of COA-net N' following the intuition from above. Remember that for every transition in N there is one corresponding MCMT transition in its encoding. Thus, in order to ensure that every MCMT transition uses at most two existentially quantified variables, we need to limit the number of arrays that such transition can manipulate. This means that every transition in a net can manipulate at most two unbounded places. In the following, we show how to transform a generic transition into one that satisfies such properties.

Let us consider a generic transition t in Fig. 13. This transition can be represented as net N_t in Fig. 14 which, instead of simultaneously performing token consumption and production, performs these operations in a sequence by imposing an arbitrary order on places from ${}^\bullet t$ and $t^\bullet$. To insure that the non-interruptibility of this sequence, the net uses a special place called **lock** $\in P'$ (in red in Fig. 14) that is used akin a critical section flag and such that $m'_0(\mathbf{lock}) = \{\bullet\}$ (that is, **lock** is 1-bounded). The consumption cycle is modelled by sequentially merging small sub-nets consisting of transition get_i, and three places pin_i, $loaded_{i-1} \in {}^\bullet get_i$ and $loaded_i \in get_i^\bullet$, such that $\mathtt{color}pin_i = \mathbb{I}_i$ and $\mathtt{color}(loaded_j) = \mathtt{color}(pin_1) \times \cdots \times \mathtt{color}(pin_j)$. Here, $loaded_j$ is a buffer place that accumulates all tokens extracted by get transitions from all preceding places pin, by incrementally merging them into a single token. Like that, given the implicit ordering imposed on pin places, extracting content of a token taken from some pin_i from a token stored in $loaded_k$, where $k > j$, is rather straightforward.

It is important to notice that every buffer place is also 1-bounded. Indeed, for some $i > 1$, transition get_i by construction can extract only a single token from its input buffer $loaded_{i-1}$. Moreover, when dealing with get_1 – the first transition in the consumption sequence – we have ${}^\bullet get_1 = \{pin_1, \mathbf{lock}\}$, where **lock** can have at most one token and thus prevents get_1 from acting as an emitter.

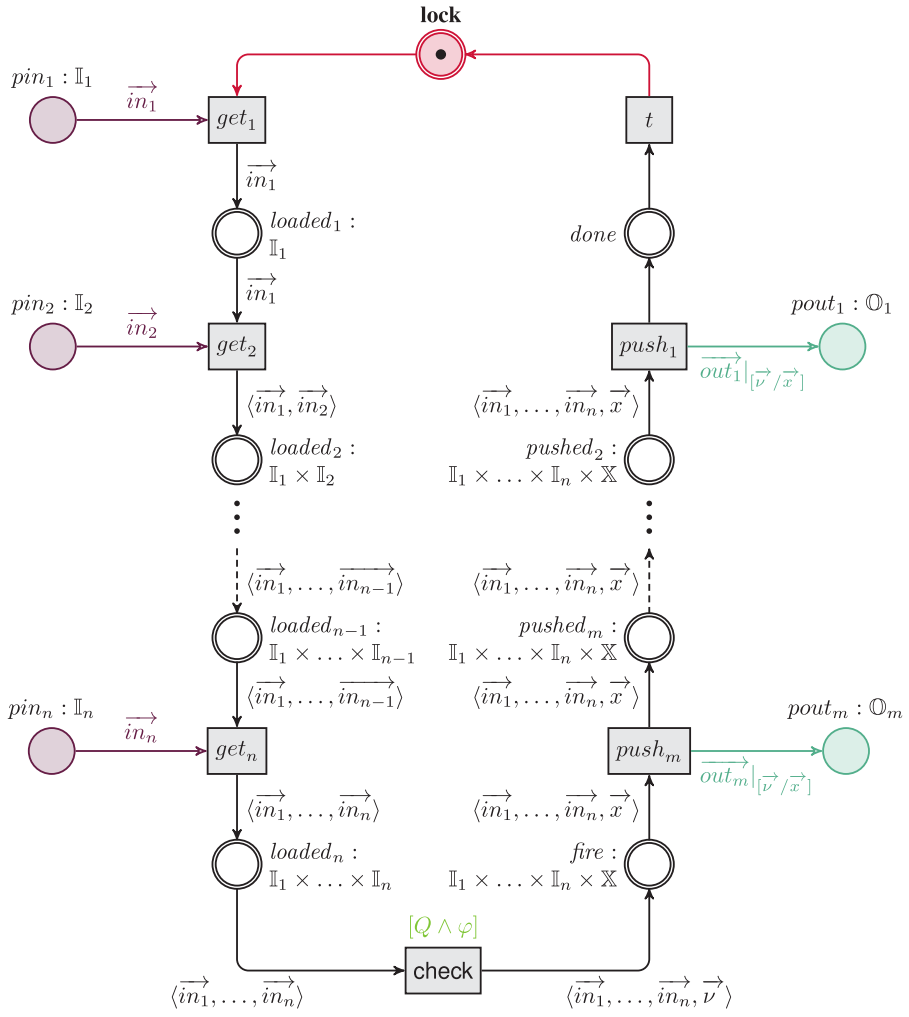


Fig. 14. Encoding of the generic COA-net transition of Fig. 13 into a corresponding subnet where each transition operates with at most one unbounded place, and possibly multiple 1-bounded places. 1-bounded places are shown as circles with double lines.

Before starting the token production sequence, N_t needs to verify whether the token stored $loaded_n$ satisfies $\texttt{guard}(t)$, and possibly extract data from the catalogue and generate fresh values for ν-variables of t. This is performed by transition *check*. If *check* is enabled, it produces a token in place *fire*, such that $\texttt{color}(fire) = \mathbb{I}_1 \times \cdots \times \mathbb{I}_n \times \mathbb{X}$, that binds to inscription $F_{out}(check, fire)$ as in Fig. 14, in which $\vec{\nu}$ is a tuple of length k containing all fresh variables appearing in the output inscriptions $\overrightarrow{out_1}, \ldots, \overrightarrow{out_m}$, and $\mathbb{X}$ denotes the colour type $X_1 \times \cdots \times X_k$, where $\texttt{type}(\nu_i) = X_i$, for each $i \in \{1, \ldots, k\}$. Then, to produce output tokens into places of $t^\bullet$, N_t uses a sequence of $push_i$ transitions, where ${}^\bullet push_i = \{pushed_{i+1}\}$ and $push_i^\bullet = \{pout_i, pushed_i\}$, for $i \in \{1, \ldots, m-1\}$.[17] This is done by extracting from a token stored in $pushed_{i+1}$ only content relevant to $pout_i$, by using inscription $\overrightarrow{out_i}|_{[\vec{\nu}/\vec{x}]}$, which indicates the inscription obtained from $\overrightarrow{out_i}$ by consistently replacing each fresh variable ν_i appearing in $\overrightarrow{out_i}$ and $\vec{\nu}$ with the corresponding variable x_i from $\vec{x}$. Here, $\vec{x}$ denotes a k-tuple of variables such that every variable x_i appearing therein has the same type of ν_i from $\vec{\nu}$, and does not appear in the original input and output inscriptions of the transitions. When the token production sequence cycle if finished, N_t is going to have only transition t enabled, which, upon firing, will release the lock by placing a token in **lock**.

Notice that, once embedded in bigger net N', internal names (with the exception of **lock**) have all to be relativised to concrete t (so that when encoding two distinct transitions, the two corresponding subnets are disjoint). Encoding transitions with multiple inscriptions per arc is done analogously (simply modularly operating over one inscription at a time). It is easy to see that N' satisfies conditions 1) and 2).

Now let us briefly discuss what happens when we want to check some property $\exists \vec{y}.\psi(\vec{y})$ on N'. By construction, N' may end up in a deadlock when *check* is not enabled, leaving a token in $loaded_m$. However, this situation does not influence the final result of BREACH which explores all possible executions of the net. Thus, if original transition t is having at least one binding that allows it to fire, it will be so as well for its representation in N_t (allowing to execute the whole token consumption–production sequence). Like that we can always regard all places in N_t (and, consequently, in N') that are different from those in ${}^\bullet t \cup t^\bullet$ as silent and only require that the backward search algorithm does not inspect the critical section, which can be easily achieved by adding to $\exists \vec{y}.\psi(\vec{y})$ a single conjunct ensuring that **lock** has a token. This can be easily encoded in MCMT by adding a boolean variable LOCK that should be always initialised with TRUE. Hence, answers returned by BREACH for $\exists \vec{y}.\psi(\vec{y})$ over $\langle N, m_0 \rangle$ and for $\exists \vec{y}.\psi(\vec{y}) \wedge \texttt{LOCK} = \texttt{TRUE}$ over $\langle N', m_0' \rangle$ will always be the same. □

We demonstrate the net encoding discussed in the theorem on a simple net from Fig. 15, which has one transition with

[17] For $push_m$, we have ${}^\bullet push_i = \{fire\}$.

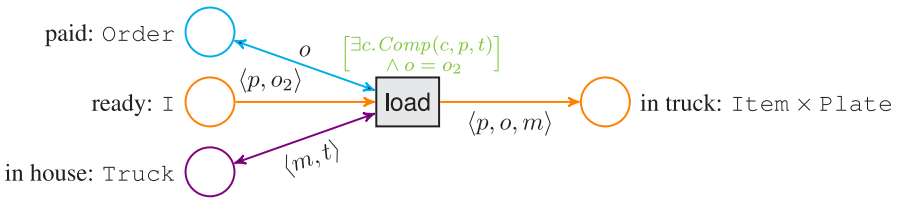


Fig. 15. A COA-net excerpt from Fig. 6.

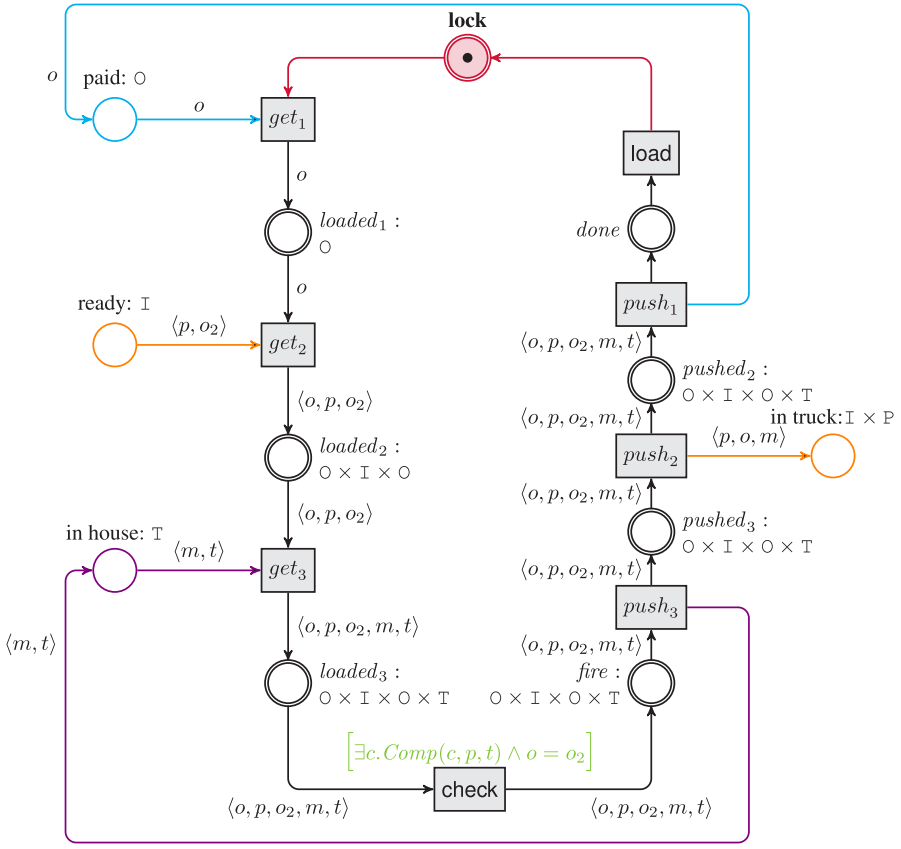


Fig. 16. An encoding of the net from Fig. 15 suing the approach presented in Theorem 5.

three input and three output places. This example is particularly interesting as it considers places that are used both as input and output. A net representing its encoding using the approach discussed in the theorem above is depicted in Fig. 16. Here, places paid and in house appear both in the token consumption and production sequences.

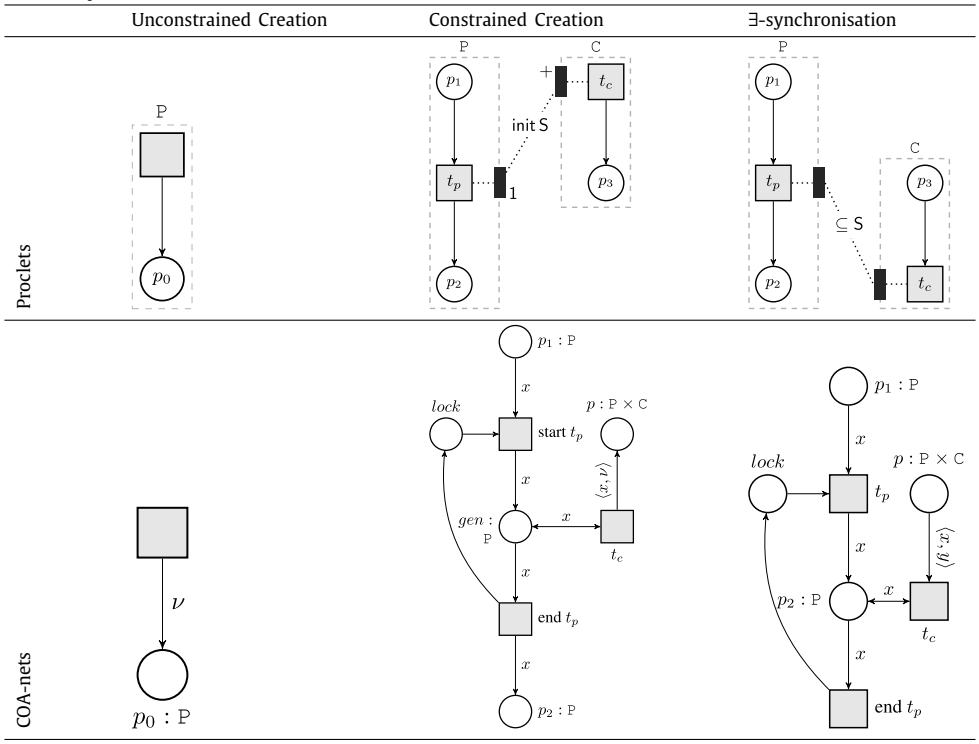
5. Comparison to other models

We comment on how the COA-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. COA-nets in their full generality match with an expressive fragment of the DB-net model [12]. 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 COA-nets, the catalogue 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, that involve solely relations that are never updated along the run of the net, have been relocated to transition guards of COA-nets. While COA-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*. More specifically, one can employ an approach presented in [27]. Using this one can define a set of additional relational places P_R, in which every p_i faithfully represents some $\mathfrak{D}$-typed relation schema R_i, a set of additional transitions T_{QR}, in which every transition is equipped with a guard that can access a content of some of the relational places and thus models a $\mathtt{UCQ}^{\neg}_{\mathfrak{D}}$ query. Moreover, such relations can be updated using a set of transitions T_{UR} using the standard token manipulation mechanism: whenever a fact has to be deleted (resp. added), a respective token is consumed (resp. added). Notice two things here. First of all, using this representation, COA-nets can update only a known number of tuples. Second, as opposed to the formalism used in [27], COA-nets cannot prioritise one enabled transition over another one. This means that, in the case of deletions, one has to update the relation places in a "lossy" manner by

Table 1
A table with three Proclet patterns and their corresponding representations in COA-nets. Here P and C respectively stand for parent and child Proclets.



introducing additional transitions that would also allow to skip the deletion (this would model cases in which a relation does not have a required tuple), whereas in the case of additions one needs to adopt the multiset semantics, allowing relation places to have potentially multiple instances of the same tuple.

While verification of DB-nets has only been studied in the bounded case, COA-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 potentially verifiable in practice.

PNIDs. The net component of our COA-nets model is equivalent to the formalism of Petri nets with identifiers (PNIDs [45]) without inhibitor arcs. Interestingly, PNIDs without inhibitor arcs form the formal basis of the *Information Systems Modelling Language* (ISML) defined in [13]. 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. COA-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. COA-nets can be seen as a sort of *explicit data* version of (a relevant fragment of) Proclets [10]. 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. 6, 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.[18] 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, COA-nets can encode correlation sets and the subset synchronisation semantics (see Table 1). 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 COA-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 COA-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

[18] In the remainder of this section, we assume that if the same creation step is activated multiple times by the same object o, then the whole set of objects created in such multiple activations are considered all part of the same correlation sets.

be captured in COA-nets by introducing a data-aware variant of wholeplace operation, which we aim to study in the future. We also briefly comment on the constrained creation in Proclets. As defined in [10], this type of creation happens uniquely for a correlation set (that is, two different creations would have two different correlation sets). However, in COA-nets it is possible to create objects asynchronously which can be later correlated (see pattern in Fig. 5).

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 systematically by showing a direct encoding into MCMT, one of the most well-established model checkers for the verification of infinite-state dynamic systems, and by applying this encoding to a simple example, for which unsafe configurations were tested by MCMT in fractions of a second. We have also shown that this model directly relates to some of the most sophisticated 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. It would be also interesting to study the impact of introducing wholeplace operations, essential to capture the most sophisticated synchronisation semantics defined for Proclets [10]. In this paper, we tried to discuss how certain formal restrictions imposed on the general formal model of COA-nets can be interpreted conceptually. In the future, it would be important to perform a further systematic investigation on modelling capabilities of COA-net subclasses for real examples. We are currently defining a benchmark for data-aware processes, translating the artefact systems benchmark defined in [32] into corresponding imperative data-aware formalisms, including COA-nets. To facilitate the translation process, we plan on implementing a tool that would allow to perform it automatically. However, we already know that this would require a preliminary study on additional heuristics improving MCMT's performance for imperative models.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research has been partially supported by the UNIBZ projects *VERBA*, *DUB*, *WineID* and *IDEE*.

References

[1] M. Reichert, Process and data: Two sides of the same coin?, in: Proc. of OTM, 2012, pp. 2–19.
[2] R. De Masellis, C. Di Francescomarino, C. Ghidini, M. Montali, S. Tessaris, Add data into business process verification: Bridging the gap between theory and practice, in: S.P. Singh, S. Markovitch (Eds.), Proc. of AAAI, AAAI Press, 2017, pp. 1091–1099.
[3] K. Batoulis, S. Haarmann, M. Weske, Various notions of soundness for decision-aware business processes, in: Proc. of ER, in: LNCS, vol. 10650, Springer, 2017, pp. 403–418.
[4] M. Dumas, On the convergence of data and process engineering, in: Proc. of ABDIS, in: LNCS, vol. 6909, Springer, 2011, pp. 19–26.
[5] R. Hull, Artifact-centric business process models: Brief survey of research results and challenges, in: Proc. of ODBASE, 2008, pp. 1152–1163.
[6] D. Calvanese, G. De Giacomo, M. Montali, Foundations of data aware process analysis: A database theory perspective, in: Proc. of PODS, ACM, 2013, pp. 1–12.
[7] V. Künzle, B. Weber, M. Reichert, Object-aware business processes: Fundamental requirements and their support in existing approaches, Int. J. Inf. Syst. Model. Des. 2 (2) (2011) 19–46.
[8] A. Meyer, L. Pufahl, D. Fahland, M. Weske, Modeling and enacting complex data dependencies in business processes, in: Proc. BPM, in: LNCS, vol. 8094, Springer, 2013, pp. 171–186.
[9] W.M.P. van der Aalst, Object-centric process mining: Dealing with divergence and convergence in event data, in: Proc. of SEFM, in: LNCS, vol. 11724, Springer, 2019, pp. 3–25.
[10] D. Fahland, Describing behavior of processes with many-to-many interactions, in: Proc. of Petri Nets 2019, Springer, 2019, pp. 3–24.
[11] A. Artale, A. Kovtunova, M. Montali, W.M.P. van der Aalst, Modeling and reasoning over declarative data-aware processes with object-centric behavioral constraints, in: Proc. of BPM, in: LNCS, vol. 11675, Springer, 2019, pp. 139–156.
[12] M. Montali, A. Rivkin, DB-Nets: on the marriage of colored Petri nets and relational databases, Trans. Petri Nets Other Model. Concurr. 28 (4) (2017).
[13] A. Polyvyanyy, J.M.E.M. van der Werf, S. Overbeek, R. Brouwers, Information systems modeling: Language, verification, and tool support, in: Proc. of CAiSE 2019, in: LNCS, vol. 11483, Springer, 2019, pp. 194–212.
[14] A. Deutsch, Y. Li, V. Vianu, Verification of hierarchical artifact systems, in: Proc. of PODS, ACM, 2016, pp. 179–194.
[15] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, SMT-based verification of data-aware processes: a model-theoretic approach, Math. Struct. Comput. Sci. 30 (3) (2020) 271–313.
[16] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Formal modeling and SMT-based parameterized verification of data-aware BPMN, in: Proc. of BPM, in: LNCS, vol. 11675, Springer, 2019, pp. 157–175.
[17] S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Petri nets with parameterised data - modelling and verification, in: D. Fahland, C. Ghidini, J. Becker, M. Dumas (Eds.), Proc. of BPM, in: LNCS, vol. 12168, Springer, 2020, pp. 55–74.
[18] MCMT: Model Checker Modulo Theories, http://users.mat.unimi.it/users/ghilardi/mcmt/. (Accessed: 25 August 2021).
[19] S. Ghilardi, S. Ranise, Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis, Log. Methods Comput. Sci. 6 (4) (2010).
[20] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, G.P. Rossi, Automated support for the design and validation of fault tolerant parameterized systems: a case study, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 35 (2010).
[21] R. Bruttomesso, A. Carioni, S. Ghilardi, S. Ranise, Automated analysis of parametric timing-based mutual exclusion algorithms, in: A. Goodloe, S. Person (Eds.), NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings, in: Lecture Notes in Computer Science, vol. 7226, Springer, 2012, pp. 279–294.
[22] F. Alberti, S. Ghilardi, N. Sharygina, A framework for the verification of parameterized infinite-state systems, Fundam. Inform. 150 (1) (2017) 1–24.
[23] P. Felli, A. Gianola, M. Montali, A SMT-based implementation for safety checking of parameterized multi-agent systems, in: Proc. of PRIMA, in: LNCS, vol. 12568, Springer, 2020, pp. 259–280.
[24] P. Felli, A. Gianola, M. Montali, SMT-based safety checking of parameterized multi-agent systems, in: Proc. of AAAI, Vol. 35, AAAI Press, 2021, pp. 6321–6330.
[25] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Verification of data-aware processes via array-based systems (extended version), arXiv.org, 2018, arXiv:1806.11459.
[26] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, From model completeness to verification of data aware processes, in: Description Logic, Theory Combination, and All that, in: LNCS, vol. 11560, Springer, 2019, pp. 212–239.
[27] M. Montali, A. Rivkin, From DB-nets to coloured Petri nets with priorities, in: Proc. of PN, 2019, pp. 449–469.
[28] F. Rosa-Velardo, D. de Frutos-Escrig, Decidability and complexity of Petri nets with unordered data, Theoret. Comput. Sci. 412 (34) (2011) 4439–4451.
[29] F. Rosa-Velardo, D. de Frutos-Escrig, Name creation vs. Replication in Petri net systems, Fund. Inform. 88 (3) (2008) 329–356.
[30] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, Workflow Patterns: The Definitive Guide, MIT Press, ISBN: 9780262029827, 2016.
[31] W.M.P. van der Aalst, C. Stahl, M. Westergaard, Strategies for modeling complex processes using colored Petri nets, Trans. Petri Nets Other Model. Concurr. 7 (2013) 6–55, http://dx.doi.org/10.1007/978-3-642-38143-0_2.
[32] Y. Li, A. Deutsch, V. Vianu, VERIFAS: A practical verifier for artifact systems, PVLDB 11 (3) (2017) 283–296.
[33] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Model completeness, covers and superposition, in: Proc. of CADE, in: LNCS (LNAI), vol. 11716, Springer, 2019, pp. 142–160, http://dx.doi.org/10.1007/978-3-030-29436-6_9.

[34] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Model completeness, uniform interpolants and superposition calculus, J. Autom. Reasoning 65 (7) (2021) 941–969, http://dx.doi.org/10.1007/s10817-021-09596-x.
[35] S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Petri Nets with Parameterised Data: Modelling and Verification (Extended Version), Technical Report, 2020, arXiv:2006.06630, arXiv.org.
[36] S. Ghilardi, A. Gianola, D. Kapur, Uniform interpolants in EUF: Algorithms using DAG-representations, Log. Methods Comput. Sci. (2022) in press.
[37] S. Ghilardi, A. Gianola, Interpolation and uniform interpolation in quantifier-free fragments of combined first-order theories, Mathematics 10 (3) (2022) http://dx.doi.org/10.3390/math10030461.
[38] A. Deutsch, R. Hull, Y. Li, V. Vianu, Automatic verification of database-centric systems, ACM SIGLOG News 5 (2) (2018) 37–56.
[39] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, G.P. Rossi, Universal guards, relativization of quantifiers, and failure models in model checking modulo theories, J. Satisf. Boolean Model. Comput. 8 (1/2) (2012) 29–61.
[40] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, M. Montali, Verification of relational data-centric dynamic systems with external services, in: Proc. of PODS, ACM, 2013, pp. 163–174.
[41] D. Calvanese, G.D. Giacomo, M. Montali, F. Patrizi, First-order (μ)-calculus over generic transition systems and applications to the situation calculus, Inform. and Comput. 259 (3) (2018) 328–347.
[42] M. Montali, A. Rivkin, Model checking Petri nets with names using data-centric dynamic systems, Form. Asp. Comput. (2016) 1–27.
[43] M. Montali, D. Calvanese, Soundness of data-aware, case-centric processes, Int. J. Softw. Tools Technol. Transf. (2016).
[44] S. Lasota, Decidability border for Petri nets with data: WQO dichotomy conjecture, in: F. Kordon, D. Moldt (Eds.), Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, ToruŃ, Poland, June 19-24, 2016. Proceedings, in: LNCS, vol. 9698, Springer, 2016, pp. 20–36.
[45] K.M. van Hee, N. Sidorova, M. Voorhoeve, J.M.E.M. van der Werf, Generation of database transactions with Petri nets, Fund. Inform. 93 (1–3) (2009) 171–184.