From DB-nets to Coloured Petri Nets with Priorities

Marco Montali and Andrey Rivkin(✉)

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

Abstract. The recently introduced formalism of DB-nets has brought in a new conceptual way of modelling complex dynamic systems that equally account for the process and data dimensions, considering local data as well as persistent, transactional data. DB-nets combine a coloured variant of Petri nets with name creation and management (which we call ν-CPN), with a relational database. The integration of these two components is realized by equipping the net with special "view" places that query the database and expose the resulting answers to the net, with actions that allow transitions to update the content of the database, and with special arcs capturing compensation in case of transaction failure. In this work, we study whether this sophisticated model can be encoded back into ν-CPNs. In particular, we show that the meaningful fragment of DB-nets where database queries are expressed using unions of conjunctive queries with inequalities can be faithfully encoded into ν-CPNs with transition priorities. This allows us to directly exploit state-of-the-art technologies such as CPN Tools to simulate and analyse this relevant class of DB-nets.

1 Introduction

During the last decade, the Business Process Management (BPM) community has gradually lifted its attention from process models mainly focusing on the flow of activities to multi-perspective models that also account for the interplay between the process and the data perspective [3,6,15]. Interestingly, the problem of modelling workflows dealing with database or document management systems has a long tradition in the Petri net field [5,14,19]. However, the resulting frameworks do not come with a clear separation of concerns between control flow and data-related aspects, nor with corresponding results on formal analysis. Recent variants of high-level Petri nets have been proposed to tackle these two challenges (see, e.g., [8,11,12,18]).

In this spectrum, the recently introduced formalism of DB-nets [12] has brought in a new conceptual way of modelling complex dynamic systems that equally account for the process and data dimensions, considering local data as well as persistent, transactional data. On the one hand, a DB-net adopts a standard relational database with constraints to store persistent data. The database can be queried through SQL/first-order queries, and updated via actions in a

S. Donatelli and S. Haar (Eds.): PETRI NETS 2019, LNCS 11522, pp. 449–469, 2019.
https://doi.org/10.1007/978-3-030-21571-2_24

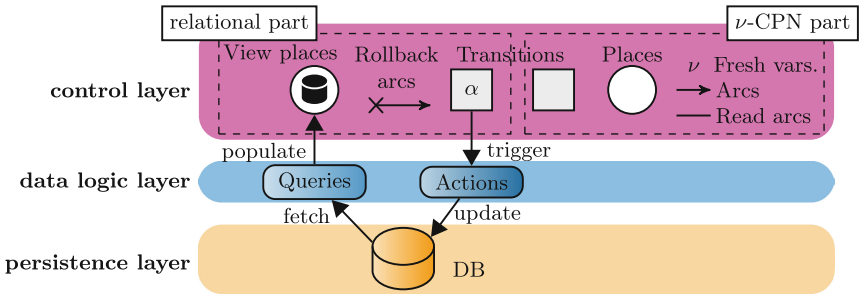


Fig. 1. The conceptual components of DB-nets

transactional way (that is, committing the update only if the resulting database satisfies all intended constraints). On the other hand, a DB-net employs a coloured variant of a Petri net with name creation and management [17] to capture the process control-flow, the injection of (possibly fresh) data such as the creation of new case identifiers [11], and tuples of typed data locally carried out by tokens. This model, which we call ν-CPN, can be seen as a fragment of standard Coloured Petri nets [7] with pattern matching on inscriptions, infinite colour domains, boolean guards, and a very limited use of SML to account for fresh data injection. This also means that ν-CPNs can be seamlessly modelled, simulated, and analysed using state-of-the-art tools such as CPN Tools.

The integration of these two components is realized in a DB-net by extending the ν-CPN with three novel constructs: *(i) view places*, special places that query the database and expose the resulting answers as coloured tokens that can be inspected but not directly consumed; *(ii) action bindings*, linking transitions to database updates by mapping inscription variables to action parameters; *(iii) rollback transition-place arcs*, capturing the emission of tokens in case a fired transition induces a failing database update, and in turn supporting the enablement of compensation transitions. All conceptual components used in the DB-net model are depicted in Fig. 1. Notably, DB-nets have been employed to formalize application integration patterns [16].

In this work, we study whether this sophisticated model can be *encoded back into ν-CPNs*, with a twofold intention. On the foundational side, we aim at understanding whether the process-data integration realized in DB-nets adds expressiveness to ν-CPNs, or it is instead conceptual, syntactic sugar. On the practical side, the existence of an encoding would allow us to directly exploit state-of-the-art tools such as CPN Tools towards simulation and analysis of DB-nets. In the case of CPN Tools, this is the only way possible when it comes to state space construction, given the fact that this feature cannot be refined through the third-party extension mechanism offered by the framework.

Specifically, we constructively show through a behavior-preserving translation mechanism that this encoding is indeed possible for a large and meaningful class of DB-nets, provided that the obtained ν-CPN is equipped with *transition priorities* [20] (a feature that is supported by virtually all CPN frameworks,

including CPN Tools). Such class corresponds to DB-nets where the database is equipped with key, foreign key, and domain constraints, and where the view places query the database using unions of conjunctive queries (UCQs) with inequalities. Such query language corresponds to the widely adopted fragment of SQL consisting of select-project-join queries with filters [1].

2 The DB-net Formal Model

In this section, we briefly present the key concepts and notions used for defining DB-nets. Conceptually, a DB-net is composed of three layers (cf. Fig. 1) (1) *persistence layer*, capturing a full-fledged relational database with constraints, and used to store background data, and data that are persistent across cases; (2) *control layer*, employing a variant of CPNs to capture the process control-flow, case data, and possibly the resources involved in the process execution; (3) *data logic layer*, interconnecting in the persistence and the control layer.

Definition 1. *A* db-net *is a tuple* $\langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N} \rangle$*, where: (i)* $\mathfrak{D}$ *is a type domain; (ii)* $\mathcal{P}$ *is a* $\mathfrak{D}$*-typed persistence layer; (iii)* $\mathcal{L}$ *is a* $\mathfrak{D}$*-typed data logic layer over* $\mathcal{P}$*; (iv)* $\mathcal{N}$ *is a* $\mathfrak{D}$*-typed control layer over* $\mathcal{L}$*.*

We next formalize the framework layer by layer.

Persistence Layer. A *type domain* $\mathfrak{D}$ is a finite set of pairwise disjoint *data types* $\mathcal{D} = \langle \Delta_{\mathcal{D}}, \Gamma_{\mathcal{D}} \rangle$, where a set $\Delta_{\mathcal{D}}$ is a *value domain*, and $\Gamma_{\mathcal{D}}$ is a finite set of *predicate symbols*. Examples of data types are: *(i)* $\mathbf{string} = \langle \mathbb{S}, \{=_s\} \rangle$, strings with the equality predicate; *(ii)* $\mathbf{real} = \langle \mathbb{R}, \{=_r, <_r\} \rangle$, reals with the usual comparison operators; *(iii)* $\mathbf{int} = \langle \mathbb{Z}, \{=_{int}, <_{int}, succ\} \rangle$, integers with the usual comparison operators, as well as the successor predicate.

We call $R(\mathcal{D}_1, \ldots, \mathcal{D}_n)$ a $\mathfrak{D}$*-typed relation schema*, where R is a relation name and $\mathcal{D}_i$ indicates the data type associated to an i-th component of R. A $\mathfrak{D}$*-typed database schema* $\mathcal{R}$ is a finite set of $\mathfrak{D}$-typed relation schemas. A $\mathfrak{D}$*-typed database instance* $\mathcal{I}$ *over* $\mathcal{R}$ is a finite set of facts of the form $R(\mathtt{o_1}, \ldots, \mathtt{o_n})$, such that $R(\mathcal{D}_1, \ldots, \mathcal{D}_n) \in \mathcal{R}$ and $\mathtt{o_i} \in \Delta_{\mathcal{D}_i}$, for $i \in \{1, \ldots, n\}$. Given a type $\mathcal{D} \in \mathfrak{D}$, the $\mathcal{D}$*-active domain of* $\mathcal{I}$, is the set of $Adom_{\mathcal{D}}(\mathcal{I}) = \{\mathtt{o} \in \Delta_{\mathcal{D}} \mid \mathtt{o} \text{ occurs in } \mathcal{I}\}$.

Given $\mathfrak{D}$, we fix a countably infinite set $\mathcal{V}_{\mathfrak{D}}$ of typed variables with a *variable typing function* $\mathtt{type} : \mathcal{V}_{\mathfrak{D}} \to \mathfrak{D}$. As query language, we adopt first-order (FO) logic extended with data types under the active-domain semantics [9], that is, the evaluation of quantifiers only depends on the values explicitly appearing in the database instance over which they are applied. This can be seen as the FO representation of SQL queries. A *(well-typed)* $\mathtt{FO}(\mathfrak{D})$ *query* Q over a $\mathfrak{D}$-typed database schema $\mathcal{R}$ has the form $\{\vec{x} \mid \varphi(\vec{x})\}$, where $\vec{x}$ is the tuple of answer variables of Q, and φ is a FO formula, with $\vec{x}$ as free variables, over predicates in $\cup_{\mathcal{D} \in \mathfrak{D}} \Gamma_{\mathcal{D}}$ and relation schemas in $\mathcal{R}$, whose variables and constants are correctly typed. We use $Q(\vec{x})$ to make the answer variables $\vec{x}$ of Q explicit, and denote the set of such variables as $Free(Q)$. When $Free(Q) = \emptyset$, we call Q a *boolean query*.

A *substitution* for a set $X = \{x_1, \ldots, x_n\}$ of typed variables, is a function $\theta : X \to \Delta_{\mathfrak{D}}$, such that $\theta(x) \in \Delta_{\mathtt{type}(x)}$, for every $x \in X$. A *substitution* θ *for*

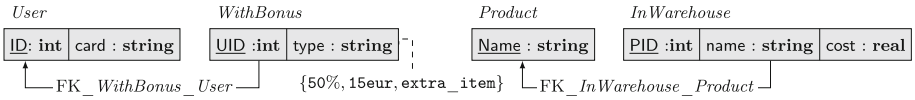


Fig. 2. The persistence layer for the online shopping scenario. UID and PID respectively represent unique user and product identifiers.

a $\texttt{FO}(\mathfrak{D})$ *query* Q is a substitution for the free variables of Q. We denote by $Q\theta$ the boolean query obtained from Q by replacing each occurrence of a free variable $x \in \mathit{Free}(Q)$ with the value $\theta(x)$. Given a $\mathfrak{D}$-typed database schema $\mathcal{R}$, a $\mathfrak{D}$-typed instance $\mathcal{I}$ over $\mathcal{R}$, and a $\texttt{FO}(\mathfrak{D})$ query Q over $\mathcal{R}$, the set of *answers* to Q in $\mathcal{I}$ is defined as the set $\mathit{ans}(Q, \mathcal{I}) = \{\theta : \mathit{Free}(Q) \to \mathit{Adom}_{\mathfrak{D}}(\mathcal{I}) \mid \mathcal{I}, \theta \models Q\}$ of substitutions for Q, where $\models$ denotes standard FO entailment (i.e., we use *active-domain semantics*). We denote by $\textsc{live}_{\mathcal{D}}(x)$ the unary query returning all the objects of type $\mathcal{D}$ that occur in the active domain (writing such a query is straightforward). When Q is boolean, we write $\mathit{ans}(Q, \mathcal{I}) \equiv \mathsf{true}$ if $\mathit{ans}(Q, \mathcal{I})$ consists only of the empty substitution (denoted $\langle\rangle$), and $\mathit{ans}(Q, \mathcal{I}) \equiv \mathsf{false}$ if $\mathit{ans}(Q, \mathcal{I}) = \emptyset$. Boolean queries are also used to express *constraints* over $\mathcal{R}$. We introduce explicitly two common types of constraints: given relations R/n and S/m, and two index-sets N and M such that $1 \leq i \leq n$ for every $i \in N$, and $1 \leq j \leq m$ for every $j \in M$, we fix the following notation: *(i)* $\textsc{pk}(R) = N$ expresses that the projection $R[N]$ of R on N is a primary key for R; *(ii)* $R[N] \subseteq S[M]$ expresses that the projection $R[N]$ of R on N refers the projection $S[M]$ of S on M, which has to be a key for S. Both kinds of constraints are obviously expressible as suitable queries [1].

Definition 2. *A* $\mathfrak{D}$*-typed* persistence layer $\mathcal{P}$ *is a pair* $\langle \mathcal{R}, \mathcal{E} \rangle$ *where: (i)* $\mathcal{R}$ *is a* $\mathfrak{D}$*-typed database schema; (ii)* $\mathcal{E}$ *is a finite set* $\{\Phi_1, ..., \Phi_k\}$ *of boolean* $\texttt{FO}(\mathfrak{D})$ *queries over* $\mathcal{R}$*, modelling* constraints over $\mathcal{R}$.

We say that a $\mathfrak{D}$-typed database instance $\mathcal{I}$ *complies with* $\mathcal{P}$, if $\mathcal{I}$ is defined over $\mathcal{R}$ and satisfies all constraints in $\mathcal{E}$.

Example 1. Let us consider a simplified shopping process used by an e-commerce website. Specifically, we are interested in a simplified scenario in which an already registered user logs in the website and immediately proceeds with selecting products. While products can be selected and added to the shopping cart, the user can occasionally choose a monthly bonus that may be applied when concluding a purchase. We restrict this scenario only by considering cases in which each user ends up buying at least one product.

The persistence layer $\mathcal{P} = \langle \mathcal{R}, \mathcal{E} \rangle$ of this scenario comprises four relation schemas (cf. Fig. 2): $\mathit{User}(\mathbf{int}, \mathbf{string})$ lists registered users together with their credit card data, $\mathit{WithBonus}(\mathbf{int}, \mathbf{string})$ indicates users that have bonuses, $\mathit{Product}(\mathbf{string})$ indexes product types offered by the website and $\mathit{InWarehouse}(\mathbf{int}, \mathbf{string}, \mathbf{real})$ indicates which products are stored in the

warehouse and with which cost. Note the constraints between these schemas. For example, in order to show that users cannot have more than one bonus at a time, we introduce a foreign key constraint between *WithBonus* and *User* that is denoted as $\mathit{WithBonus}[\{1\}] \subseteq \mathit{User}[\{1\}]$ and formalized in FO logic as: $\forall \mathit{uid}, \mathit{bt}.\, \mathit{WithBonus}(\mathit{uid}, \mathit{bt}) \rightarrow \exists \mathit{card}.\, \mathit{User}(\mathit{uid}, \mathit{card})$. Another constraint limits the bonus type values in *WithBonus* and can be expressed as $\forall \mathit{uid}, \mathit{bt}.\, \mathit{WithBonus}(\mathit{uid}, \mathit{bt}) \rightarrow \mathit{bt} = \texttt{50\%} \vee \mathit{bt} = \texttt{15eur} \vee \mathit{bt} = \texttt{extra\_item}$. □

Data Logic Layer. The data logic layer allows one to *extract* data from the database instance using queries as well as to *update* the database instance by adding and deleting possibly multiple facts at once. The updates follow the *transactional* semantics: if a new database instance obtained after some update is still compliant with the persistence layer, the update is *committed*; otherwise it is *rolled back*. Such updates are realized in parametric atomic actions, resembling ADL actions in planning [4], and consist of fact templates – expressions that, once instantiated, assert which facts will be added to and deleted from the database. Specifically, given a typed relation $R(\mathcal{D}_1, \ldots, \mathcal{D}_n) \in \mathcal{R}$, an R-fact template over $\vec{p}$ has the form $R(y_1, \ldots, y_n)$, such that for every $i \in \{1, \ldots, n\}$, y_i is either a value $\mathtt{o} \in \Delta_{\mathcal{D}_i}$, or a variable $x \in \vec{p}$ with $\mathtt{type}(x) = \mathcal{D}_i$.

A *(parameterized) action* over a $\mathfrak{D}$-typed persistence layer $\langle \mathcal{R}, \mathcal{E} \rangle$ is a tuple $\langle \mathtt{n}, \vec{p}, F^+, F^- \rangle$, where: *(i)* $\mathtt{n}$ is the *action name*; *(ii)* $\vec{p}$ is a tuple of pairwise distinct variables from $\mathcal{V}_{\mathfrak{D}}$, denoting the *action (formal) parameters*; *(iii)* F^+ and F^- respectively represent a finite set of $\mathcal{R}$-fact templates (i.e., some R-fact templates for some $R \in \mathcal{R}$) over $\vec{p}$, to be *added* to and *deleted* from the current database instance. To access the different components of an action α, we use a dot notation: $\alpha{\cdot}\mathtt{name} = \mathtt{n}$, $\alpha{\cdot}\mathtt{params} = \vec{p}$, $\alpha{\cdot}\mathtt{add} = F^+$, and $\alpha{\cdot}\mathtt{del} = F^-$. Given an action α and a (parameter) substitution θ for $\alpha{\cdot}\mathtt{params}$, we call *action instance* $\alpha\theta$ the (ground) action resulting by substituting parameters of α with corresponding values from θ. Then, given a $\mathfrak{D}$-typed database instance $\mathcal{I}$ compliant with $\mathfrak{D}$, the *application* of $\alpha\theta$ on $\mathcal{I}$, written $\mathtt{apply}(\alpha\theta, \mathcal{I})$, is a database instance over $\mathcal{R}$ obtained as $(\mathcal{I} \setminus F^-_{\alpha\theta}) \cup F^+_{\alpha\theta}$, where: *(i)* $F^-_{\alpha\theta} = \bigcup_{R(\vec{y}) \in \alpha \cdot \mathtt{del}} R(\vec{y})\theta$; *(ii)* $F^+_{\alpha\theta} = \bigcup_{R(\vec{y}) \in \alpha \cdot \mathtt{add}} R(\vec{y})\theta$. If $\mathtt{apply}(\alpha\theta, \mathcal{I})$ complies with $\mathcal{P}$, $\alpha\theta$ can be *successfully applied* to $\mathcal{I}$. Note that, in order to avoid situations in which the same fact is asserted to be added and deleted, we prioritize additions over deletions.

Definition 3. *Given a $\mathfrak{D}$-typed persistence layer $\mathcal{P}$, a* $\mathfrak{D}$-typed data logic layer over $\mathcal{P}$ *is a pair $\langle \mathcal{Q}, \mathcal{A} \rangle$, where: (i) $\mathcal{Q}$ is a finite set of* $\mathtt{FO}(\mathfrak{D})$ *queries over $\mathcal{P}$; (ii) $\mathcal{A}$ is a finite set of actions over $\mathcal{P}$.*

Example 2. We make the scenario of Example 1 operational, introducing a data logic layer $\mathcal{L}$ over $\mathcal{P}$. To inspect the persistence layer, we use the following queries:

- $\mathtt{Q}_{\mathtt{products}}(\mathit{pid}, n, c)$:- $\mathit{Product}(n) \wedge \mathit{InWarehouse}(\mathit{pid}, n, c) \wedge c \neq \mathtt{null}$, to extract products available in the warehouse and whose price is not $\mathtt{null}$ (those without prices can be undergoing the stock-taking process);
- $\mathtt{Q}_{\mathtt{users}}(\mathit{uid})$:- $\exists \mathit{card}.\, \mathit{User}(\mathit{id}, \mathit{card})$, to get registered users;
- $\mathtt{Q}_{\mathtt{wbonus}}(\mathit{uid}, \mathit{bt}')$:- $\mathit{WithBonus}(\mathit{uid}, \mathit{bt}')$, to inspect all users with bonuses.

In addition, $\mathcal{L}$ provides key functionalities for organizing the shopping process. Such functionalities are realized through four actions (where, for simplicity, we blur the distinction between an action and its name). To manage bonuses we use two actions ADDB and CHANGE. The former is used to assign a bonus of type bt to a user with id uid (ADDB·`params` $= \langle uid, bt \rangle$) and record it into the persistent storage: ADDB·`add` $= \{ WithBonus(uid, bt) \}$, ADDB·`del` $= \emptyset$. Note that, before logging in, the user may have already a bonus assigned during one of the previous sessions. At will, such a bonus can be changed using action CHANGE with CHANGE·`params` $= \langle uid, bt, bt' \rangle$, CHANGE·`add` $= \{ WithBonus(uid, bt) \}$ and CHANGE·`del` $= \{ WithBonus(uid, bt') \}$. In fact, CHANGE realizes an update by first deleting a tuple that is characterized by uid and bt' (the old bonus), and then adding its modified version. We use RESERVE (RESERVE·`params` $= \langle pid, n, c \rangle$) to reserve product pid of price c and stored in cart cid for further processing (e.g., the preparation for shipment) by deleting it from the list of available products: RESERVE·`add` $= \emptyset$, RESERVE·`del` $= \{ InWarehouse(pid, n, c) \}$. At last, the user may utilize her monthly bonus bt (if it has not yet been used) to consider it when paying the order. For that, we use an action called APPLY (with APPLY·`params` $= \langle uid, bt \rangle$) such that: APPLY·`add` $= \emptyset$, APPLY·`del` $= \{ WithBonus(uid, bt) \}$. □

Control Layer. The control layer employs a fragment of Coloured Petri net to capture the process control flow and a data logic to interact with an underlying persistence layer. We fix some preliminary notions. We consider the standard notion of a *multiset*. 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. Given $a \in A$ and $n \in \mathbb{N}$, 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 $k \in \mathbb{N}$, $k \cdot S_1 = \{a^{kn} \mid a^n \in S_1\}$.

We shall call *inscription* a tuple of typed variables (and, possibly, values) and denote the set of all possible inscriptions over set $\mathcal{Y}$ as $\Omega_{\mathcal{Y}}$, and the set of variables appearing inside an inscription $\omega \in \Omega_{\mathcal{Y}}$ as $Vars(\omega)$ (such notation naturally extends to sets and multisets of inscriptions). In the spirit of CPNs, the control layer assigns to each place a color type, which in turn combines one or more data types from $\mathfrak{D}$. Formally, a $\mathfrak{D}$-*color* is $\mathcal{D}_1 \times \ldots \times \mathcal{D}_m$, where for each $i \in \{1, \ldots, m\}$, we have $\mathcal{D}_i \in \mathfrak{D}$. We denote by Σ the set of all possible $\mathfrak{D}$-colors. To account for fresh external inputs, we employ the well-known mechanism adopted in ν-Petri nets [11,17] and introduce a countably infinite set $\Upsilon_{\mathfrak{D}}$ of $\mathfrak{D}$-typed *fresh variables*. To guarantee an unlimited provisioning of fresh values, we impose that for every variable $\nu \in \Upsilon_{\mathfrak{D}}$, we have that $\Delta_{\mathtt{type}(\nu)}$ is countably infinite. Hereinafter, we shall fix a countably infinite set of $\mathfrak{D}$-typed variable $\mathcal{X}_{\mathfrak{D}} = \mathcal{V}_{\mathfrak{D}} \cup \Upsilon_{\mathfrak{D}}$ as the disjoint union of "normal" variables $\mathcal{V}_{\mathfrak{D}}$ and fresh variables $\Upsilon_{\mathfrak{D}}$.

As we have mentioned before, the control layer can be split into two parts. Let us first define the ν-CPN part that can be seen as an extension of ν-Petri nets with concrete data types, boolean (type-aware) guards and read arcs.

Definition 4. *A $\mathfrak{D}$-typed ν-CPN $\mathcal{N}$ is a tuple $\langle P, T, F_{in}, F_{out}, \texttt{color}\rangle$, where:*

1. *P is a finite set of places.*
2. *$\texttt{color} : P \to \Sigma$ is a color type assignment over P mapping each place $p \in P$ to a corresponding $\mathfrak{D}$-type color.*
3. *T is a finite set of transitions, such that $T \cap P = \emptyset$.*
4. *$F_{in} : P \times T \to \Omega^{\oplus}_{\mathcal{V}_\mathfrak{D}}$ is an input flow from P to T assigning multisets of inscriptions (over variables $\mathcal{V}_\mathfrak{D}$) to input arcs, s.t. that each of such inscriptions $\langle x_1, \ldots, x_m\rangle$ is compatible with each of its input places p, i.e., for every $i \in \{1, \ldots, m\}$, we have $\texttt{type}(x_i) = \mathcal{D}_i$, where $\texttt{color}(p) = \mathcal{D}_1 \times \ldots \times \mathcal{D}_m$.*
5. *$\texttt{guard} : T \to \mathbb{F}_\mathfrak{D}$ is a transition guard assignment over T assigning to each transition $t \in T$ a $\mathfrak{D}$-typed guard φ, s.t.:*
   - *$InVars(t) = \{x \in \mathcal{V}_\mathfrak{D} \mid$ there exists $p \in P$ such that $x \in Vars(F_{in}(\langle p, t\rangle))\}$ is the set of all variables occurring on input arc inscriptions of t;*
   - *a $\mathfrak{D}$-typed* guard *from is a formula (or a quantifier- and relation-free $\texttt{FO}(\mathfrak{D})$ query) of the form $\varphi ::= \texttt{true} \mid S(\vec{y}) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$, where $S/n \in \Gamma_\mathcal{D}$ and, for $\vec{y} = \langle y_1, \ldots, y_n\rangle \subseteq \mathcal{V}_\mathfrak{D}$, we have that y_i is either a value $\texttt{o} \in \Delta_\mathcal{D}$, or a variable $x_i \in \mathcal{V}_\mathfrak{D}$ with $\texttt{type}(x_i) = \mathcal{D}$ $(i \in \{1, \ldots, n\})$;*
   - *$\mathbb{F}_\mathfrak{D}$ is the set of all possible $\mathfrak{D}$-typed guards and, with a slight abuse of notation, $Vars(\varphi)$ is the set of variables occurring in φ.*
6. *$F_{out} : T \times P \to \Omega^{\oplus}_{\mathcal{X}_\mathfrak{D} \cup \Delta_\mathfrak{D}}$ is an* output flow *from transitions T to places P assigning multisets of inscriptions to output arcs, such that all such inscriptions are compatible with their output places.*

According to the diagram in Fig. 1, the DB-net control layer can be obtained on top of ν-CPNs by essentially adding three mechanisms that allow the net to interact with the underlying persistent storage: *(i)* view places, allowing the net to inspect parts of the database using queries; *(ii)* action binding, linking atomic actions and their parameters to transitions and their inscription variables; *(iii)* rollback transition-place arcs, enacted when the action application induced by a transition firing violates some database constraint, so as to explicitly account for "error-handling".

Definition 5. *A $\mathfrak{D}$-typed* control layer *over a data logic layer $\mathcal{L} = \langle \mathcal{Q}, \mathcal{A}\rangle$ is a tuple $\langle P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{act}\rangle$, where:*

1. *$\langle P_c, T, F_{in}, F_{out}, \texttt{color}\rangle$ is a $\mathfrak{D}$-typed ν-CPN, where P_c is a finite set of control places.*
2. *$P = P_c \cup P_v$ is a finite set of places, where P_v are view places (decorated as*  *and connected to transitions with special read arcs).*
3. *$\texttt{query} : P_v \to \mathcal{Q}$ is a query assignment mapping each view place $p \in P_v$ with $\texttt{color}(p) = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ to a query $Q(x_1, \ldots, x_n)$ from $\mathcal{Q}$, s.t. the color of p component-wise matches with the types of the free variables in Q: for each $i \in \{1, \ldots, n\}$, we have $\mathcal{D}_i = \texttt{type}(x_i)$.*

4. $\texttt{act} : T \to \mathcal{A} \times \Omega_{\mathcal{X}_\mathfrak{D} \cup \Delta_\mathfrak{D}}$ *is a partial function assigning transitions in* T *to actions in* $\mathcal{A}$*, where* $\texttt{act}(t)$ *maps* t *to an action* $\alpha \in \mathcal{A}$ *together with a (binding) inscription* $\langle y_1, \ldots, y_m \rangle$*, s.t. if* $\alpha{\cdot}\texttt{params} = \langle z_1, \ldots, z_m \rangle$ *and, for each* $i \in \{1, \ldots, m\}$*, we have* $\texttt{type}(y_i) = \texttt{type}(z_i)$ *if* y_i *is a variable from* $\mathcal{X}_\mathfrak{D}$*, or* $y_i \in \Delta_{\texttt{type}(z_i)}$ *if* y_i *is a value from* $\Delta_\mathfrak{D}$*.*
5. F_{rb} *is an output flow from* T *to* P_c *called* rollback flow *(we shall refer to* F_{out} *as* normal output flow*).*

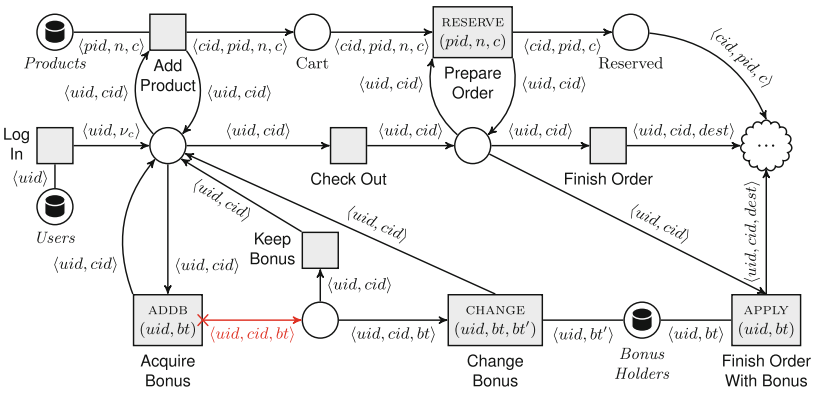


Fig. 3. The control of a DB-net for online shopping. Here, ν_c is a fresh input variable corresponding to a newly created cart, whereas *dest* is an arbitrary input variable representing a destination address. The rollback output arc (corresponds to the rollback flow) is in red and decorated with an "x".

Figure 3 shows the control layer of the shopping cart example. The queries specified in Example 2 are assigned to the corresponding view places: $\texttt{query}(\textit{Products}) := \texttt{Q}_{\texttt{products}}$, $\texttt{query}(\textit{Users}) := \texttt{Q}_{\texttt{users}}$ and $\texttt{query}(\textit{Bonus Holders}) := \texttt{Q}_{\texttt{wbonus}}$. The actions (with their formal parameters) assigned to transitions via $\texttt{act}$ graphically appear in grey transition boxes.

The execution semantics of a DB-net simultaneously accounts for the progression of a database instance compliant with the persistence layer of the net, and for the evolution of a marking over the control layer of the net. Due to space limitations, we refer to the definition of the formal semantics studied in [12]. We thus assume that the execution semantics of both ν-CPNs and DB-nets can be captured with a possibly infinite-state labeled transition system (LTS) that accounts for all possible executions starting from their initial markings. While transitions in such LTSs model the effect of firing nets under given bindings, their state representations slightly differ. Namely, in the case of ν-CPNs we have markings (like, for example, in coloured Petri nets [7]), while in the case of DB-nets one also has to take into account database states. W.l.o.g., we shall use $\Gamma_\mathcal{N}^{M_0} = \langle \mathcal{M}, M_0, \to, L \rangle$ to specify an LTS for a ν-CPN $\mathcal{N}$ with initial marking

M_0 and $\Gamma_{\mathcal{B}}^{s_0} = \langle S, s_0, \rightarrow, L \rangle$ to specify an LTS for a DB-net $\mathcal{B}$ with initial snapshot $s_0 = \langle \mathcal{I}_0, m_0 \rangle$, where $\mathcal{I}_0$ is the initial database instance and m_0 is the initial marking of the control layer.

3 Translation

We are now ready to describe the translation from DB-nets to ν-CPNs with priorities (we assume the reader is familiar with transition priorities). Recall that this is not just of theoretical interest, but has also practical implications. In [16], we have presented a prototypical implementation of DB-nets in CPN Tools that, using Access/CPN and Comms/CPN, allow to model and simulate DB-nets. However, we realized that CPN Tools would not correctly generate the state space of the DB-net at hand. This is due to the fact that the CPN Tools state space construction module does not consider third-party extensions which, in our setting, implies that the content of the view places is not properly recomputed after each transition firing.

The first challenge to overcome is how the database schema is represented in the target net. To this aim, we introduce special *relation places* that copy corresponding database relations by mirroring their signature to the type definitions of places.[1] In this light, database instances will correspond to relation place markings, where tokens are nothing but tuples. All other DB-net elements (for example, bindings for fresh variables, action execution) require actual computation that happens when a transition fires. Intuitively, every DB-net transition T is represented using the following four phases:

1. Collect bindings and compute the content of view places adjacent to T.
2. If there is an action assigned to T, execute it. We employ auxiliary boolean places that control whether an update has actually happened (that is, a token representing a tuple has been removed from or added to a relation place).
3. Check the satisfaction of integrity constraints.
4. Finish the computation and generate a new marking.
   (a) If all constraints are satisfied, empty the auxiliary boolean places used in (2), release the lock, and populate the postset of T.
   (b) If some constraint is violated, roll-back the effects. This is done in reverse order w.r.t. phase (2), applying or skipping a reverse update depending on how the values in the special places. After this, the relation places have the content they had before the action was applied. Then, one releases the lock and pushses the special postset corresponding to the roll-back arc (if any) attached to T.

To realize the execution of an original DB-net transition, all the four phases are executed uninterruptedly (under lock). In the reminder of the section we formalize the phases discussed above.

[1] Relation places do not differ from the normal ν-CPN places. We use the different name in order to conceptually distinguish their origin.

A generic DB-net $\mathcal{B}_\tau$ that we use to demonstrate the translation is represented in Fig. 4(a). Here, we assume that T contains enough of tokens assigned by its input flow and its eventual firing is subject to the $G(\vec{y})$ guard evaluation. $\vec{y}$, in turn, is bound to values from $\vec{z}$ and from $m \in \mathbb{N}$ ordered view places, where each view place V_i has a query $\mathtt{Q}_{\mathtt{V}_i}$ assigned to it. The ν-CPN $\mathcal{N}_\tau$ representing

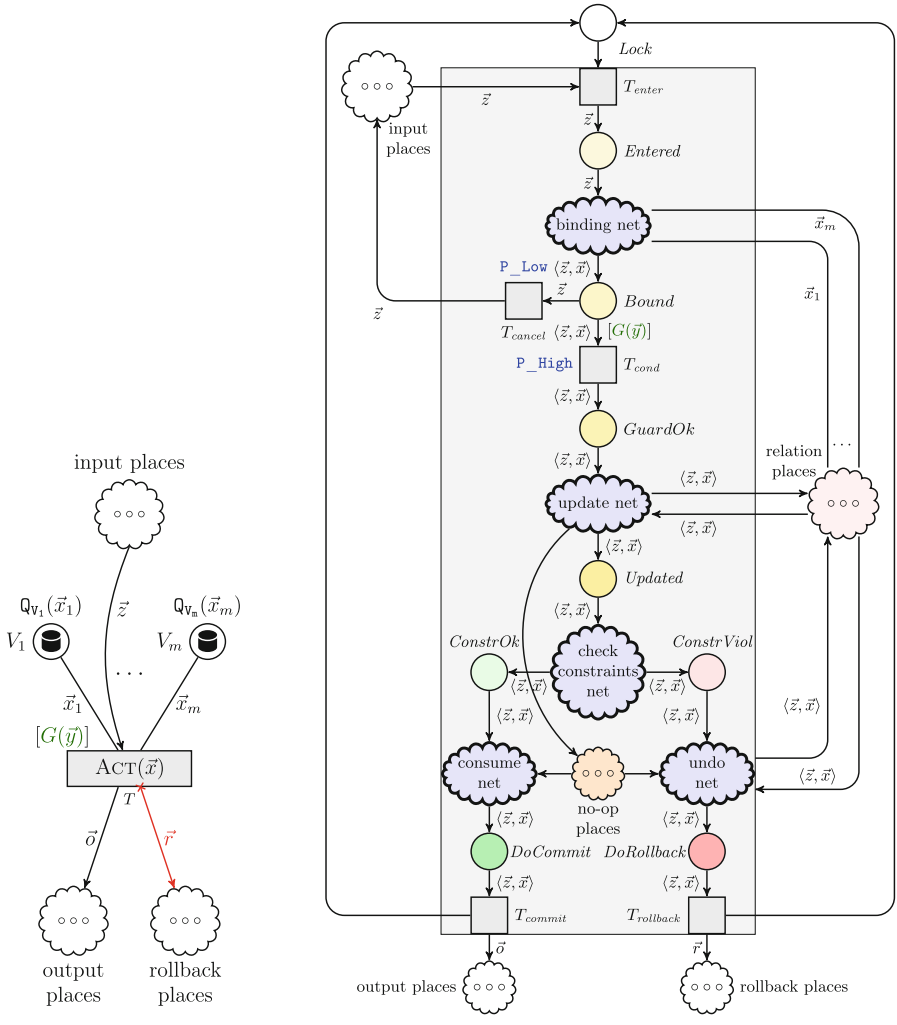
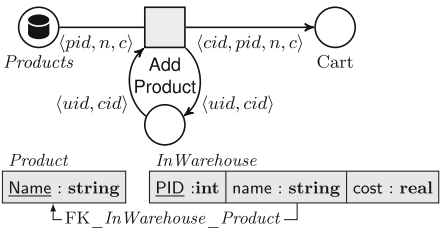


Fig. 4. A generic DB-net transition accessing multiple view places (left) and its overall ν-CPN encoding (right). Blue clouds stand for subnets that are expanded next, and $\vec{x}$ is a shortcut for $\vec{x}_1, \ldots, \vec{x}_m$. Elements within the gray rectangle are local to the transition, whereas external elements are shared at the level of the whole net. (Color figure online)

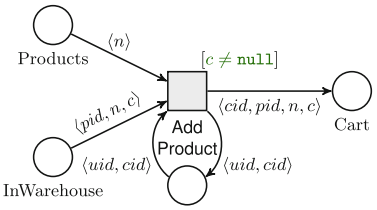
$\mathcal{B}_\tau$ is depicted in Fig. 4(b). To facilitate the translation, we make three working hypothesis. First, we assume that the relational schema is equipped only with three types of constraints: primary keys, foreign keys and domain constraints. Second, for ease of presentation, we consider that the resulting ν-CPN model can deal with DB-net external inputs. For each $t \in T$, such inputs are in general modeled using variables that do not appear in $InVars(t)$ (including those from $\Upsilon_{\mathfrak{D}}$). The preliminary implementation of DB-nets in CPN Tools [16] has provided functionality demonstrating the feasibility of the binding computation for such variables. Third, we naturally extend the notion of ν-CPN with read arcs.

3.1 Computing Views Using CPN Places

We start by describing how the view computation should work using only ν-CPN places. Let us consider as an example a subnet $\mathcal{B}_{tr}$ of the DB-net present in Fig. 3 that models only the selection of available products. To access products that are available in the warehouse and that have prices assigned to them, we need to run a query $\mathtt{Q}_{\mathtt{products}}(pid, n, c) = Product(n) \wedge InWarehouse(pid, n, c) \wedge c \neq \mathtt{null}$. Interestingly, such a query can be formulated directly using standard elements of ν-CPNs. Indeed, we may transfer the DB-net in Fig. 5(a) into a ν-CPN $\mathcal{N}_{tr}$ in Fig. 5(b) representing the project selection step. As one can see, the relations of $\mathcal{B}_{tr}$ have been copied to the same-named relation places, when $\mathtt{Q}_{\mathtt{products}}$ is treated as follows: $\mathcal{N}_{tr}$ accesses relation places with read-arcs (that have relation attributes as their inscriptions) so as to realize the projection, while the filter (i.e., $c \neq \mathtt{null}$) is basically plugged into the guard of Add Product. The result of the query is then propagated into the post-set of Add Product using the free variables of $\mathtt{Q}_{\mathtt{products}}$ (i.e., pid, n and c) in the arc inscriptions.



(a) Fragments of the process and persistence layers of the DB-net in Figure 3



(b) A ν-CPN representation of the DB-net in Figure 5(a)

Fig. 5. Example of a view computation in ν-CPNs

However, one may see that not every query can be handled when only using standard ν-CPN elements. Assume a query $\mathtt{Q}_{\neg\mathtt{available}}(n) = Product(n) \wedge \neq \exists pid, c.InWarehouse(pid, n, c)$ that lists products not available in the warehouse. In order to represent $\mathtt{Q}_{\neg\mathtt{available}}$ in a ν-CPN, one would need to extend the net with constructs allowing to fire a transition only if a certain element does

not exists in a place incident to it. Thus, we restrict ourselves to the union of conjunctive queries with negative filters (or atomic negations) (UCQFs$^{\neq}$), that is $\texttt{FO}(\mathfrak{D})\mathfrak{D}$ queries of the form $\bigvee_{i=1}^{n} \exists \vec{y_i}.conj_i(\vec{x})$, where $conj_i(\vec{x})$ is also a $\texttt{FO}(\mathfrak{D})\mathfrak{D}$ query that is a conjunction of relations $R(\vec{z})$, predicates $P(\vec{y})$ and their negations $\neg P(\vec{y})$. Henceforth, we use $\mathcal{Q}^{UCQF^{\neq}}$ to define a UCQF$^{\neq}$ subset of $\mathcal{Q}$. In SQL, a conjunctive query is a query representable with a SELECT-FROM-WHERE expression. As it has been already shown, the filter conditions (of the UCQFs$^{\neq}$ attached to view places) can be modeled using transition guards.

In case of multiple view places attached to one transition, we construct a net that computes them in a sequential manner. One may see the computation process as a pipeline. Whenever a transition that corresponds to a certain view place is enabled, it fires and generated tokens that represent one of the tuples of the view. Then, acquired tokens are transferred to the next transition using variables in the arc inscriptions. The computation continues until the last view. After that, the results of all the computations are transferred to the corresponding places, following the topology (i.e., the organization of arcs defined by the flow relations) of the original DB-net. Note that the order in which views are computed has to be the same as the one defined for $\mathcal{B}_\tau$.

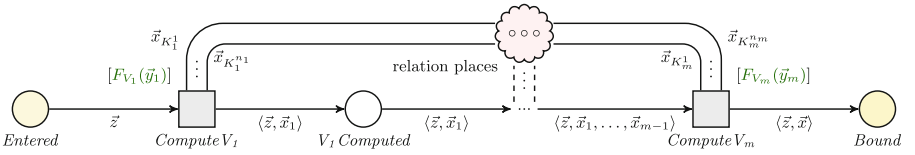


Fig. 6. Expansion of the binding net from Fig. 4(b)

A ν-CPN in Fig. 6 shows how bindings and view places are computed in the case of the generic DB-net $\mathcal{B}_\tau$. The computation process per view V_i is realized by a transition called *Compute* V_i and analogous to the one explained before: we read necessary data from relation places, representing relations used in $\texttt{Q}_{\texttt{V}_\texttt{i}}$, and filter these data by means of $F_{V_i}(\vec{y})$. Note that variables on every read-arc adjacent to *Compute* V_i represent attributes of some relation R. The intermediate result of the view computation is then stored in a place called V_i *Computed*. As one can see from Fig. 6, all the intermediate results are accumulated along the computation cycle. Moreover, we carry data provided with input variables of T so as to check the validity of the guard G (see Fig. 4(b)). This is done using prioritized transition T_{cond}. If the guard is not satisfied, one has to reset the computation process by returning tokens that have been consumed at the beginning of the view computation (that is, tokens that have been assigned to z). We resolve this issue by introducing an auxiliary transition called T_{cancel} that may fire only when the guard has been evaluated to false. Scheduling between T_{cond} and T_{cancel} is managed by means of two priority labels `P_High` and `P_Low` (where `P_High` > `P_Low`) respectively assigned to them.

3.2 Modelling RDBMS Updates in CPNs

We now show how database updates exploited by DB-nets could be represented using regular coloured Petri nets. We recall, that actions assigned to DB-net transitions support addition and deletion of $\mathcal{R}$-fact, which should preserve the set semantics adopted by the persistence layer.

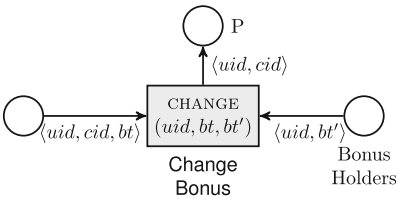


Fig. 7. A subnet of the DB-net in Fig. 3 describing the bonus change step

In Fig. 7 we consider a DB-net describing the bonus change step of he online shopping process. Here, for ease of presentation, instead of considering a view place for bonus holders, we use a regular (control) place that stores the same kind of data.

The translation of DB-net-like database updates into ν-CPNs is conceptually similar to the representation of the view computation process: DB-net actions must be performed sequentially within a critical section that can be entered whenever a special write lock is available (cf. place *Lock* in Fig. 4(b)). For preserving the set semantics over every relation place, we use prioritized transitions so as to check whether a tuple to be added or deleted already exists in the relation place. Specifically, for each tuple we would introduce two transitions, one with a higher priority and another with a lower priority, and an auxiliary (*no-op*) boolean place. The first transition can fire if the tuple is in the corresponding relation place, while the second one would fire otherwise. Both transitions are adopted to deal with additions and deletions. In case of additions, the highly prioritized transition would not add the tuple, while the one with the lower priority would do otherwise. To deal with deletions, we mirror the previous case: if the tuple exists, then one can safely remove it; otherwise, one proceeds without changes. Upon firing of any of these transitions, the auxiliary place receives a boolean token. If the value of the token is true, then it means that the tuple has been successfully added or deleted. In case the database update has not taken

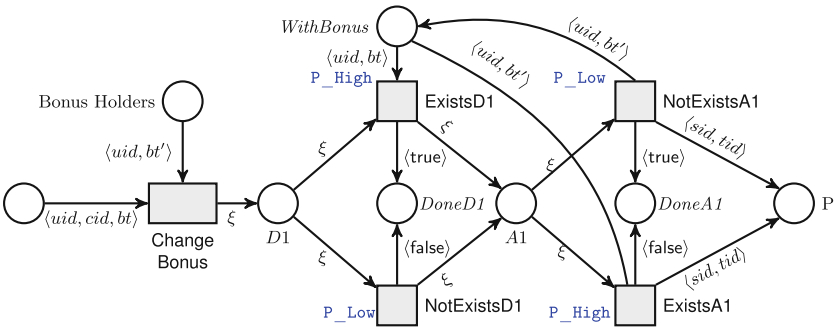


Fig. 8. The CPN representation of the DB-net in Fig. 7, where ξ, for ease of reading, denotes the tuple $\langle uid, cid, bt, bt' \rangle$

place, the token value is going to be false. It is important to note that the update execution order of DB-net actions must be also preserved in their ν-CPN representation. That is, since additions are prioritized over deletions, for every action α we first delete all the tuples from $\alpha{\cdot}\texttt{del}$, and only then add those from $\alpha{\cdot}\texttt{add}$.

We incorporate aforementioned modelling guidelines in the ν-CPN depicted in Fig. 8. Since CHANGE in $\mathcal{B}^{\alpha}_{tr}$ contains multiple database updates, the model starts with deleting $\mathit{WithBonus}(\mathit{uid}, \mathit{bt})$ from $\mathit{WithBonus}$. To do so, at first one checks whether the relation place $\mathit{WithBonus}$ contains the tuple we would like to remove. This is done using Exists D1 that performs conditional removal of $\mathit{WithBonus}(\mathit{uid}, \mathit{bt})$, that is, if there is a token in $\mathit{WithBonus}$ such that bindings of inscriptions on $(D1, \mathit{ExistsD1})$ and $(\mathit{WithBonus}, \mathit{ExistsD1})$ coincide, then ExistsD1 is enabled, and upon firing consumes the selected token from $\mathit{WithBonus}$ and populates one token with value $\langle \mathit{true} \rangle$ (the value true means that the update has been successfully accomplished) in $\mathit{DoneD1}$. Note that Exists D1 is always checked first given the higher priority label assigned to it. If the tuple does not exist, then one proceeds with firing NotExistsD1 and populating one token with value false in $\mathit{DoneD1}$. Now, when we reach the first control place allowing to perform the add operation over $\mathit{WithBonus}$, we start by checking whether $\mathit{WithBonus}$ already contains the $\mathit{WithBonus}(\mathit{uid}, \mathit{bt}')$ tuple. Specifically, we use the read arc $(\mathit{ExistsA1}, \mathit{WithBonus})$ that has the only purpose of checking whether the token is present in the place. In case there is no token that matches values assigned to ξ, we proceed with adding $\mathit{WithBonus}(\mathit{pid}, \mathit{bt}')$ with NotExistsA1 that has the lower priority label assigned to it and consequently populate a $\langle$true$\rangle$-valued token in $\mathit{DoneA1}$. Note that the whole computation process is "guarded" with the global lock variable (needed for the consequent execution of all the steps defined in Fig. 4(b)): whenever started, the token is removed from it and can be returned only after the last operation of the action has been carried out.

Next we show how an action is encoded considering the general DB-net $\mathcal{B}$ in Fig. 4(a). Note that T is equipped with action ACT, where some of the action parameters $\vec{x}$ coincide with external inputs. ACT is defined on top of $\mathcal{P}$ with ACT$\cdot$params $= \langle x_1, \ldots, x_n \rangle$, ACT$\cdot$del $= F^-$ and ACT$\cdot$add $= F^+$, where F^- and F^+ are two sets of $\mathcal{R}$-facts that should be respectively deleted and added. The CPN representing that expansion of the update net from Fig. 4(b) is depicted in Fig. 9. The computation starts by checking the guard of T with transition T_{cond} (cf. Fig. 4(b)). If the guard evaluates to true, T_{cond} puts a token in a place called $\mathit{GuardOk}$ that, in turn, allows to initiate the action execution process that is sequentially realized for all R-facts from F^- or F^+ in ACT.

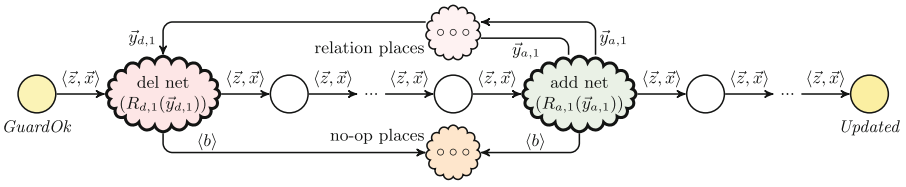
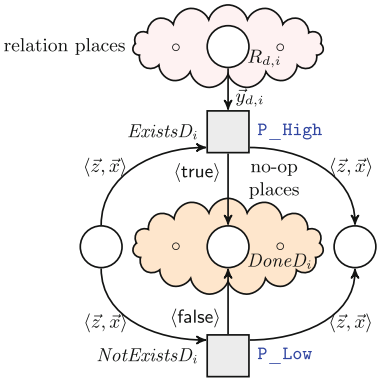
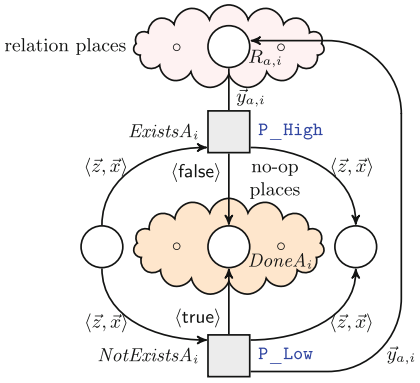


Fig. 9. Expansion of the update net from Fig. 4(b)

We first proceed with deleting all the $R_{d,i}$-facts from F^- (i.e., facts of the form $R_{d,i}(\vec{y}_{d,i})$). This process is sequntialized and at each of its step the net models the deletion of only one $R_{d,i}$-fact. Specifically, the deletion of each $R_{d,i}$-fact (see Fig. 10(a)) is realized by a pair of prioritized transitions $\mathsf{ExistsD}_i$ and $\mathsf{NotExistsD}_i$ and one auxiliary place $DoneD_i$, and is analogous to the example in Fig. 8. After all the R-facts from F^- have been deleted, the net switches to performing the insertion of R-facts from F^+. We omit the details of the addition process as it can be defined analogously to the one from the bonus change example and refer to Fig. 10(b). As soon as all R-facts are added, the update net completes its work by putting a token into a place called *Updated*.



(a) Expansion of the i-th deletion component in the net of Figure 9



(b) Expansion of the i-th addition component in the net of Figure 9

Fig. 10. Expansion of deletion and addition nets from Fig. 9

3.3 Checking Integrity Constraints and Generating a New Marking

Let us now remind that the relational schema of $\mathcal{B}_\tau$ is equipped with three types of integrity constraints: primary keys, foreign keys and domain constraints. When the first and the last one could be relatively easy to check during the update phase, assuming that the computation results are accumulated in arc inscriptions analogously to the binding net in Fig. 6[2], the process of managing updates in the presence of arbitrary many foreign key dependencies is quite involved. To manage it correctly we first perform the updates and only then check whether the

[2] Both primary keys and domain constraints can be violated when a tuple is about to be inserted into a table. Specifically, to guarantee that former are respected, it is enough to check with $ExistsA_i$ whether there is a token in $R_{a,i}$ that has the same primary key value, and, if so, cancel the computation process. In the case of domain constraints, one may insert a third transition t' that has a normal priority and that will be fired whenever one of the values we want to insert is not in the allowed range. Firing of t' will have the same consequences as in the case of primary keys.

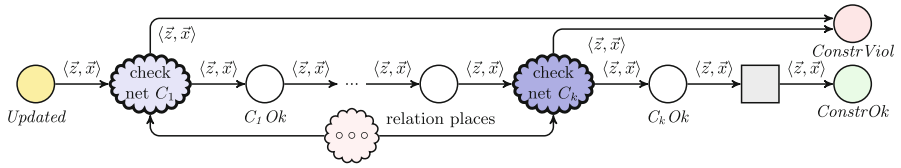
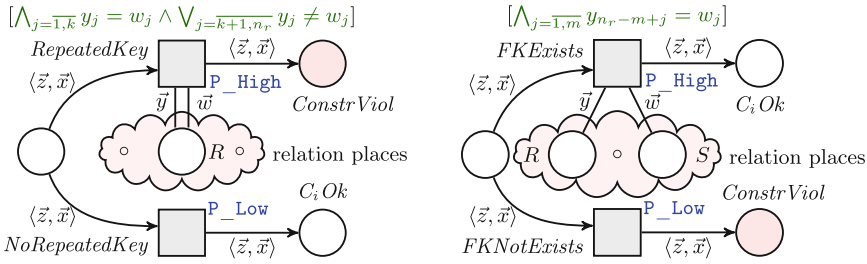


Fig. 11. Expansion of the check constraint net from Fig. 4(b)

generated marking represents a database instance that satisfies all the integrity constraints contained in the persistence layer of $\mathcal{B}_\tau$. A ν-CPN representing the check constraint phase is depicted in Fig. 11. The net works as follows: it consequently runs small nets for verifying the integrity of constraints and, in case of violation, puts a token in a special place called *ConstrViol*. As soon as there is at least one token in *ConstrViol* place, the big net in Fig. 4(b) terminates the constraint checking process and switches to the phase 4.(b) (that is, runs the undo net) explained in the beginning of this section. For ease of presentation we assume that every relation R/n_r from $\mathcal{R}$ of $\mathcal{B}_\tau$ will have the following look: the first k attributes form a primary key, while the rest of $n_r - k + 1$ attributes can be unconstrained or bounded by domain constraints. Moreover, if R is referencing some other relation S, then among these $n_r - k + 1$ attributes we reserve the last m such that $S[\{1, \ldots, m\}] \subseteq R[\{n_r - m, \ldots, n_r\}]$.

The constraint checking process starts with verifying that all the updates performed using the net in Fig. 9 are satisfying primary key constraints C_i. This is done by sequentially running small check nets in Fig. 12(a), where the constraint integrity is verified for some relation R by a pair of prioritized transitions *RepeatedKey* (high priority) and *NoRepeatedKey* (low priority). Note that *RepeatedKey* accesses the content of R with two read arcs and using the guard assigned to it verifies whether there exist two tokens, such that their first



(a) Expansion for a primary key constraint C_i indicating that the first k components of relation R with arity $n_r \geq k$ form a key for R. In the figure, $\vec{y}$ and $\vec{w}$ are tuples containing n_r variables.

(b) Expansion for a foreign key constraint C_i indicating that the last m components of relation R with arity $n_r \geq m$ reference the last m components of relation S with arity $n_s \geq m$. In the figure, $\vec{y}$ and $\vec{w}$ are tuples containing n_r and n_s variables.

Fig. 12. Expansion of the check net from Fig. 11 in the case of key constraints

k values coincide and in the rest of $n_r - k + 1$ values there is at least one distinct pair of values. The satisfaction of such guard would mean that, essentially, we have inserted a token in R whose primary key values were not unique. Firing of $RepeatedKey$ will produce one token in $ConstViol$ and terminate the run of the check constraint net.

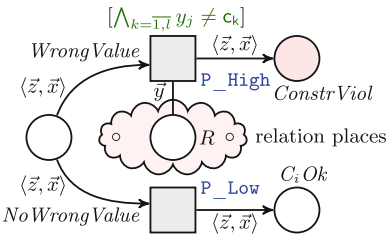


Fig. 13. Expansion of the check net from Fig. 11 in the case of a domain constraint C_i indicating that the j-th component of relation R with arity $n_r \geq j$ must contain a value that belongs to $\{\mathsf{c_1}, \ldots, \mathsf{c_l}\}$; In the figure, $\vec{y}$ is a tuple containing n_r variables.

The next type of constraints to verify is the foreign key dependency. Analogously to the previous case, we successively run small check nets like the one in Fig. 12(b) and in each of them control that R correctly references S (that is, there are no tuples in R that do not depend on any tuple in S). This is realized with two prioritized transitions $FKExists$ (high priority) and $FKNotExists$ (low priority). The first one, as the name suggests, checks whether the dependency between R and S is preserved for all the tokens in the corresponding relation places. $FKExists$ makes use of the guard attached to it that performs pairwise comparison of m last values of a token from R to m first values of a token from S. If the guard is not satisfied, then the dependency relation between R and S has been violated and one fires $FKNotExists$ so as to terminate the constraint checking process.

The last series of constraints to be checked is the one of domain constraints. The net in Fig. 13 employs two prioritized transitions, $WrongValue$ and $NoWrongValue$, to verify whether all the tuples inserted into R had correct values. First, using the guard of $WrongValue$, we check whether there is at least one value that breaks the integrity of the domain constraint C_i of R. If $WrongValue$ fires, the process is terminated by putting a token into $ConstrViol$. Otherwise, $NoWrongValue$ is executed and the constraint checking process continues.

Now let us show how the computation of all the effects of T is finished and a new marking is generated. If one of the constraints has been violated, we have to roll back all the effects pushed using the net in Fig. 9. To do so, we employ the net in Fig. 14 that reverts the update process by first canceling all the additions, and

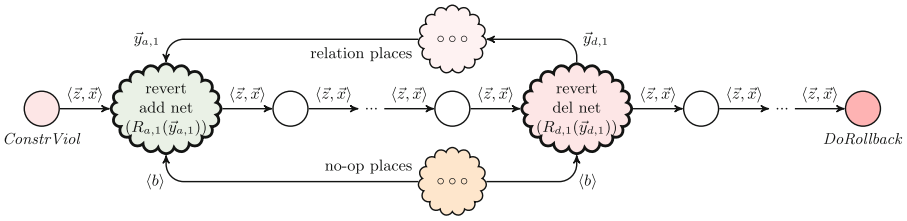
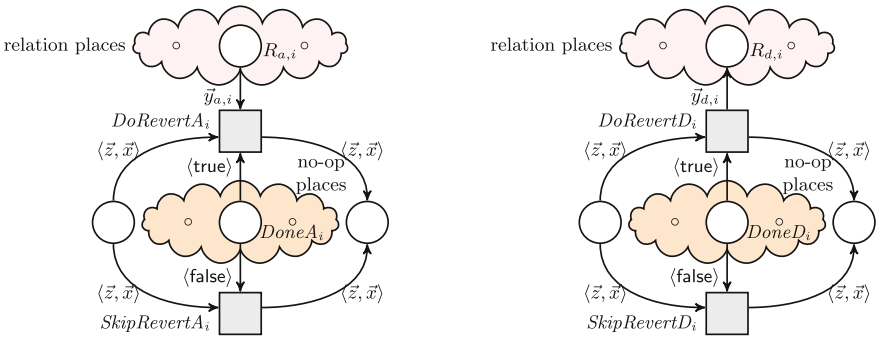


Fig. 14. Expansion of the undo net from Fig. 4(b)



(a) Expansion of the i-th revert addition component

(b) Expansion of the i-th deletion component

Fig. 15. Expansion of revert deletion and addition nets from Fig. 14

then canceling all the deletions. Let us briefly explain how the rollback process is performed for each component net.

We start by removing all the tuples that have been successfully added to relation places following the definition of Act. The net in Fig. 15(a) shows how to revert the result of inserting $R_{a,i}$-fact from F^+ (i.e., a fact of the form $R_{a,i}(\vec{y}_{a,i})$). If a fact has been added, that is, there is a token with value $\langle$true$\rangle$ in $DoneA_i$, then the net removes it by firing $DoRevertA_i$. Otherwise, if the fact has not been added, that is, there is a token with value $\langle$false$\rangle$ in $DoneA_i$, then the net proceeds without reverting by firing $SkipRevertA_i$. Then, for each $R_{d,i}$-fact, we go on with adding all the tuples that have been deleted by using the net depicted in Fig. 15(b). The update reverting processis analogous to the one dealing with reverted additions, but with only one exception: whenever $DoneD_i$ has a $\langle$true$\rangle$ token, then we put the deleted tuple (specified in F^-) back into $R_{d,i}$. Note that every revert deletion or addition net removes a token from a corresponding auxiliary no-op place.

As soon as all the operations of Act have been undone and all the corresponding tokens have been withdrawn from relation places, the net places a token in $DoRollBack$ (cf. Figure 4(b)) and allows us to fire a transition called $T_{rollback}$ that implements the generation of the tokens in the postset corresponding to the rollback flow of T.

If the check constraint net's work has not been interrupted an the token was placed in $ConstrOk$ (cf. Fig. 4(b)), then we proceed with the consume net (cf. Fig. 16) that removes all tokens from the auxiliary no-op places and places a token into $DoCommit$. This, in turn, allows $\mathcal{N}_\tau$ to execute T_{commit} that populates tokens in the postset corresponding to the normal flow of T.

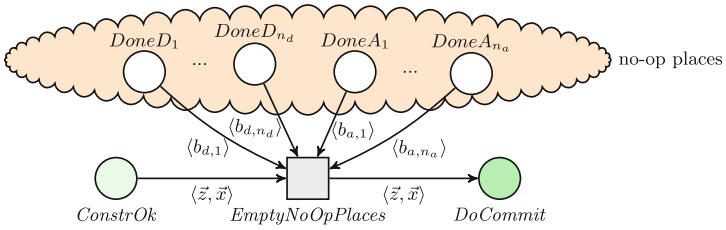


Fig. 16. Expansion of the consume net from Fig. 4(b)

3.4 The General Translation

In this section we bring together the modelling approaches described in the previous three sections and quickly summarize the translation from DB-nets to ν-CPNs with priorities. Specifically, we show that, given a DB-net, it is possible to build a ν-CPN that is weakly bisimilar to it.

Intuitively, $\mathcal{N}_\tau$ from Fig. 4(b) behaves just like the $\mathcal{B}_\tau$ in Fig. 4(a) and hence LTSs of these two nets are weakly bisimilar [10]. Notice that, in order to correctly represent the behavior of $\mathcal{B}_\tau$, $\mathcal{N}_\tau$ includes many intermediate steps that are, however, not relevant for comparing content of the states and behavior of the nets. For this we are going to resort to a form of bisimulation that allows to "skip" transitions irrelevant for the behavioral comparison [10]. Specifically, given two transition systems $\Gamma_1 = \langle S_1, s_{01}, \rightarrow_1, L\rangle$ and $\Gamma_2 = \langle S_2, s_{02}, \rightarrow_2, L\rangle$ defined over a set of labels L, relation $wb \subseteq S_1 \times S_2$ is called a *weak bisimulation* between Γ_1 and Γ_2 iff for every pair $\langle p, q\rangle \in wb$ and $a \in L \cup \{\epsilon\}$ the following holds: (1) if $p \stackrel{a}{\Rightarrow}_1 p'$, then there exists $q' \in S_2$ such that $q \stackrel{a}{\rightarrow}_2 q'$ and $\langle p', q'\rangle \in wb$; (2) if $q \stackrel{a}{\Rightarrow}_2 q'$, then there exists $p' \in S_1$ such that $p \stackrel{a}{\rightarrow}_1 p'$ and $\langle p', q'\rangle \in wb$. Here, $\epsilon \neq a$ is a special silent label and $p \stackrel{a}{\rightarrow} q$ is a weak transition that is defined as follows: *(i)* $p \stackrel{a}{\rightarrow} q$ iff $p(\stackrel{\epsilon}{\Rightarrow})^* q_1 \stackrel{a}{\Rightarrow} q_2(\stackrel{\epsilon}{\Rightarrow})^* q$; *(ii)* $p \stackrel{\epsilon}{\rightarrow} q$ iff $p(\stackrel{\epsilon}{\Rightarrow})^* q$. We use $(\stackrel{\epsilon}{\Rightarrow})^*$ to define the reflexive and transitive closure of $\stackrel{\epsilon}{\Rightarrow}$. We say that a state $p \in S_1$ is weakly bisimilar to $q \in S_2$, written $p \approx^{wb} q$, if there exists a weak bisimulation wb between Γ_1 and Γ_2 such that $\langle p, q\rangle \in wb$. Finally, Γ_1 is said to be weakly bisimilar to Γ_2, written $\Gamma_1 \approx^{wb} \Gamma_2$, if $s_{01} \approx^{wb} s_{02}$. Let us now define a theorem that sets up the behavioral correspondence between DB-nets and ν-CPNs.

Theorem 1. *Let* $\mathcal{B} = \langle \mathfrak{D}, \mathcal{P}, \mathcal{L}, \mathcal{N}\rangle$ *be a DB-net with* $\mathcal{P} = \langle \mathcal{R}, \emptyset\rangle$, $\mathcal{L} = \langle \mathcal{Q}^{UCQF^{\neq}}, \mathcal{A}\rangle$ *and* $\mathcal{N} = \langle P, T, F_{in}, F_{out}, F_{rb}, \texttt{color}, \texttt{query}, \texttt{guard}, \texttt{act}\rangle$, *and* s_0 *is the initial snapshot. Then, there exists a* ν*-CPN* $\mathcal{N} = \langle P \cup P_{rel} \cup P_{aux}, T \cup T_{aux}, F_{in}, F_{out}, \texttt{color}, M_0\rangle$ *with (1) a set of relation places* P_{rel} *acquired from* $\mathcal{R}$*, (2) two sets* P_{aux} *and* T_{aux} *of auxiliary places and transitions (required by the encoding algorithm), and such that* $\Gamma_{\mathcal{B}}^{s_0} \approx^{wb}_{flat} \Gamma_{\mathcal{N}}^{M_0}$.

The proof of the theorem (see [13] for the full version) can be obtained by induction on the construction of $\Gamma_{\mathcal{B}}^{s_0}$ and $\Gamma_{\mathcal{N}}^{M_0}$, where the latter is induced by $\mathcal{N}$ that has been modularly generated using the encoding defined in Sects. 3.1, 3.2 and

3.3. Intuitively, such encoding lifts the persistence and data logic layers to the control layer, resulting in a "pristine" ν-CPN. To show behavioral correspondence, one should make sure that states of $\Gamma_{\mathcal{B}}^{s_0}$ and $\Gamma_{\mathcal{N}}^{M_0}$ are comparable. This can be achieved by slightly modifying the notion of weak bisimulation in such a way that, for each $\langle\langle \mathcal{I}, m\rangle, M\rangle \in wb$, we compare elements stored in $\mathcal{I}$ only with their "control counterparts" in P_{rel} of M, whereas $m \subseteq M$. Moreover, we assume that states of $\Gamma_{\mathcal{N}}^{M_0}$ are restricted only to places in $P \cup P_{rel}$, that is, each marking M shall reveal tokens stored only in P and P_{rel}, and that when constructing $\Gamma_{\mathcal{N}}^{M_0}$ all the auxiliary transitions of $\mathcal{N}$ (i.e., all the transitions within the grey lane in Fig. 4(b)) are going to be labeled with ϵ. Note that such an extended definition allows to establish equivalence not only in terms of behaviors of two systems, but also in terms of their (data) content.

4 Conclusions

We have shown that the large and relevant fragment of DB-nets employing unions of conjunctive queries with negative filters as a database query language, can be faithfully encoded into a special class of Coloured Petri nets with transition priorities. This result is of particular interest as it demonstrates how to represent full-fledged databases with corresponding data manipulation operations in a conventional Petri net class. Since the encoding is based on a constructive technique that can be readily implemented, the next step is to incorporate the encoding into the DB-net extension of CPN Tools [16], in turn making it possible to make the state-space construction mechanisms available in CPN Tools also applicable to DB-nets. It must be noted that, due to the presence of data ranging over infinite colour domains, the resulting state-space is infinite in general. However, in the case of state-bounded DB-nets [12], that is, DB-nets for which each marking contains boundedly many tokens and boundedly many database tuples, a faithful abstract state space can be actually constructed using the same approach presented in [2]. Interestingly, this can be readily implemented by replacing the ML code snippet dealing with fresh value injection with a slight variant that recycles, when possible, old data values that were mentioned in a previous marking but are currently not present anymore.

Acknowledgments. This work has been partially supported by the UNIBZ projects PWORM and REKAP.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Boston (1995)
2. Calvanese, D., De Giacomo, G., Montali, M., Patrizi, F.: First-order mu-calculus over generic transition systems and applications to the situation calculus. Inf. Comput. **259**(3), 328–347 (2018)
3. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: a database theory perspective. In: Proceedings of PODS (2013)

4. Drescher, C., Thielscher, M.: A fluent calculus semantics for ADL with plan constraints. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 140–152. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87803-2_13
5. Emmerich, W., Gruhn, V.: Funsoft nets: a petri-net based software process modeling language. In: Proceedings of IWSSD, pp. 175–184 (1991)
6. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88873-4_17
7. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009). https://doi.org/10.1007/b95112
8. de Leoni, M., Felli, P., Montali, M.: A holistic approach for soundness verification of decision-aware process models. In: Trujillo, J.C., et al. (eds.) ER 2018. LNCS, vol. 11157, pp. 219–235. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_17
9. Libkin, L.: Fixed point logics and complexity classes. Elements of Finite Model Theory. LNCS, vol. 7360. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07003-1
10. Milner, R.: Communication and Concurrency. Prentice-Hall Inc., Upper Saddle River (1989)
11. Montali, M., Rivkin, A.: Model checking Petri nets with names using data-centric dynamic systems. Formal Asp. Comput. **28**(4), 615–641 (2016)
12. Montali, M., Rivkin, A.: DB-nets: on the marriage of colored petri nets and relational databases. In: Koutny, M., Kleijn, J., Penczek, W. (eds.) Transactions on Petri Nets and Other Models of Concurrency XII. LNCS, vol. 10470, pp. 91–118. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-55862-1_5
13. Montali, M., Rivkin, A.: From DB-nets to coloured petri nets with priorities (extended version). Technical Report arXiv:1904.00058, arXiv.org (2019)
14. Oberweis, A., Sander, P.: Information system behavior specification by high level petri nets. ACM Trans. Inf. Syst. **14**(4), 380–420 (1996)
15. Reichert, M.: Process and data: two sides of the same coin? In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) OTM 2012. LNCS, vol. 7565, pp. 2–19. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_2
16. Ritter, D., Rinderle-Ma, S., Montali, M., Rivkin, A., Sinha, A.: Formalizing application integration patterns. pp. 11–20 (2018)
17. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability and complexity of petri nets with unordered data. Theor. Comput. Sci. **412**(34), 4439–4451 (2011)
18. Triebel, M., Sürmeli, J.: Homogeneous equations of algebraic petri nets. In: Proceedings of CONCUR, LNCS, pp. 1–14. Springer (2016)
19. Weitz, W.: SGML nets: integrating document and workflow modeling. In: Proceedings of HICSS, vol. 2, pp. 185–194 (1998)
20. Westergaard, M., Verbeek, H.M.W.E.: Efficient implementation of prioritized transitions for high-level Petri nets. In: Proceedings of PNSE, pp. 27–41 (2011)