Under consideration for publication in Formal Aspects of Computing

Model Checking Petri Nets with Names Using Data-Centric Dynamic Systems

Marco Montali, Andrey Rivkin

KRDB Research Centre for Knowledge and Data, Faculty of Computer Science, Free University of Bozen-Bolzano

Abstract. Petri nets with name creation and management (ν-PNs) have been recently introduced as an expressive model for dynamic (distributed) systems, whose dynamics are determined not only by how tokens flow in the system, but also by the pure names they carry. On the one hand, this extension makes the resulting nets strictly more expressive than P/T nets: they can be exploited to capture a plethora of interesting systems, such as distributed systems enriched with channels and name passing, service interaction with correlation mechanisms, and resource-constrained workflow nets that explicitly account for process instances. On the other hand, fundamental properties like coverability, termination and boundedness are decidable for ν-PNs. In this work, we go one step beyond the verification of such general properties, and provide decidability and undecidability results of model checking ν-PNs against variants of first-order μ-calculus, recently proposed in the area of data-aware process analysis. While this model checking problem is undecidable in the general case, decidability can be obtained by considering different forms of boundedness, which still give raise to an infinite-state transition system. We then ground our framework to tackle the problem of soundness checking over workflow nets enriched with explicit process instances and resources. Notably, our decidability results are obtained via a translation to data-centric dynamic systems, a recently devised framework for the formal specification and verification of data-aware business processes working over full-fledged relational databases with constraints. In this light, our results contribute to the cross-fertilization between the area of formal methods for concurrent systems and that of foundations of data-aware processes, which has not been extensively investigated so far.

1. Introduction

Formal modeling and analysis of complex, dynamic, and possibly distributed systems, such as interacting web services and (interorganizational) business processes, require to not only consider how the system components interoperate with each other in terms of their control- and message-flow, but also to properly incorporate the information they manipulate and exchange. This fundamental need has spawned an extensive research in different areas of computer science, such as formal methods, concurrent systems, knowledge representation, and database theory [Var05, Via09, CDGM13]. In the Petri nets literature, several variants of *colored* nets [JK09, vdAS11] have been proposed and thoroughly studied so as to enrich classical P/T

Correspondence and offprint requests to: Marco Montali, Andrey Rivkin, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy. e-mail: montali,rivkin@inf.unibz.it

nets with information that goes far beyond the control-flow dimension, and to augment modeling power by including data and resources. However, in the general case where color domains are infinite (such as strings or numbers), colored nets are Turing-complete [Pet80]. Consequently, virtually all reasoning tasks turn out to be undecidable. This motivated a research on restricted forms of colored Petri nets that mediate between expressiveness and decidability. Examples along this line are data nets [LNO$^+$08], object nets [KR04] and ν-PNs [RVdFE08, RVdFE11]. In this work, we concentrate on the ν-PN model, which enriches standard P/T nets with the possibility of creating and managing *pure names* [Nee89].

In ν-PNs, the evolution of a marked net is determined not only by the distribution of tokens over the net, but also by the pure names they carry. Special transitions are used to inject fresh names into the net. Such names can be compared for equality and inequality when firing transitions, and are propagated in the net by making transitions able to match input and output names. This name creation and management mechanism leads to a strict increase in the expressiveness of the resulting formalism, attested by the fact that while marking reachability is decidable for P/T nets, it becomes undecidable for ν-PNs [RVdFE08]. This increased expressivity has been exploited to capture a plethora of interesting dynamic systems, such as:

- Distributed systems enriched with channels and name passing, where pure names represent process identifiers [Nee89]. In this light, ν-PNs can be used to combine in a unique formalism the pure name-based approaches typically studied for π-calculi [HMM13], with the richness and compactness of Petri nets in modeling the control-flow perspective [vdA05].
- Cryptographic protocols (with a presence of active adversary and without cryptanalysis), which are using Spi calculus [AG97] – a version of π-calculus equipped with abstract cryptographic primitives – for representation and analysis, can be modeled with ν-PNs [RVdFE11], where typical Petri net approaches for protocol simulation [WM11, AK95] are enriched with names representing encryption keys and communication channels.
- Service interaction with correlation mechanisms, where names are used to identify and ultimately isolate different conversations established by the same services [DW08].
- Resource-constrained workflow nets, where names are used to represent different process instances [MR11]. Notably, in Section 2.4 we present a variant of the approach [MR11], where workflow nets are equipped with global resources and with an explicit notion of process instance with control-flow isolation; these nets are called *resource and instance-aware workflow nets* (RIAW-nets).

From the foundational point of view, the execution semantics of ν-PNs is given in terms of infinite-state transition systems, whose infinity arises from the fact that unboundedly many names could be injected and evolved. In spite of this source of infinity, it has been shown that such transition systems are strictly well-structured. As a consequence, fundamental properties like coverability, termination and boundedness are decidable for ν-PNs [RVdFE08, RVdFE11]. This departs from colored Petri nets, whose formal analysis typically require to adopt finite color domains [JK09].

In this work, we go one step beyond the verification of such general properties, and provide key decidability and undecidability results of model checking ν-PNs against rich temporal/dynamic properties that predicate about the evolution of names across states. In particular, we consider variants of first-order μ-calculus, borrowed from verification logics recently proposed in the area of data-aware process analysis [BHCDG$^+$13, BHCM$^+$13, CDGM13]. Notably, we show in Section 3.2 how these logics can capture a notion of dynamic soundness over RIAW-nets, lifting the standard notion of soundness for workflow nets, which has been extensively studied in the literature [vdA97, vHSSV05, vdAvHtH$^+$11, MR11].

Our approach is based on the identification of classes of ν-PNs that, while still being inherently infinite-state, can be effectively verified against such logics through a faithful reduction to standard finite-state temporal model checking of propositional μ-calculus [Eme96], leveraging the construction of a robust, finite-state abstract transition system that only depends on the input net (and not on the specific property of interest). In principle, this would allow one to exploit state-of-the-art, conventional model checkers for the verification of the infinite-state transition systems generated by these ν-PNs. To this aim, we consider different forms of boundedness over the usage of names and the repetitions of the same name.

To prove decidability and show how to construct the aforementioned finite-state abstractions, we resort to a different line of research where pure names are central. In fact, pure names have not only been investigated in the area of distributed systems, process algebras, and Petri nets, but explicitly or implicitly also in databases and query languages [CH80], name binding for programming languages [GP02], and nominal sets[1]

[1] Considering in particular those with equality symmetry.

[BBKL12]. In particular, we leverage the flourishing literature on verification of database-driven dynamic systems and data-aware business processes [BLP12, BHCDG+13, BHCM+13, CDGMP13, BST13], where pure names are used to model data objects that are not important per sè, but because of the relations they form with other objects (this is the case, e.g., for primary keys). We refer the interested reader to [CDGM13] for a survey on this area.

Specifically, we adopt here the framework of data-centric dynamic systems (DCDSs), which has been recently proposed as a rich framework for the formal specification and verification of business processes working over full-fledged relational database with constraints [BHCDG+13]. We provide a translation mechanism from ν-PNs to DCDSs, and exploit such a translation to systematically transfer properties and (un)decidability results from one setting to the other. Thanks to this correspondence, we consequently obtain as a side result an interesting class of DCDSs for which boundedness is decidable, a property that has been recently shown to be undecidable even for very restricted forms of DCDSs [BHCDM14]. In this light, our results contribute to the cross-fertilization between the area of formal methods for concurrent systems and the foundations of data-aware processes, which has not been extensively investigated so far.

2. ν-Petri Nets

ν-Petri nets (ν-PNs for short) are an extension of P/T nets [Rei13] with pure name creation and management [RVdFE08, RVdFE11]. In a ν-PN, each token carries a name. Fresh names can be dynamically created, and transitions may impose matching restrictions on token names to fire. In this section, we formally introduce ν-PNs following the formulation in [RVdFE11], and then provide an execution semantics that refines the one in [RVdFE11]. We then discuss different notions of boundedness for ν-PNs. Finally, we show how ν-PNs can be used to model resource-constrained workflow nets with explicit process instance identifiers.

We start with some preliminary definitions. We consider the standard notion of *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 $S(a) = n$. The *support* of S is the set of elements that appear in S at least once: $supp(S) = \{a \in A \mid S(a) > 0\}$. We also consider the usual operations on multisets: given $S_1, S_2 \in A^\oplus$,

- $S_1 \subseteq S_2$ (resp., $S_1 \subset S_2$) if, for each $a \in A$, $S_1(a) \leq S_2(a)$ (resp., $S_1(a) < S_2(a)$);
- $S_1 + S_2 = \{a^n \mid a \in A \text{ and } n = S_1(a) + S_2(a)\}$;
- if $S_1 \subseteq S_2$, $S_2 - S_1 = \{a^n \mid a \in A \text{ and } n = S_2(a) - S_1(a)\}$.

2.1. Formal Definition

Name management in ν-PNs is formalized by adding to ordinary tokens also a special form of colored tokens, each one carrying a name taken from a countable, unordered infinite set Id of names. To define transitions that are affected not only by the presence of tokens in certain places, but also by the names carried by such tokens, all arcs in the net are labelled with matching variables, taken from a countably infinite set Var. Furthermore, we want to model the creation of new, fresh names, i.e., names that are not currently employed by any of the existing tokens. To do so, a special subset $\Upsilon \subset Var$ of variables is introduced, with the constraint that a variable $\nu \in \Upsilon$ can only match with a fresh name, that is, a name not currently present in the net. To preserve the standard P/T net notation in ν-PNs, we support also the usage of black, uncolored tokens. Concretely, this is done by fixing a special name $\bullet \in Id$ for such tokens, and by correspondingly introducing a special variable $\epsilon \in Var$ that matches with black tokens only. With these notions at hand, we can define ν-PNs formally.

Definition 2.1. A *ν-PN* is a tuple $N = \langle P, T, F \rangle$, where:

- P is a finite set of *places*;
- T is a finite set of *transitions*, disjoint from P;
- $F : (P \times T) \cup (T \times P) \to Var^\oplus$ is a *flow relation* such that, for every $t \in T$, we have *(i)* $\Upsilon \cap pre(t) = \emptyset$, and *(ii)* $post(t) \setminus \Upsilon \subseteq pre(t)$, where $pre(t) = \bigcup_{p \in P} supp(F(p,t))$ and $post(t) = \bigcup_{p \in P} supp(F(t,p))$;
- for every transition $t \in T$, ${}^\bullet t = \{y \in P \mid (y, x) \in F\}$ is the *pre-set* of t and $t^\bullet = \{y \in P \mid (x, y) \in F\}$ is the *post-set* of t (similarly for places). ■

The first condition for the flow relation indicates that new name variables cannot be associated to arcs that

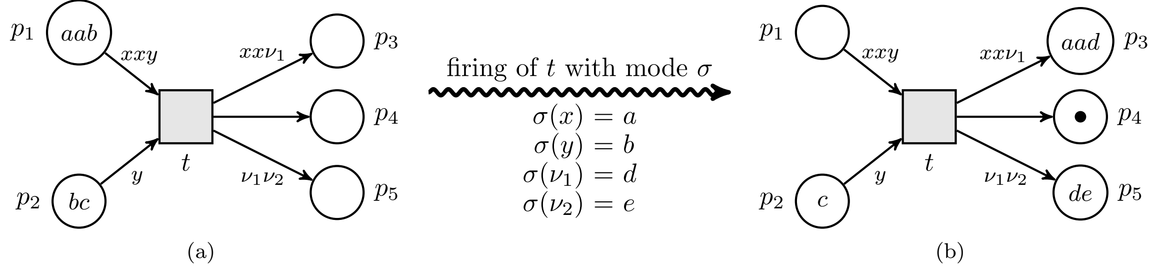

Fig. 1. A marked ν-Petri net 1(a), and the marked net 1(b) resulting from the firing of a transition with a given mode

go from a place to a transition; in fact, by definition, a variable in Υ cannot match with any name present in the net. The second condition expresses instead that arcs that go from a transition t to a place can be decorated with fresh name variables and/or variables that appear in one of the incoming arcs pointing to t; the former case denotes the ability of t of generating new names upon firing, whereas the latter case models the matching between names of tokens consumed by t with names of tokens produced by t upon firing.

The usual notion of marking in Petri nets is suitably extended for ν-PNs so as to assign a name to each of the tokens present in the net.

Definition 2.2. A *marking* m over ν-PN $N = \langle P, T, F \rangle$ is a function $m : P \rightarrow Id^{\oplus}$. A *marked* ν*-PN* $\overline{N}$ is a tuple $\langle P, T, F, m \rangle$, where $N = \langle P, T, F \rangle$ is a ν-PN, and m is a marking over N. ∎

Given a place $p \in P$, $m(p)$ denotes the multiset of names assigned to p by m, and $Id(m)$ denotes the overall set of names present in m: $Id(m) = \bigcup_{p \in P} supp(m(p))$. Furthermore, given a place $p \in P$ and a name $a \in Id$, $m(p)(a)$ denotes the number of times a is assigned to p by m. We now discuss how the standard notion of firing in P/T nets is suitably extended to deal with names in ν-PN. As customary in colored Petri nets, the firing of a transition $t \in T$ is defined w.r.t. a *mode* $\sigma : Var(t) \rightarrow Id$, where $Var(t) = pre(t) \cup post(t)$. The mode σ assigns a specific name to each of the variables that annotate the input or output arcs of t. However, to properly fire t, the mode σ must satisfy the matching conditions expressed by the variables involved in the firing, in particular guaranteeing freshness of names bound to variable in Υ, and the equality of names bound to different repeatitions of the same variable.

Definition 2.3. Consider a ν-PN $N = \langle P, T, F \rangle$, a transition $t \in T$, a marking m over N, and a mode σ for t. We say that t is *enabled in* m *with mode* σ , written $m[t, \sigma\rangle$, if:
1. the mode agrees with the distribution of named tokens in m, i.e., $\sigma(F(p, t)) \subseteq m(p)$ for every $p \in P$;
2. the mode assigns fresh names to the new name variables attached to the output arcs of t, i.e., $\sigma(\nu) \notin Id(m)$ for every $\nu \in \Upsilon \cap Var(t)$.

Given two markings m and m' over N, a transition $t \in T$, and a mode σ for t, we say that t *fires with mode* σ *in* m *producing* m', written $m[t, \sigma\rangle m'$ if:
1. t is enabled in m with mode σ;
2. m' is such that for every $p \in P$, we have $m'(p) = (m(p) - \sigma(F(p, t))) + \sigma(F(t, p))$. ∎

Example 2.1. Figure 1 shows how a marked ν-PN can be represented graphically, and illustrates the effect of firing a transition with a given mode. Places, transitions, and arcs are represented using the standard notation. Markings are represented, as usual, by distributing tokens over the corresponding places. Black tokens are depicted in the standard way, while tokens carrying names are represented using the names themselves. For example, the marking of the net in Figure 1(a) indicates that p_1 contains three tokens, two of which carry name a and one of which carries name b, while p_2 contains two tokens, one carrying name b and the other carrying name c. Arcs have associated variables, representing the matching conditions for tokens. In accordance with the formal definition, different variables match only with different names. In this light, transition t in Figure 1(a) can fire only when the following conditions hold:
- p_1 contains at least two tokens carrying the same name, and a third token with a different name;
- p_2 contains at least one token whose name correponds to the one of the third token in p_1 mentioned before.

The effect of firing t is the following:

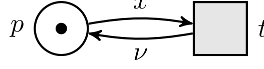

Fig. 2. A simple marked ν-PN whose unique transition t updates the content of place p by replacing the current token with a token that carries a different (locally fresh) name; the net is width- and depth-bounded but not run-bounded

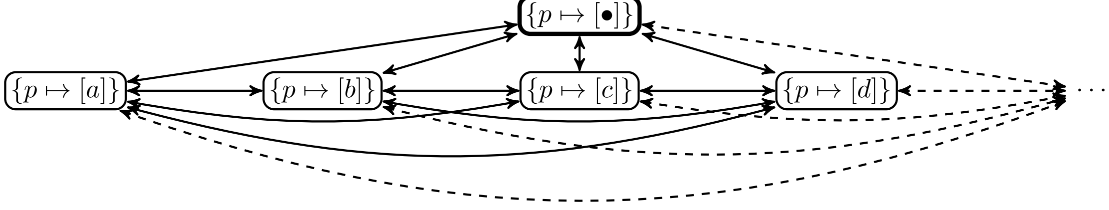

Fig. 3. Infinite-state MTS representing the execution semantics of the marked ν-PN of Figure 2; even if not explicitly shown, each transition is labelled with t

- the two selected tokens in p_1 that share the same name are forwarded to p_3 (this is represented by the xx inscription, common to the two involved arcs);
- a fresh name is generated, and two tokens carrying that name are respectively inserted intop_3 and p_5 (cf. the ν_1 inscription);
- another fresh name is generated, and a token carrying that name is inserted into p_5 (cf. the ν_2 inscription);
- a black token is inserted into p_4 (the corresponding arc does not contain any explicit variable, which means that its inscription is implicitly the special variable ϵ).

The marked net of Figure 1(b) shows the effect of firing t with the specified mode. ∎

2.2. Execution Semantics

The execution semantics of a marked ν-PN $\overline{N}$ is defined in terms of a possibly infinite-state transition system, whose states are labeled by reachable markings, and where each transition corresponds to the firing of a transition in N with a given mode.

Formally, the execution semantics of a marked ν-PN $\overline{N} = \langle P, T, F, m_0 \rangle$ is a *marking-labeled transition system* (MTS) $\Gamma_{\overline{N}} = \langle M, m_0, \rightarrow \rangle$, where:

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

A *run* τ over $\Gamma_{\overline{N}}$ is a possibly infinite sequence of markings $m_0, m_1, \cdots$ where, for every m_i, m_{i+1} in τ, there exists $t \in T$ s.t. $m_i \xrightarrow{t} m_{i+1}$.

Observe that the definition of $\Gamma_{\overline{N}}$ does not coincide with that of ν-PN reachability graph in [RVdFE11]. In fact, such a reachability graph does not faithfully describe all possible executions of $\overline{N}$, but incorporate a *name abstraction* technique, called α-equivalence [RVdFE11], which collapses different states into a single state. While this abstraction preserves reachability, it does not preserves temporal properties that relate names over time (cf. Section 3).

Example 2.2. Consider the marked ν-PN in Figure 2. The MTS capturing the execution semantics of such net is shown in Figure 3. Intuitively, each marking contains just one token placed in p. Transition t is always enabled and the effect of its firing is to replace the name of such a token with a different (i.e., "locally fresh") name. ∎

2.3. Boundedness Properties

In standard P/T nets, the only possible source of unboundedness is the possibility of producing unboundedly many tokens during the evolution of the net. On the contrary, ν-PNs may be unbounded for different reasons, which depend on the one hand on the number of tokens, and on the other hand on the number of names. Specifically, a marked ν-PN $\overline{N}$, $\Gamma_{\overline{N}}$ could give raise to an infinite-state MTS for different reasons:
- *width-unboundedness* [RVdFE11], i.e., accumulation of unboundedly many names in a state;
- *depth-unboundedness* [RVdFE11], i.e., accumulation of unboundedly many tokens with the same name;
- *run-unboundedness* (adapted from [BHCDG+13]), i.e., presence of unboundedly many names along a run.

We mirror this intuition by formally defining these three possible sources of (un)boundedness.

Definition 2.4. A marked ν-PN $\overline{N} = \langle P, T, F, m_0 \rangle$ with MTS $\Gamma_{\overline{N}} = \langle M, m_0, \rightarrow \rangle$ is:
- *width-bounded* if there is $n \in \mathbb{N}$ such that $|Id(m)| \leq n$ for every $m \in M$;
- *depth-bounded* if there is $n \in \mathbb{N}$ such that $m(p)(a) \leq n$ for every $m \in M$, name $a \in Id(m)$ and place $p \in P$;
- *run-bounded* if it is *depth-bounded* and there is $n \in \mathbb{N}$ such that, for every run τ over $\Gamma_{\overline{N}}$, we have $|\bigcup_{m \text{ in } \tau} Id(m)| \leq n$. ■

While it is clear that run-boundedness implies width-boundedness, the converse does not hold. For example, there are ν-PNs (such as that of Example 2.3) that are width-bounded and depth-bounded, but not run-bounded.

Example 2.3. Consider the marked ν-PN in Figure 2. The net is width- and depth-bounded: as witnessed by Figure 3, each marking in its MTS contains exactly one token. However, it is not run-bounded. In fact, there are runs in which only boundedly many names are introduced (due to the reintroduction of locally fresh names that were already encountered in the past), but there are also runs in which unboundedly many globally fresh names are introduced (e.g., there is an infinite run in which no name is repeated twice). ■

Observe that in the original formulation of ν-PN reachability graph [RVdFE11], run-unboundedness does not appear as a source of unboundedness, again due to the implicit abstraction induced by α-equivalence. Even more, width-boundedness and depth-boundedness characterize boundedness in the sense of [RVdFE11], guaranteeing that such a reachability graph is finite-state. However, according to the natural execution semantics introduced in Section 2.2, $\Gamma_{\overline{N}}$ can become infinite-state even in those cases where $\overline{N}$ is width- and depth-bounded but not run-bounded. This implies that, in our setting, boundedness of $\overline{N}$ does not imply that $\Gamma_{\overline{N}}$ is finite-state. Still, it guarantees that each marking of $\Gamma_{\overline{N}}$ contains a bounded number of tokens and names. In this light, it corresponds to a variant of the notion of *state-boundedness* as defined in [BHCDG+13]. Intuitively, a state-bounded system can still exhibit runs in which unboundedly many different pieces of information are encountered, provided that they do not accumulate in a single state. We precisely formalize this intuition.

Definition 2.5. A marked ν-PN $\overline{N} = \langle P, T, F, m_0 \rangle$ with MTS $\Gamma_{\overline{N}} = \langle M, m_0, \rightarrow \rangle$ is *state-bounded* if there is $n \in \mathbb{N}$ s.t., for each $m \in M$, we have $\sum_{p \in P, a \in Id} m(p)(a) \leq n$; ■

Lemma 2.1. A marked ν-PN is state-bounded if and only if it is width-bounded and depth-bounded.

Proof. Let $\overline{N} = \langle P, T, F, m_0 \rangle$ be a ν-PN with MTS $\Gamma_{\overline{N}} = \langle M, m_0, \rightarrow \rangle$. We prove the two directions separately.

$\Rightarrow$
Suppose that $\overline{N}$ is state-bounded. From Definition 2.5, we know that there exists $n \in \mathbb{N}$ such that, for each $m \in M$, we have $\sum_{p \in P, a \in Id} m(p)(a) \leq n$. Furthermore, for each $m \in M$, the following inequalities hold:

$$|Id(m)| \leq \sum_{p \in P, a \in Id} m(p)(a) \leq n \qquad max\{m(p)(a) \mid p \in P, a \in Id(m)\} \leq \sum_{p \in P, a \in Id} m(p)(a) \leq n$$

This attests that n is also a width- and depth-bound for $\overline{N}$.

$\Leftarrow$
If $\overline{N}$ is width- and depth-bounded, by Definition 2.4 there exist $n_1, n_2 \in \mathbb{N}$ such that, for every marking

$m \in M$, we have that $|Id(m)| \leq n_1$, and $m(p)(a) \leq n_2$ for each $p \in P$ and $a \in Id(m)$. Hence, we get that

$$\sum_{p \in P, a \in Id} m(p)(a) \leq |Id(m)| \cdot max\{m(p)(a) \mid p \in P, a \in Id(m)\} \leq n_1 \cdot n_2$$

which implies that $n_1 \cdot n_2$ is a state-bound for $\overline{N}$. □

The next theorem employs one of the crucial results in [RVdFE11] to show that state-boundedness is a decidable property for ν-PNs. In the light of the strong undecidability results for state-boundedness reported in [BHCDM14], this witnesses that ν-PNs are an interesting class of "data-aware" dynamic systems for which state-boundedness can be effectively checked.

Theorem 2.1. Checking whether a marked ν-PN is state-bounded is decidable.

Proof. From Lemma 2.1, we know that a marked ν-PN is state-bounded if and only if it is both width- and depth-bounded. Decidability then is directly obtained from Proposition 3 in [RVdFE11]. □

2.4. Modeling Workflow Nets with Resources and Explicit Process Instances

To show the modeling power of ν-PNs, we use them to capture workflow nets enriched with (global) resources and explicit process instances. Workflow nets are the de-facto reference model for capturing the control-flow dimension and the processing of tasks in workflows [vdA97, vdAvHtH+11]. Intuitively, they are P/T nets with two special places, called input and output places, which respectively mark the entry and the exit points of the various executions of the workflow, i.e., of its process instances. All the other elements of the net are then required to rely in a path connecting such two special places.

As such, workflow nets only capture the control-flow dimension of a workflow. Building on [vHSV06], we enrich this dimension with global resources, consumed and released by tasks. However, differently from [vHSV06], we leverage the name creation and management capabilities of ν-PNs, by explicitly modeling the notion of process instance, which is typically left implicitly in the workflow net literature. In fact, properties of workflow nets are always studied by making the implicit assumption that each process instance evolves in isolation from the others. This implicit assumption is then exploited to study formal properties of workflow nets by simply inserting a single token into the input place, and consequently analyzing the evolution of a single process instance as a prototype of all process instances.

Thanks to names in ν-PNs, we can instead fully model the generation of (possibly unboundendly many) process instances, each one associated with a distinct name (representing the instance distinguished identifier). We can then use name matching by ensuring that different instances do not interfere with each other via control-flow constructs, but only indirectly due to resource contention. This notion of isolation, which is customary in standard workflows, makes our approach different from that of [MR11], where process instances may interact directly. To provide a formal basis for all such intuitions, we proceed in two steps. We first define a notion of instance-aware workflow net, enriching the standard definition in [vHSV06, vdAvHtH+11].

Definition 2.6. An *instance-aware workflow net (IAW-net)* is a ν-PN $N = \langle P, T, F \rangle$, where:

1. Only one fresh variable $\nu \in \Upsilon$ and one non-fresh variable $x \in Var \setminus \Upsilon$ are employed.
2. N contains a special transition $t_g \in T$, called *instance generator*, and a special place $i \in P$, called *input place*, such that:
   - $F(t_g, i) = \{\nu\}$;
   - $F(t_g, p) = \emptyset$ for every $p \in P \setminus \{i\}$, and $F(p, t_g) = \emptyset$ for every $p \in P$;
   - $F(t, i) = \emptyset$ for every $t \in T \setminus \{t_g\}$.
3. N contains a special transition $t_r \in T$, called *instance removal*, and a special place $o \in P$, called *output place*, such that:
   - $F(o, t_r) = \{x\}$;
   - $F(p, t_r) = \emptyset$ for every $p \in P \setminus \{o\}$, and $F(t_r, p) = \emptyset$ for every $p \in P$;
   - $F(o, t) = \emptyset$ for every $t \in T \setminus \{t_r\}$.
4. For every "normal" place $p \in P \setminus \{i, o\}$, and for every "normal" transition $t \in T \setminus \{t_g, t_r\}$, we have:
   - either $F(p, t) = \emptyset$, or $F(p, t) = \{x\}$;
   - either $F(t, p) = \emptyset$, or $F(t, p) = \{x\}$.
5. For every element $n \in P \cup (T \setminus t_g, t_r)$, there exists a path from i to n and from n to o. ■

Intuitively, point (1) states that an IAW-net employs a fresh variable to generate new process instances, and a non-fresh variable to match an already created process instance. Point (2) states that the net contains an emitter transition t_g that injects new process instances in the special input place i. Point (3) states that the net contains a sink transition t_r that removes completed process instances from the special output place o. Point (4) captures instead the "instance isolation" principle illustrated above, by requiring that all normal transitions in the net consume and produce tokens with the same name, i.e., associated to the same process instance. Finally, point (5) tackles the usual requirement in workflow nets, which states that every task is part of a workflow that starts in i and ends in o.

Clearly, IAW-nets are always state-unbounded: the emitter, instance generator transition is always enabled, and when fired it injects a new process instance, associated to its own fresh identifier. The situation radically changes, and becomes more interesting, when also resources are considered.

Definition 2.7. A *resource and instance-aware workflow net (RIAW-net)* is a triple $\langle N, P_R, F_R \rangle$, where:
- $N = \langle P, T, F \rangle$ is an IAW-net;
- P_R is a set of additional, *resource places* modeling the different kinds of resources present in the system, which are assumed to carry only black tokens;
- $F_R : (P_R \times T) \cup (T \times P_R) \to \{\epsilon\}^{\oplus}$ is a *resource flow relation*, linking resource places to transitions. ■

A tuple (p_R, t, ϵ^k) in F_R indicates that, for t to be fired, k resources of type p_R have to be provided. Similarly, a tuple (t, p_R, ϵ^j) in F_R indicates that, when t is fired, j resources of type p_R are released.

In this light, an initialized RIAW-net is a net that indicates how many resources are available for each resource type (i.e., how many black tokens have to be inserted in the different resource places).

Definition 2.8. An *initialized RIAW-net* is a tuple $\langle N, P_R, F_R, \mathit{rdist} \rangle$, where:
- $\langle N, P_R, F_R \rangle$ is a RIAW-net;
- $\mathit{rdist} : P_r \to \{\epsilon\}^{\oplus}$ is a function that distributes black tokens over resource places. ■

Clearly, a RIAW-net $\langle N, P_R, F_R \rangle$ with $N = \langle P, T, F \rangle$ can be understood as a standard ν-PN that suitably incorporates the resource information into N. More precisely, $\langle N, P_R, F_R \rangle$ corresponds to the ν-PN $N_R = \langle P \cup P_R, T, F \cup F_R \rangle$. In this light, an initialized RIAW-net $\langle N, P_R, F_R, \mathit{rdist} \rangle$ with $N = \langle P, T, F \rangle$ corresponds to the marked ν-PN $\langle P \cup P_R, T, F \cup F_R, \mathit{rdist} \rangle$.

We now present an example that illustrates the powerful modeling capabilities of RIAW-nets.

Example 2.4. Consider the workflow adopted by an HW assistance company, so as to support the company employees in fixing broken devices of customers. There are three resource types involved in the workflow from the company side:
- *HW experts*, providing their expertise in understanding and reporting about the nature of the problem, dealing in some cases with the actual repair;
- *Shipping clerks*, managing the shipment of the device whenever the reparation has to be handled externally;
- *Secretaries*, in charge of printing the produced reports and receipts.

A process instance is created whenever a customer arrives and asks for support. A (front-end) HW expert is in charge of welcoming the customer, and of following the case of the customer end-to-end. First, the expert performs an initial check so as to understand the nature of the problem. In particular, two outcomes may arise: either the HW expert understands that the problem can be solved in-house, or she decides that the repair must be handled externally. In the first case, the actual fixing is done by a second HW expert working in the back-end, who is not only responsible of the actual repair, but is also in charge of writing a short summary explaining the problem and the repair. The summary is then printed by a secretary. In the second case, two different tasks have to be performed, independently from each other. The first task is again under the responsibility of the front-end HW expert, who writes a detailed report about the nature of the problem. At the same time, a shipping clerk becomes in charge of the device expedition, which consists in the preparation of a package (and the corresponding documents). Once the report has been written and the package prepared, the shipping clerk assembles the entire package (which will be then automatically sent to the post office). Finally, the secretary prints a receipt (which includes also an expedition code to track where the device is currently located), and the process instance terminates.

Figure 4 shows how this workflow can be captured as a RIAW-net (i.e., a ν-PN), considering not only the acceptable orderings among tasks, but also their interplay with resources. ■

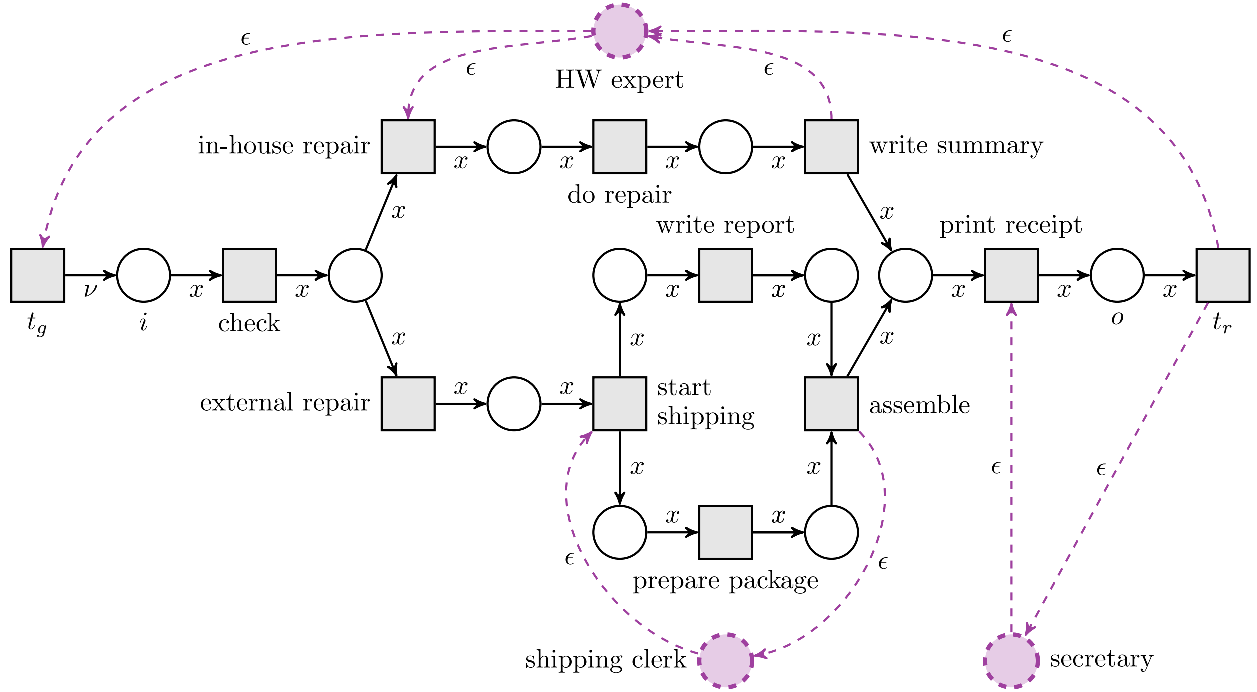

Fig. 4. RIAW-net of Example 2.4; control-flow information (i.e., the inner IAW-net) is shown in solid black, whereas resource-related elements are depicted in dashed violet

It is worth noting that state-boundedness is a very important property for RIAW-nets, which can be actually checked thanks to Theorem 2.1. In fact, a state-unbounded RIAW-net is a net that either produces more resources than those that are consumed, and/or allows unboundedly many process instances to be executed in parallel. The first case is undesirable because in reality it is never possible to produce unboundedly many resources. The second case is undesirable because it witnesses that resources are not properly intertwined with process instances, i.e., there are tasks that can be executed without any dedicated resources. Notice, however, that once a RIAW-net is detected to be state-unbounded, it is not possible to carry on a further finer-grained analysis, so as, e.g., to determine whether the source of unboundedness is related to resources or process instances. In fact, this would require to check state-boundedness of specific places present in the net, a reasoning task that has been shown to be undecidable for ν-PNs [RVdFE11].

Even more interesting is the problem of checking *soundness* over this extended class of workflow nets, building on the line of research in [vdA97, vHSSV05, vdAvHtH$^+$11, MR11]. We come back to this problem in Section 3.

We close this section by discussing an example that enriches standard workflow nets, showing how names can be exploited not only to handle process instances and their isolation, but also to capture a form of abstract data creation and comparison. This gives a further glimpse on the versatility and modeling power of ν-PNs.

Example 2.5. We consider the grain harvesting ν-PN shown in Figure 5. It consists of a standard workflow net that employs name creation to attach abstract data to the tokens representing the execution of process threads, and that compares such names so as to determine how to proceed next.

A process instance in this context represents the management of a specific harvesting plan. A crop expert analyzes the plan and sets up a strategy, which results in a decision about the target granularity of the grain to be obtained. This granularity, generated as an outcome of the *set up strategy* task through name creation, is given as an input to the task of mechanical harvesting, so as to suitably set the working parameters of the machine. In parallel, also manual harvesting is executed, so as to ensure that all areas within the crop field are suitably tackled. The manually obtained grain is then analyzed, so as to determine its granularity. The result of this analysis is again modeled by using name creation within the net.

The desired and manual granularities are then compared for equality. If they match, then the grains

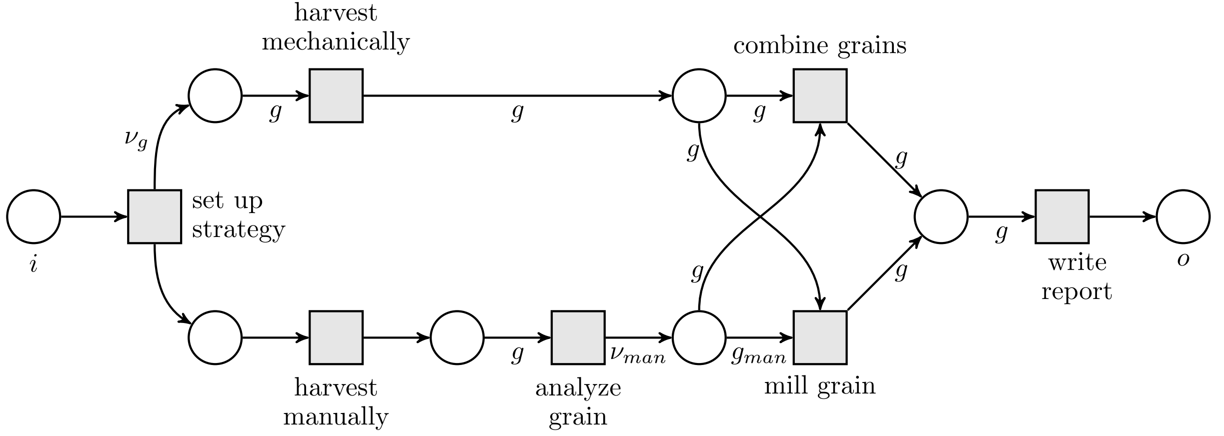

Fig. 5. ν-PN that enriches a workflow net with abstract data creation and comparison

obtained mechanically and manually can be directly combined. If instead they are different, the manual grain is subject to mill, so as to make its granularity homogeneous with the planned one. Finally, a report is written, in which it is confirmed that the obtained grain is of the desired granularity.

■

3. Model Checking ν-PNs

We now turn to the formal verification of ν-PNs. We are in particular interested in model checking ν-PNs against suitable temporal properties of interest, declaratively capturing the (un)desired behaviors of the net under study. However, differently from the standard setting of finite-state model checking, in ν-PNs also names and their evolution over time have to be properly considered. In particular, to specify temporal properties of interest, the logic used for verification must provide support for: *(i)* temporal operators to express dynamics of the net; *(ii)* first-order (FO) formulae to query the local states of the system, i.e., to inspect markings and retrieve the corresponding token names; *(iii)* FO quantification across states, so as to relate names in different moments of the system, and compare them for equality and inequality.

3.1. The $\mu\mathcal{L}_A^N$ Logic

To answer the aforementioned requirements, we resort to a first-order variant of the μ-calculus, one of the most well-established branching-time temporal logics [Eme96]. μ-calculus combines local temporal operators, which connect the current state with its immediate successors, with least and greatest fixpoints that predicate over states that are arbitrarily far away from the current one. To inspect the markings associated to the states of the system, we extend μ-calculus with queries that test whether a place contain (at least/at most/exactly) a certain number of tokens, possibly matching a given name. Furthermore, formulae can quantify over the *active* token names (i.e., names present in the current marking), and use such names later on in the future, providing a form of name correlation across system states. This is crucial to express properties about the evolution of tokens that maintain their identity in terms of name. We call the resulting logic $\mu\mathcal{L}_A^N$.

Definition 3.1. Given a marked ν-PN $\overline{N} = \langle P, T, F, m_0 \rangle$, a $\mu\mathcal{L}_A^N$ formula Φ over $\overline{N}$ is defined as:

$$\Phi ::= \mathit{true} | \, Z \, | \, [\#p \leq c] \, | \, [\#p(x) \leq c] \, | \, x = y \, | \, \Phi_1 \wedge \Phi_2 \, | \, \neg\Psi \, | \, \exists x.\text{LIVE}(x) \wedge \Psi \, | \, \langle - \rangle \Psi \, | \, \mu Z.\Psi$$

where $p \in P$, $c \in \mathbb{N}$, x is either a variable or a constant name from $Id(m_0) \cup \{\bullet\}$, and Z is a second order predicate variable of arity 0, used to quantify over the states of the system. $\mu Z.\Phi$ is a least fixpoint formula whose bounding formula Φ may contain free individual variables. Such variables act as parameters of the fixpoint, that is, the value of the fixpoint is obtained only after having fixed an assignment for such free variables (cf. [Lib04]).

We make use of the following abbreviations: $[\#p \circledast c]$ and $[\#p(x) \circledast c]$, where $\circledast \in \{>, \geq, =, <\}$; $\forall x.\textsc{live}(x) \rightarrow \Psi = \neg(\exists x.\textsc{live}(x) \wedge \neg\Psi)$; $\Phi_2 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$; $[-]\Psi = \neg\langle - \rangle\neg\Psi$; $\nu Z.\Psi = \neg\mu.\neg\Psi\,[Z/\neg Z]$. ■

The intuitive meaning of such formulae is:

- $[\#p \leq c]$ is true if the overall amount of tokens assigned by the current marking to p does not exceed c;
- $[\#p(x) \leq c]$ is true if the overall amount of tokens matching name x and assigned by the current marking to p does not exceed c;
- $\textsc{live}(x)$ is true if name x is a "live" name, i.e., a name present in the current marking;
- $\langle - \rangle\Psi$ is true if there exists a successor marking in which Ψ holds;
- $[-]\Psi$ is true if in all successor markings, Ψ holds;
- $\mu Z.\Psi$ and $\nu Z.\Psi$ respectively represent the least and greatest fixpoint operator.

As usual in the μ-calculus, for fixpoints we require the *syntactic monotonicity* of Ψ w.r.t. Z, that is, every occurrence of the variable Z in Ψ must be within the scope of an even number of negation signs. This guarantees that the least and greatest fixpoints always exist.

Formally, a $\mu\mathcal{L}_A^N$ formula Φ over $\overline{N} = \langle P, T, F, m_0 \rangle$ is interpreted over the MTS $\Gamma_{\overline{N}} = \langle \Sigma, m_0, \rightarrow \rangle$. Since Φ can contain both individual and predicate free variables, the semantics uses an individual variable valuation v mapping individual variables to names in Id, and a predicate variable valuation V that, once fixed v, determines a v-dependent mapping of the predicate variables Z to sets of states in Σ. More in details, given a formula Φ with free individual variables $\vec{x}$ and free predicate variables $\vec{Z}$, and given an individual variable valuation v for $\vec{x}$ and a predicate variable valuation V for $\vec{Z}$, Φ is evaluated as follows. First, v is applied to Φ, grounding $\vec{x}$ to corresponding actual names according to v. Second, V is applied to the so-obtained, partially grounded formula, mapping $\vec{Z}$ to corresponding sets of states according to V. The fully grounded formula is then evaluated, returning those states in which it evaluates to true.

Formally, the semantics of $\mu\mathcal{L}_A^N$ formulae is defined through an *extension function* $||\cdot||_{v,V}^{\Gamma_{\overline{N}}}$ that maps a formula to the subset of Σ in which the formula holds. The extension function is inductively defined as follows:

$$
\begin{aligned}
||\top||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \Sigma \\
||Z||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \mathcal{V}v(Z) \\
||[\#p \leq c]||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \{m \in \Sigma \mid m(p) \leq c\} \\
||[\#p(x) \leq c]||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \{m \in \Sigma \mid m(p)(xv) \leq c\} \\
||\textsc{live}(x)||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \{m \in \Sigma \mid x/d \in v \text{ implies } d \in Id(m)\} \\
||x = y||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \{m \in \Sigma \mid x/d \in v \text{ if and only if } y/d \in v\} \\
||\Phi_1 \wedge \Phi_2||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= ||\Phi_1||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} \cap ||\Phi_2||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} \\
||\neg F||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= X \setminus ||F||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} \\
||\exists x.\Phi||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \{m \in \Sigma \mid \exists a \in Id \text{ s.t. } m \in ||\Phi||_{\mathcal{V},v[x/a]}^{\Gamma_{\overline{N}}}\} \\
||\langle - \rangle\Phi||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \{m \in \Sigma \mid \exists m' \in \Sigma \text{ s.t. } m \rightarrow m' \text{ and } m' \in ||\Phi||_{\mathcal{V},v}^{\Gamma_{\overline{N}}}\} \\
||\mu Z.\Phi||_{\mathcal{V},v}^{\Gamma_{\overline{N}}} &= \bigcap\{Y \subseteq \Sigma \mid ||\Phi||_{v,\mathcal{V}[Z/Y]}^{\Gamma_{\overline{N}}} \subseteq Y\}
\end{aligned}
$$

When Φ is a closed formula, its valuation does not depend neither on V nor on v. Hence, we denote its extension by simply using $||\Phi||^{\Gamma_{\overline{N}}}$. Given a closed $\mu\mathcal{L}_A^N$ property Φ and a marked ν-PN $\overline{N} = \langle P, T, F, m_0 \rangle$, we say that $\overline{N}$ *verifies* Φ, written $\overline{N} \models \Phi$, if $m_0 \in ||\Phi||^{\Gamma_{\overline{N}}}$.

Example 3.1. Formula $\Phi_r = \forall x.\textsc{live}(x) \rightarrow \mu Z.[\#p(x) \geq 1] \vee \langle - \rangle Z)$ is a reachability property stating that for every name present in the initial marking, there exist an evolution of the ν-PN in which that name eventually appears in place p. According to the semantics of $\mu\mathcal{L}_A^N$, the property is evaluated by substituting x in the fixpoint formula $\mu Z.[\#p(x) \geq 1] \vee \langle - \rangle Z)$ with every name present in the initial marking. For each such substitution, the fixpoint formula is evaluated. For example, if the initial marking contains name a, the formula with individual variable valuation mapping x to a becomes $\textsc{live}(\mathsf{a}) \rightarrow \mu Z.[\#p(\mathsf{a}) \geq 1] \vee \langle - \rangle Z) = \mu Z.[\#p(\mathsf{a}) \geq 1] \vee \langle - \rangle Z)$. Due to the least fixpoint semantics, this formula then corresponds to the set of states containing all those states in which a is contained in p, plus those states from which a state where a

is contained in p can be reached. In fact, we have that:

$$\mu Z.[\#p(\mathsf{a}) \geq 1] \vee \langle - \rangle Z) = [\#p(\mathsf{a}) \geq 1] \vee \langle - \rangle [\#p(\mathsf{a}) \geq 1] \vee \langle - \rangle \langle - \rangle [\#p(\mathsf{a}) \geq 1] \vee \ldots$$

Example 3.2. Formula $\Phi_d = \nu Z.(\forall x.[\#p(x) = 1] \rightarrow \nu Y.[\#p(x) = 0] \wedge \langle - \rangle Y) \wedge [-]Z$ holds in the ν-PN of Fig. 2. Φ_d expresses that it is always the case that, whenever place p contains exactly one token named x, then there exists an ongoing run in which x never reappears. By adopting the syntax of CTL, this property in fact corresponds to $\mathbf{AG}(\forall x.[\#p(x) = 1] \rightarrow \mathbf{EG}[\#p(x) = 0])$. Notice that, although LIVE is not explicitly used in Φ_d, the formula is still in $\mu\mathcal{L}_A^N$, since $[\#p(x) = 1]$ implies $\text{LIVE}(x)$. ■

It is worth noting that the model checking problem for first-order variants of temporal logics (including μ-calculus) has been subject of extensive research in the area of verification for data-aware business processes and services [CDGM13]. In this light, $\mu\mathcal{L}_A^N$ can be seen as a Petri net-variant fo the $\mu\mathcal{L}_A$ logic defined in [BHCDG+13]. The main difference between $\mu\mathcal{L}_A^N$ and $\mu\mathcal{L}_A$ is indeed that local queries in $\mu\mathcal{L}_A^N$ are specifically shaped to inspect the markings of a ν-PN, whereas $\mu\mathcal{L}_A$ exploits full FO (with active domain semantics) to query full-fledge relational databases forming the states of a generic data-aware process.

3.2. Formalizing Soundness of RIAW-Nets in $\mu\mathcal{L}_A^N$

We now lift the classical notion of soundness [vdA97, vHSSV05, vdAvHtH+11, MR11] to the case of RIAW nets, and show that it can be suitably formalized in $\mu\mathcal{L}_A^N$. Intuitively, a RIAW-net is sound if, for every process instance generated by its emitter transition and inserted into the input place, the following conditions hold:

1. *(eventual termination)* the process instance can always reach the output place;
2. *(proper termination)* once the process instance reaches the output place, it does so in a clean way, that is, no token labeled with the identifier of that instance is left behind in other places;
3. *(task executability)* for each task in the net, there is one possible evolution of the process instance where that task is indeed executed.

Let $\langle N, P_R, F_R \rangle$ be a RIAW-net with $N = \langle P, T, F \rangle$. We show how to formalize such three conditions in $\mu\mathcal{L}_A^N$. Suppose that we already have three unary $\mu\mathcal{L}_A^N$ formulae $\alpha(x)$, $\beta(x)$, and $\gamma(x)$, respectively formalizing eventual termination, proper termination, and task executability for process instance x. Soundness can be then formalized by imposing that these three properties always hold for x:

$$\Phi_{sound} = \nu Z.(\forall x.[\#i(x) = 1] \rightarrow \alpha(x) \wedge \beta(x) \wedge \gamma(x)) \wedge [-]Z$$

where:

- $\alpha(x) = \nu W.(\mu Y.([\#o(x) = 1]) \vee (\text{LIVE}(x) \wedge \langle - \rangle Y)) \wedge (\text{LIVE}(x) \wedge [-]W)$ captures eventual termination by modeling that, in every marking, there must exist a possible future continuation where x persists into the net until it eventually enters into the output place.
- $\beta(x) = \nu W.([\#o(x) = 1] \rightarrow \bigwedge_{p \in P}[\#p(x) = 0]) \wedge (\text{LIVE}(x) \wedge [-]W)$ captures proper termination by requiring that, whenever x is present in the output place, then it is not present in any other place.
- $\gamma(x)$ captures task executability by expressing that, for every task t in the RIAW-net, there exists a run that eventually reaches a situation where t is enabled, i.e., has at least one x-named token in each incoming place of the core IAW-net, and enough black tokens in the resource places attached to t:

$$\bigwedge_{t \in T} \mu Y.\Big(\bigwedge_{p \in \{p' \in P \mid F(p',t) = \{x\}\}} [\#p(x) \geq 1] \wedge \bigwedge_{\langle p_R, k \rangle \in \{\langle p'_R, k' \rangle \mid F(p',t) = \{\epsilon^{k'}\}\}} [\#p(\bullet) \geq k] \Big) \vee (\text{LIVE}(x) \wedge \langle - \rangle Y)$$

Thanks to this encoding, we can now check whether an initialized RIAW-net $\langle N, P_R, F_R, \mathit{rdist} \rangle$ with $N = \langle P, T, F \rangle$ is sound, by verifying whether $\Gamma_{\overline{N_R}} \models \Phi_{sound}$, where $\overline{N_R} = \langle P \cup P_R, T, F \cup T_f, \mathit{rdist} \rangle$ is the marked ν-PN obtained by incorporating resource information into N.

Notably, other properties of interest can be formulated using $\mu\mathcal{L}_A^N$ on top of RIAW-nets so as, e.g., to check that resources are properly managed, or that it is always possible to create, sometimes in the future, a new process instance (which implicitly means that the resources allocated to current process instance will be sooner or later released).

4. Negative Results

We prove here two key negative results related to the verification of $\mu\mathcal{L}_A^N$ properties over ν-PNs. Such results motivate the fine-grained analysis presented in Section 5.

Theorem 4.1. Verification of $\mu\mathcal{L}_A^N$ properties over ν-PNs is undecidable even when formulae do not make use of FO quantification, and only employ a single, constant name.

Proof. Consider marked ν-PNs that only use the special name $\bullet$ and the special variable ϵ. This class of ν-PNs coincides with classical P/T nets. Now consider $\mu\mathcal{L}_A^N$ properties that do not make use of FO quantification, and whose local predicates only employ the special name $\bullet$. This class of formulae resembles the propositional (branching) μ-calculus whose verification over P/T nets has been shown to be undecidable in [Esp94]. $\square$

To constrast this strong undecidability result, one can act along two directions. On the one hand, one could limit the analysis to classes of ν-PNs that enjoy different boundedness requirements (cf. Section 2.3). On the other hand, one could consider only a fragment of the very sophisticated $\mu\mathcal{L}_A^N$ logic. We operate along both directions.

As a warm-up, negative result, we study what happens when the ν-PN of interest enjoys state-boundedness, as defined in Section 2.3. We show that, in this case, there is no hope of achieving decidability of verification via a reduction to conventional finite-state model checking, in which the constructed finite-state transition system only depends on the input net (and not on the specific property of be verified).

Lemma 4.1. There exists a state-bounded, marked ν-PN $\overline{N}$ for which no faithful finite-state abstraction exists that satisfies the same $\mu\mathcal{L}_A^N$ properties of $\overline{N}$.

Proof. Let $\overline{N}$ be the ν-PN shown in Figure 2, and $\Gamma_{\overline{N}}$ be its MTS (shown in Figure 3. We have already argued that $\overline{N}$ is state-bounded. Now consider the formula Φ_d introduced in Example 3.2. It is easy to see that $\overline{N} \models \Phi_d$: every state s has always an outgoing infinite run τ that never comes back to that state, i.e., visits infinitely many states that all contain names different from that in s. On the other hand, there is no finite-state transition system $\Gamma^f_{\overline{N}}$ that satisfies Φ_d. In fact, since $\Gamma^f_{\overline{N}}$ is finite-state, every infinite run necessarily visits some states infinitely often, and consequently repeatedly encounters some names infinitely often. $\square$

It is interesting to notice that the proof of Lemma 4.1 also shows that the reachability graph of [RVdFE11] does not properly describe the execution semantics of ν-PNs in the light of the model checking problem we are interested in. In fact, the reachability graph of the marked ν-PN shown in Figure 2 has a finite number of states. The intuitive reason for this discrepancy is that $\mu\mathcal{L}_A^N$ properties can remember visited names inside a FO quantifier, and check the presence or absence of such names in markings that are arbitrarily far away from the state in which the quantifier was bound. Consequently, the abstraction of renaming names allowed by α-equivalence in [RVdFE11] does not preserve $\mu\mathcal{L}_A^N$ properties.

5. Decidability of Verification

In this section, we provide the two key decidability results of this work. They are orthogonal, in the following sense. On the one hand, the first result shows decidability of model checking for the whole $\mu\mathcal{L}_A^N$ logic, but considering the notion of run-boundedness, which is stronger than state-boundedness. On the other hand, the second result shows decidability of model checking for state-bounded ν-PNs, but focusing on a (relevant) fragment of $\mu\mathcal{L}_A^N$. We open the section by introducing the two results, and we dedicate the remainder of the section to prove them.

For run-bounded ν-PNs, the following holds.

Theorem 5.1. Verification of $\mu\mathcal{L}_A^N$ properties over run-bounded marked ν-PNs is decidable, and reducible to finite-state model checking.

This result means that, for a run-bounded marked ν-PN, it is possible to construct a faithful, finite-state abstraction of its infinite-state MTS, and model check $\mu\mathcal{L}_A^N$ properties over such a finite-state abstraction. In principle, this can be done using conventional model checking techniques, given the fact that:

- over a finite-state transition system, a $\mu\mathcal{L}_A^N$ property can re-expressed as a propositional μ-calculus property via quantifier elimination, by considering the finite domain of names used in the abstraction;
- verification of propositional μ-calculus properties over finite-state transition systems is decidable [Eme96].

The notion of run-boundedness is quite strong, because it does not allow for infinite runs that trigger unboundedly many times an arc that injects fresh names into the system. This is a severe restriction that is not compatible with RIAW-nets (cf. Section 2.4), in which the instance generator transition is meant to create unboundedly many process instances over time. For example, the RIAW-net of Figure 4 is not run-bounded. As we argued in Section 2.4, state-boundedness is instead a typically desired property that RIAW-nets must enjoy. For example, the RIAW-net of Figure 4 is state-bounded. We are consequently interested in obtaining a meaningful decidability result related to model checking state-bounded ν-PNs.

Towards this goal, we build on [BHCDG[+]13], where decidability of verification for state-bounded data-aware business processes is achieved by controlling the power of the $\mu\mathcal{L}_A$ logic (from which $\mu\mathcal{L}_A^N$ is derived) as far as FO quantifaction across states is concerned. In particular, decidability is shown for $\mu\mathcal{L}_P$, the fragment of $\mu\mathcal{L}_A$ in which FO quantification only applies to those objects that explicitly *persist* in the system. When quantification is applied to an object that disappears from the system, then it is possible to control whether the property, applied to that object, becomes *true* or *false*. In our setting, this means that the formulae can only quantify in a meaningful way over names that continue to be present in the system. By incorporating this intution into our setting, we obtain the logic $\mu\mathcal{L}_P^N$.

Definition 5.1. Given a marked ν-PN $\overline{N} = \langle P, T, F, m_0\rangle$, a $\mu\mathcal{L}_P^N$ formula Φ over $\overline{N}$ is defined as:

$$\Phi ::= true \mid Z \mid [\#p \leq c] \mid [\#p(x) \leq c] \mid x = y \mid \Phi_1 \wedge \Phi_2 \mid \neg\Psi \mid \exists x.\textsc{live}(x) \wedge \Psi \mid \textsc{live}(\vec{x}) \wedge \langle - \rangle \Psi \mid \textsc{live}(\vec{x}) \wedge [-]\Psi \mid \mu Z.\Psi$$

where $\textsc{live}(x_1, \ldots, x_k)$ is a shortcut for $\bigwedge_{i \in \{1,\ldots,k\}} \textsc{live}(x_i)$. For $\mu\mathcal{L}_P^N$ formulae, the usual assumptions done for $\mu\mathcal{L}_A^N$ hold, as well as the additional assumption that in $\textsc{live}(\vec{x}) \wedge \langle - \rangle \Psi$ and $\textsc{live}(\vec{x}) \wedge [-]\Psi$, the variables $\vec{x}$ are exactly the free variables of Φ, once we replace each bound predicate variable Z in Φ with its bounding formula $\mu Z.\Phi'$. Beside the usual abbreviations introduced for $\mu\mathcal{L}_A^N$, we make use of $\textsc{live}(\vec{x}) \rightarrow \langle - \rangle \Psi = \neg(\textsc{live}(\vec{x}) \wedge \langle - \rangle \neg\Psi)$ and $\textsc{live}(\vec{x}) \rightarrow [-]\Psi = \neg(\textsc{live}(\vec{x}) \wedge [-]\neg\Psi)$. ■

It is easy to see that $\mu\mathcal{L}_P^N$ is a syntactic fragment of $\mu\mathcal{L}_A^N$, i.e., every $\mu\mathcal{L}_P^N$ formula is also a $\mu\mathcal{L}_A^N$ formula.

As shown in the following example, $\mu\mathcal{L}_P^N$ properties are particularly useful in all those cases where names maintain an identity only if they persist in the system, but once they re-enter after a period of time in which they disappeared, they cannot be distinguished from globally fresh names. This is, e.g., the case of internal identifiers used as primary keys in a database table: when a tuple is deleted, the same primary key is typically re-employed in the future, but to identify a different tuple.

Example 5.1. The $\mu\mathcal{L}_P^N$ formula

$$\nu Z.(\exists n.[\#p_1(n) = 1] \wedge \mu Y.([\#p_2(n) = 1]) \vee (\textsc{live}(n) \wedge \langle - \rangle Y)) \wedge [-]Z$$

expresses that in every state of the system, place p_1 must contain a single name n that persists in the system until it is the unique name present in p_2. Instead, formula

$$\nu Z.(\exists n.[\#p_1(n) = 1] \wedge \mu Y.([\#p_2(n) = 1]) \vee (\textsc{live}(n) \rightarrow \langle - \rangle Y)) \wedge [-]Z$$

expresses that in every state of the system, place p_1 must contain a single name n that either disappears from the system or becomes eventually the unique name present in p_2. ■

For $\mu\mathcal{L}_P^N$ properties and state-bounded ν-PNs, the most interesting decidability result of this paper holds.

Theorem 5.2. Verification of $\mu\mathcal{L}_P^N$ properties over state-bounded marked ν-PNs is decidable, and reducible to finite-state model checking.

Notably, this result also suggests a methodology for verifying $\mu\mathcal{L}_P^N$ properties over ν-PNs. Given a marked ν-PN $\overline{N}$ of interest, we can proceed as follows:

1. We check whether $\overline{N}$ is state-bounded (this is decidable, as shown in Theorem 2.1).
2. If $\overline{N}$ is not state-bounded, we return a warning.
3. If instead $\overline{N}$ is state-bounded, we can verify whether a $\mu\mathcal{L}_P^N$ property of interest holds over $\Gamma_{\overline{N}}$ using conventional model checking techniques (thanks to Theorem 5.2).

Finally, observe that $\mu\mathcal{L}_P^N$ is a very rich logic. For example, the notion of soundness for RIAW-nets as

formalized in Section 3.2 is captured by a complex $\mu\mathcal{L}_A^N$ formula that actually falls inside the $\mu\mathcal{L}_P^N$ fragment. From this observation and Theorem 5.2, we obtain that:

Corollary 5.1. Checking whether a state-bounded RIAW-net is sound (in the sense of Section 3.2) is decidable, and reducible to finite-state model checking.

The remainder of this section is devoted to prove Theorems 5.1 and 5.2. Notably, the proofs rely on a translation mechanism from ν-PNs to data-centric dynamic systems (DCDSs), recently proposed in [BHCDG+13] to formalize data-aware business processes working over relational databases. We believe that this translation is interesting *per se*, and can provide the basis for a more systematic transfer of results between the area of formal methods for concurrent systems and that of foundations of data-aware processes.

5.1. Data-Centric Dynamic Systems

We recall the main aspects underlying DCDSs [BHCDG+13]. A DCDS is a pair $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ where $\mathcal{D}$ is a *data layer* and $\mathcal{P}$ is a *process layer*. The two layers are interacting as follows: the data layer stores all the data of interest inside a full-fledged relational database with constraints, while the process layer modifies and evolves such data, possibly injecting fresh data.

To formally defined DCDSs, we need some preliminary notions. Fix a a countably infinite set $\mathcal{C}$ of constants/values (we use standard names, hence blurring the distinction between constants and values). A relational schema $\mathcal{R}$ is a finite set $\{R_1/a_1, \ldots, R_n/a_n\}$ of relation names R_i, each with its own arity $a_i \in \mathbb{N}$. A database instance $\mathcal{I}$ over schema $\mathcal{R}$ is a set $\{R_1^{\mathcal{I}}, \ldots, R_n^{\mathcal{I}}\}$, where $R_i^{\mathcal{I}} \subset \mathcal{C}^{a_i}$ for each $i \in \{1, \ldots, n\}$. We denote with $\mathtt{ADOM}(\mathcal{I})$ the *active domain* of $\mathcal{I}$, i.e., the set of constants such that $c \in \mathtt{ADOM}(\mathcal{I})$ if and only if c occurs in some $R_i^{\mathcal{I}}$. A FO query over schema $\mathcal{R}$ is a (possibly open) formula of the form:

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

where, in $R(x_1, \ldots, x_n)$, we have that $R/n \in \mathcal{R}$, and each x_i is either a variable or a constant. We make use of standard abbreviations $Q_1 \vee Q_2 = \neg(\neg Q_1 \wedge \neg Q_2)$, and $\forall x.Q = \neg \exists x.\neg Q$.

Given a database instance $\mathcal{I}$ over $\mathcal{R}$, and a FO query Q over $\mathcal{R}$, the *answers* of Q over $\mathcal{I}$ correspond to the set $ans(Q, \mathcal{I})$ of substitutions σ of the free variables in Q with values in $\mathtt{ADOM}(\mathcal{I})$, such that $\mathcal{I} \models Q\sigma$, where $\models$ denotes FO entailment with active domain semantics [Lib04]. The active domain semantics is exploited to ensure that quantification is interpreted by considering only those values present in the current database, and not unrestrictedly those present in the overall quantification domain $\mathcal{C}$.

Notation $Q\sigma$ denotes the query obtained from Q by applying substitution σ over the free variables of Q. In this light, $Q\sigma$ can be seen as a boolean query, and with some abuse of notation, we write $ans(Q\sigma, \mathcal{I}) \equiv \mathsf{true}$ if and only if $\mathcal{I} \models Q\sigma$, and $ans(Q\sigma, \mathcal{I}) \equiv \mathsf{false}$ otherwise. Using these notions at hand, we can now provide a formal definition of the layers of a DCDS.

Definition 5.2. A *data layer* is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ where:

- $\mathcal{C}$ is a countably infinite set of values;
- $\mathcal{R}$ is a database schema;
- $\mathcal{E}$ is a finite set $\{\mathcal{E}_1, ..., \mathcal{E}_m\}$ of constraints, i.e., domain-independent closed FO formulae whose terms are either constants from $\mathtt{ADOM}(\mathcal{I}_0)$, or quantified variables.
- $\mathcal{I}_0$ is a database instance that represents the initial state of the data layer, and that conforms to the schema $\mathcal{R}$ and satisfies the constraints $\mathcal{E}$, i.e., for each constraint $\mathcal{E}_i \in \mathcal{E}$, we have $ans(\mathcal{E}_i, \mathcal{I}_0) \equiv \mathsf{true}$. ■

The process layer constitutes the progression mechanism for the DCDS. It is assumed that at every time the current instance of the data layer can be both arbitrarily queried and updated (through action executions), possibly calling external services calls to import new values into the system.

Definition 5.3. A *process layer* $\mathcal{P}$ over a data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \rho \rangle$. $\mathcal{F}$ is a finite set of *functions*, each representing the interface to an external service. Such services can be called, and as a result the function is activated and the answer is produced. How the result is actually computed is unknown to the DCDS since the services are indeed external.

$\mathcal{A}$ is a finite set of *actions*, whose execution updates the data layer, and may involve external service calls. Formally, an *action* $\alpha \in \mathcal{A}$ is an expression $\alpha(p_1, ..., p_n) : \{e_1, ..., e_m\}$ where:

- $\alpha(p_1, ..., p_n)$ is the action *signature*, constituted by a name α and a sequence $p_1, ..., p_n$ of *parameters*, to be substituted with values when the action is invoked;
- $\{e_1, ..., e_m\}$, also denoted as $\texttt{EFFECT}(\alpha)$, is a set of *specifications of effects*, which are assumed to take place simultaneously. Each e_i has the form $Q \rightsquigarrow E_i$, where:
  - Q is a (possibly open) query over $\mathcal{R}$ whose terms are variables, action parameters, and constants from $\texttt{ADOM}(\mathcal{I}_0)$;
  - E_i is the *effect*, i.e., a set of facts for $\mathcal{R}$, which includes as terms the following: terms in $\texttt{ADOM}(\mathcal{I}_0)$, free variables of q_i^+ and Q_i^- (including action parameters), and Skolem terms formed by applying a function $f \in \mathcal{F}$ to one of the previous kinds of terms. Such Skolem terms involving functions represent external (nondeterministic) service calls and are interpreted as the returned value chosen by an external user/environment when executing the action.

Finally, ρ is a finite set of *condition-action rules* of the form $Q(\vec{x}) \mapsto \alpha(\vec{x})$, where $\alpha \in \mathcal{A}$ and Q is a FO query over $\mathcal{R}$, which can contain as terms variables and constants in $\texttt{ADOM}(\mathcal{I}_0)$, and whose free variables exactly match the parameters of α. Together, all such rules form the specification of a *process*, which tells at any moment which actions can be executed. ■

Execution semantics. The execution semantics of a DCDS $\mathcal{S}$ is defined in terms of a possibly infinite *relational transition system* (RTS) $\Lambda_\mathcal{S}$, that is, a transition system whose states are labeled by database instance. The RTS represents all possible computations that the process layer can do on the data layer. Formally, $\Lambda_\mathcal{S} = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where: *(i)* Δ is a countably infinite set of values; *(ii)* Σ is a set of states; *(iii)* $s_0 \in \Sigma$ is the initial state; *(iv)* db is a function such that for each state $s \in \Sigma$ returns a database $D \subseteq \Delta$ conforming to $\mathcal{R}$; *(v)* $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation over states.

Given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \rho \rangle$, one can construct $\Lambda_\mathcal{S}$ as follows. Starting from $\mathcal{I}_0$, the condition-action rules in ρ are evaluated, so as to determine the set of all possible executable actions with corresponding ground parameter assignments. Then, nondeterministically, one such action with parameter assignment $\alpha\sigma$ is selected, and applied over $\mathcal{I}_0$. This is done by calculating all the answers of in the left-hand side of every effect in $\alpha\sigma$, and grounding the right-hand side accordingly. In case that the right-hand contains service calls, they are issued, obtaining back results nondeterministically picked in $\mathcal{C}$. Such results are then used to complete the grounding of the right-hand sides of the effects. The overall set of so-obtained ground facts constitutes the next database instance. The construction then proceeds recursively over the newly generated states. For a formal description of the DCDS execution semantics, refer to [BHCDG+13].

Example 5.2. We consider a DCDS showing how to manipulate (a simplified version of) user carts in an e-commerce website. Specifically, the DCDS schema contains the following relations:

- *Prod* is a read-only unary relation containing the catalogue of the website; $\mathit{Prod}(\mathsf{p})$ indicates that product p is sold on the website (for simplicity, we do not deal with the actual availability of products).
- *Cart* is a unary relation representing the active carts present on the website.
- *In* is a binary relation showing which products are present in which carts.

The data layer is also equipped with two *foreign key* constraints, respectively indicating that the first argument of the *In* relation points to a product, while the second refers to a cart. Such foreign keys are captured by the following FO constraint: $\forall p, c. \mathit{In}(p, c) \rightarrow \mathit{Prod}(p) \wedge \mathit{Cart}(c)$. The initial state $\mathcal{I}_0$ contains the catalogue of sold products, and no cart: $\mathcal{I}_0 = \{\mathit{Prod}(\mathsf{pc}), \mathit{Prod}(\mathsf{tv}), \mathit{Prod}(\mathsf{phone})\}$.

The process layer supports three main functionalities:

- creation of a user cart;
- manipulation of a user cart, through the addition and deletion of products;
- emptying a user cart (useful when the corresponding customer order is finalized and paid).

The creation of a user cart is managed by a dedicated action with no parameters:

$$\textsc{createCart}() : \left\{ \begin{array}{rcl} \mathsf{true} & \rightsquigarrow & \mathit{Cart}(\mathit{getCId}()), \\ \mathit{Cart}(x) & \rightsquigarrow & \mathit{Cart}(x), \\ \mathit{Prod}(y) & \rightsquigarrow & \mathit{Prod}(y), \\ \mathit{In}(x, y) & \rightsquigarrow & \mathit{In}(x, y) \end{array} \right\}$$

The first effect creates the cart, exploiting the 0-ary service call *getCId* to inject the cart identifier into the system. The remaining effects copy the extensions of all relations, so as to keep unaltered the information already present in the system. Condition-action rule $\mathsf{true} \mapsto \textsc{createCart}()$ indicates that it is always possible to create a cart.

The manipulation of a user cart is managed though two dedicated actions and a corresponding condition-action rule, respectively focused on the addition and deletion of a product to/from a cart. Addition is managed by an action equipped with a parameter that denotes the cart to be manipulated:

$$\textsc{addProd}(\mathsf{c}) : \left\{ \begin{array}{l} \mathit{Cart}(\mathsf{c}) \rightsquigarrow \mathit{In}(\mathit{getProd}(\mathsf{c}), \mathsf{c}), \\ \mathit{Cart}(x) \rightsquigarrow \mathit{Cart}(x), \\ \mathit{Prod}(y) \rightsquigarrow \mathit{Prod}(y), \\ \mathit{In}(x, y) \rightsquigarrow \mathit{In}(x, y) \end{array} \right\}$$

The first effect checks whether parameter c is indeed pointing to a cart and, if so, introduces a new product in the cart. To do so, it exploits a dedicated service call *getProd* that simulates a user interaction: given the cart identifier as input, it returns the product to be added. In accordance with the DCDS execution semantics, the presence of the foreign key constraint that types the first component of the *In* relation, it is guaranteed that the value returned by *getProd* is indeed a product. Condition-action rule $\mathit{Cart}(x) \mapsto \textsc{addProd}(x)$ indicates that the $\textsc{addProd}$ action can be applied with parameter x provided that x points to a cart.

Deletion is instead captured by an action that takes two parameters, representing the product of interest, and the cart from which that product must be deleted:

$$\textsc{delProd}(\mathsf{p}, \mathsf{c}) : \left\{ \begin{array}{r} \mathit{Cart}(x) \rightsquigarrow \mathit{Cart}(x), \\ \mathit{Prod}(y) \rightsquigarrow \mathit{Prod}(y), \\ \mathit{In}(x, y) \land \neg(x = \mathsf{p} \land y = \mathsf{c}) \rightsquigarrow \mathit{In}(x, y) \end{array} \right\}$$

The third effect implicitly realizes deletion by expressing that all tuples of the *In* relation have to be maintained, provided that they do not match with the tuple $\langle \mathsf{p}, \mathsf{c} \rangle$. Condition-action rule $\mathit{In}(x, y) \mapsto \textsc{delProd}(x, y)$ indicates that $\textsc{delProd}$ can be applied with parameters x and y provided that they respectively denote a product and a cart that contains that product.

Finally, a cart can be flushed through a dedicated action that takes a cart identifier and removes it from the database, together with all the products contained in it:

$$\textsc{flushCart}(\mathsf{c}) : \left\{ \begin{array}{r} \mathit{Cart}(x) \land x \neq \mathsf{c} \rightsquigarrow \mathit{Cart}(x), \\ \mathit{Prod}(y) \rightsquigarrow \mathit{Prod}(y), \\ \mathit{In}(x, y) \land y \neq \mathsf{c} \rightsquigarrow \mathit{In}(x, y) \end{array} \right\}$$

The first effect deletes the cart, by copying all carts that do not match with c. Similarly, the third effect deletes the content of the cart, by copying the content of all carts but the one matching with c. Condition-action rule $\mathit{Cart}(x) \mapsto \textsc{flushCart}(x)$ ensures that the $\textsc{flushCart}$ action can be applied only with a parameter that points to an active cart.

■

5.2. From ν-PNs to DCDSs: Issues and Intuitive Translation

In this section, we provide the ground for a behavior-preserving, systematic translation of ν-PNs into DCDSs. In particular, we discuss the intuition behind the translation, and the main issues that must be addressed.

As a running example, we consider the ν-PN $\overline{N}$ depicted in Fig. 1, and show how it can be faithfully encoded in terms of a corresponding DCDS $\langle \mathcal{D}_N, \mathcal{P}_N \rangle$, able to reconstruct the execution semantics of $\overline{N}$. The general translation mechanism is then presented in Section 5.3.

The two major mismatches between ν-PNs and DCDSs are:

- *Set vs multiset semantics.* On the one hand, the states of a DCDS correspond to standard relational databases, which define the extension of relations as *sets* of tuples; on the other hand, the markings of a ν-PN map places to *multisets* of names.
- *Fresh vs arbitrary input data.* On the one hand, ν-PNs use special variables to inject *fresh* names, i.e., names not already present in the current marking, also guaranteeing that distinct special variables are bound to different names. On the other hand, DCDSs use service calls to inject *arbitrary* input data that are not necessarily fresh, but could in fact correspond to data already present in the current database. Furthermore, distinct service calls could return the same value.

The behavior of DCDSs in the light of these two observations is apparent by considering again Example 5.2. In fact, the addition of a product to a cart already containing that product results in a no-op (cf. the looping transitions at the extreme right of Figure 6). Moreover, when generating a cart, the service call used to

Fig. 6. Fragment of the infinite-state RTS capturing the execution semantics of the cart management DCDS discussed in Example 5.2

obtain the cart identifier may return an already existing identifier, resulting again in a no-op (cf. the looping transitions at the bottom and top of Figure 6).

In the remainder of this section, we argue that these two characteristics of DCDSs are not a limitation, that is, we show how multiset semantics and fresh inputs can be seamlessly simulated by DCDSs.

To represent a ν-PN marking into a DCDS, a naive solution would be to map places to unary relations, and the distribution of tokens in the ν-PN to tuples of such relations. However, due to set semantics, this would make it impossible for the DCDS to distinguish two tokens positioned in the same place and carrying the same name. For example, with this strategy $m(p_1) = \{\mathsf{a}, \mathsf{a}, \mathsf{b}\}$ of p_1 in Fig. 1(a) would be wrongly represented by $\{p_1(\mathsf{a}), p_1(\mathsf{b})\}$. To solve this problem, we relationally represent places and their tokens using binary relations that combine token names with internal token identifiers, whose only purpose is to faithfully reconstruct the multiplicity of tokens belonging to the same place and carrying the same name. For example, $m(p_1) = \{\mathsf{a}, \mathsf{a}, \mathsf{b}\}$ could be represented relationally as $\{p_1(345, \mathsf{a}), p_1(712, \mathsf{a}), p_2(486, \mathsf{b})\}$. This usage of internal identifiers must be consistently exploited when, upon firing, two distinct tokens with the same name enter into the same place.

To represent the firing of a ν-PN transition into a DCDS, a naive solution would be to use DCDS actions that employ service calls to simulate the generation of fresh names. However, this would lead to a "lossy" simulation of the original ν-PN, due to the fact that service calls provide arbitrary results, not guaranteed to be fresh. For example, by simulating transition t of Figure 1 using two service calls to account for ν_1 and ν_2, it could happen that such service calls return both value a, with the consequence that, after firing, p_3 and p_5 would contain tokens all carrying name a. This spurious evolution cannot occur according to the execution semantics of ν-PNs. Exactly the same kind of problem applies to the generation and manipulation of internal identifiers, introduced before to suitably reconstruct the multi-set semantics of ν-PNs into DCDSs.

This problem would be solved if we could "prune away" the nondeterministic evolutions were any of these two service calls returns a non-fresh value, as well as those were the two service calls return the same value. To enforce this pruning, we encode the firing of a $\nu - PN$ transition into a non-interruptible sequence of two DCDS action applications as follows:

- The first action manages the consumption and generation of tokens induced by a transition of the ν-PN. In particular, the generation of required tokens amounts to inject their names and internal identifiers, and to insert them into a special relation acting as dedicated, temporary container. The applicability of

this action is determined by a condition-action rule that encodes the enablement condition of the ν-PN transition, and that passes to the generation action the identifiers and names of the tokens that must be consumed.

- The second action takes care of transferring the generated tokens from the temporary container into the corresponding target places, in accordance with the output arcs of the original ν-PN transition.
- Specific database constraints over the temporary relations are used to check whether the containers indeed store fresh values that are also distinct from each other. This ensures that spurious evolutions are blocked, and that the two steps are indeed applied only when service calls faithfully maintain the multiplicity of tokens via their identifiers, at the same time reproducing the freshness requirement of ν-PNs regarding the generated names.

We now substantiate these two strategies to the ν-PN $\overline{N} = \langle P, T, F, m_0 \rangle$ of Figure 1. We use the following typographical conventions: v for variables, v for constants, and _ for anonymous variables, i.e., existentially quantified variables not appearing elsewhere.

Data Layer. The data layer is $\mathcal{D}_N = \langle \mathcal{C}_N, \mathcal{R}_N, \mathcal{E}_N, \mathcal{I}_0^N \rangle$. The data domain $\mathcal{C}_N$ contains the following components:

- The countably infinite set Id of names.
- Distinguished constants used to denote the different elements of N: the set T of transitions, the set P of places, and the set $\{\mathsf{x}, \mathsf{y}, \nu_1, \nu_2\}$ of constants corresponding to the variables inscribed on the arcs of $\overline{N}$ (these are needed to identify the corresponding temporary containers).
- Countably infinite values used as internal identifiers for the tokens; since internal identifiers and names are always treated separately by the DCDS, identifiers can be directly picked from the set Id of names.

The relation schema $\mathcal{R}_N$ is constituted by the following relations:

- $\{p_i/2 \mid i \in \{1, \ldots, 5\}\}$, mirroring the five places in P; fact $p_i(\mathsf{id}, \mathsf{n})$ indicates that place p_i currently contains a token that carries name n and has internal identifier id.
- $\mathit{NewID}/5$, the temporary relation used for the generation of fresh names and identifiers; fact $\mathit{NewID}(\mathsf{t}, \mathsf{p}, \mathsf{x}, \mathsf{n}, \mathsf{id})$ indicates that: *(i)* the transition identified by t has produced a token with identifier id and name n, *(ii)* this token matches variable x attached to t, and *(iii)* the destination of this token in the next step is place p.
- $\mathit{FL}/0$, a nullary relation used to distinguish the two execution modes of the DCDS, i.e., generation of tokens vs transfer of tokens from the temporary container to the proper place relations.

Let us discuss in more details the two DCDS execution modes distinguished by the presence or absence of the flag FL into the DCDS sate. The *generation phase* models the initial effect induced by the firing of a transition, namely the consumption of tokens from the input places of t according to the involved variables and the corresponding matching. At the same time, it generates the necessary new token identifiers and names, by invoking dedicated service calls and storing the results inside the temporary relation *NewID*. The *transfer phase* consists in the forwarding of tokens from such temporary relations to the output places of t, provided that the previous step has completed correctly, i.e., without violating any constraint in $\mathcal{E}_N$.

Specifically, $\mathcal{E}_N = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4\}$ Constraints ε_1 and ε_2 ensure that identifiers present in the temporary relation *NewID* are unique, making sure that tokens are distinguished and their multiplicitly properly reflected by the DCDS. In particular, ε_1 checks that such identifiers are distinct from those present in the place relations, while ε_2 models the fact that the identifier column is a primary key for *NewID*, that is, there are no two tokens in *NewID* sharing the same identifier:

$$\varepsilon_1 = \forall id.\mathit{NewID}(_, _, _, id, _) \wedge \bigvee_{i \in \{1,\ldots,5\}} p_i(id, _) \rightarrow \bot$$
$$\varepsilon_2 = \forall id, t, p, v, n, t', p', v', n'.\mathit{NewID}(t, p, v, id, n) \wedge \mathit{NewID}(t', p', v', id, n') \rightarrow t = t' \wedge p = p' \wedge v = v' \wedge n = n'$$

Constraints ε_3 and ε_4 mirror ε_1 and ε_2, focusing on names rather than identifiers. In particular, they target names referring to the new name variables in Υ, i.e., in our case ν_1 and ν_2 (which are constants in the DCDS). Since such variables must be bound to tokens carrying fresh names, ε_3 guarantees that such names are not already used in the place relations, whereas ε_4 ensures that such names are distinct from each other (i.e., that different new name variables are bound to different names):

$$\varepsilon_3 = \forall n. \bigvee_{\mathsf{c} \in \{\nu_1, \nu_2\}} \mathit{NewID}(_, _, \mathsf{c}, _, n) \wedge \bigvee_{i \in \{1,\ldots,5\}} p_i(_, n) \rightarrow \bot$$
$$\varepsilon_4 = \forall n_1, n_2.\mathit{NewID}(_, _, \nu_1, _, n_1) \wedge \mathit{NewID}(_, _, \nu_2, _, n_2) \rightarrow n_1 \neq n_2$$

Finally, $\mathcal{I}_0^N$ encodes m_0 by maintaining the name distribution of tokens, as well as their multiplicity. The latter requirement is enforced by picking pairwise distinct identifiers from $\mathcal{C}_N$. Recall that the DCDS handles

identifiers and names separately, so we can pick identifiers from the set Id of names. Hence, a suitable choice for $\mathcal{I}_0^N$ is $\{p_1(\mathsf{a},\mathsf{a}), p_1(\mathsf{b},\mathsf{a}), p_1(\mathsf{c},\mathsf{e}), p_2(\mathsf{d},\mathsf{e}), p_2(\mathsf{e},\mathsf{c})\}$.

Process layer. The process layer resulting from the translation of $\overline{N}$ is defined as $\mathcal{P}_N = \langle \mathcal{F}_N, \mathcal{A}_N, \rho_N\rangle$. $\mathcal{F}_N$ contains service calls that are used to inject token identifiers and names into the system. In particular, each time a transition t fires, tokens are generated according to the arcs that have t as input. The generation of a new identifier depends on: *(i)* the considered arc, which is in turn determined by the input transition and output place; *(ii)* the variable name present on the arc; *(iii)* the occurrence of the variable name.[2] For this reason, we introduce a service call $genID/4$, where $genID(\mathsf{t},\mathsf{p},\mathsf{x},\mathsf{i})$ represents the new id generation for the i-th occurrence of variable x on the arc that connects t to p. Name generation depends instead only on the considered new name variable in Υ. Therefore, we introduce a service call $genName/1$, where $genName(\nu)$ represents the generation of a name for the new name variable ν.

The action component $\mathcal{A}_N$ contains:

- a dedicated action $\textsc{gen}_{t_i}$ for each transition t_i in T, used to handle the token generation phase upon firing t_i, and the insertion of token names/identifiers into the temporary relation $NewID$;
- an additional, generic action TRANSF to handle the token transfer phase, removing tuples from $NewID$ and inserting names/identifiers into the corresponding place relations.

The process ρ_N contains one condition-action rule per action. The executability condition for TRANSF just checks for the presence of flag FL. The executability condition for each $\textsc{gen}_{t_i}$ is instead a query that checks for the absence of flag FL, and that mirrors the enablement of transition t_i in the original net. This requires to check for the presence of a sufficient amount of tokens in the input places of t_i, such that their names properly match the variable attached to the arcs connecting such input places to t_i. Notice that such matched tokens need to be transferred by t_i together with the newly generated ones. For this reason, such tokens are not only matched, but are actually used as parameters for $\textsc{gen}_{t_i}$.

Specifically, in Figure 1 we have a single transition t, hence $\mathcal{A}_N = \{\textsc{gen}_t(id_1, id_2, id_3, id_4, x, y), \textsc{transf}()\}$. Action $\textsc{gen}_t$ has in fact 6 parameters, because transition t of Figure 1 is associated to the consumption of 4 tokens (represented by their identifiers), and involves two distinct names. Since the input arcs of t mention four distinct tokens with two involved names (matching variables x and y), $\textsc{gen}_t$ has 6 parameters. The process ρ_N expresses the executability of $\textsc{gen}_t$ as follows:

$$p_1(id_1,x) \wedge p_1(id_2,x) \wedge p_1(id_3,y) \wedge p_2(id_4,y) \wedge \bigwedge_{i,j\in\{1,4\}, i\neq j} id_i \neq id_j \wedge \neg FL \mapsto \textsc{gen}_t(id_1,id_2,id_3,id_4,x,y)$$

Observe the consistent usage of variables x and y w.r.t. the ν-PN in Figure 1, and the fact that the four token identifiers and the two matched names are used as parameters for $\textsc{gen}_t$. This allows us to formalize the effects of $\textsc{gen}_t$ in such a way that the selected tokens are removed, and at the same time the matched names are transferred to the proper output places. Specifically, $\textsc{gen}_t$ is an action containing multiple effects, and that is built as follows (observe the difference between constant x and parameter x):

$$\textsc{gen}_t(id_1,id_2,id_3,id_4,x,y):$$
$$\left\{\begin{array}{rl}
p_1(id,n) \wedge id \neq id_1 \wedge id \neq id_1 \wedge id \neq id_3 & \rightsquigarrow \{p_1(id,n)\}, \\
p_2(id,n) \wedge id \neq id_4 & \rightsquigarrow \{p_2(id,n)\}, \\
p_i(id,n) & \rightsquigarrow \{p_i(id,n)\}, \qquad \text{for } i \in \{3,4,5\} \\
true & \rightsquigarrow \left\{\begin{array}{l} NewID(\mathsf{t},\mathsf{p}_3,\mathsf{x},genID(\mathsf{t},\mathsf{p}_3,\mathsf{x},1),x), \\ NewID(\mathsf{t},\mathsf{p}_3,\mathsf{x},genID(\mathsf{t},\mathsf{p}_3,\mathsf{x},2),x) \\ NewID(\mathsf{t},\mathsf{p}_3,\nu_1,genID(\mathsf{t},\mathsf{p}_3,\nu_1,1),genName(\nu_1))\} \end{array}\right\}, \\
true & \rightsquigarrow \{NewID(\mathsf{t},\mathsf{p}_4,\epsilon,genID(\mathsf{t},\mathsf{p}_3,\epsilon,1),\bullet)\} \\
true & \rightsquigarrow \left\{\begin{array}{l} NewID(\mathsf{t},\mathsf{p}_5,\nu_1,genID(\mathsf{t},\mathsf{p}_5,\nu_1,1),genName(\nu_1)), \\ NewID(\mathsf{t},\mathsf{p}_5,\nu_1,genID(\mathsf{t},\mathsf{p}_5,\nu_2,1),genName(\nu_2)) \end{array}\right\}, \\
true & \rightsquigarrow \{FL\}
\end{array}\right\}$$

The first block of effects is dedicated to maintain those tokens that are not consumed by transition t. For places p_1 and p_2, which constitute the preset of t, this requires to properly filter out those tokens that have

[2] Recall that, in general, the same variable could label the same arc multiple n times. This will lead to the generation of n tokens sharing the same name, that is, n distinct token identifiers in the DCDS.

been selected in the precondition of $\textsc{gen}_t$ (which, in fact, has non-empty answers only if t is enabled). The second block of effects is instead focused on the generation of one distinct token for each of the variables that decorate the output arcs of t, with the corresponding restrictions over matching names. For such tokens, fresh identifiers are always generated (so as to guarantee that their multiplicity is preserved). Names that match with input variables are obtained from the parameters of $\textsc{gen}_t$, whereas fresh names are generated with the dedicated *genName* service call. All such tokens are inserted into the temporary *NewID* relation. The last effect switches the flag from false to true, inhibiting the possibility of reapplying actions of the type $\textsc{gen}_t$, and making sure that the transfer generation is the only possible next phase to be executed; this is imposed in the precondition of $\textsc{gen}_t$, which contains $\neg FL$ among its conjuncts.

Observe that, once $\textsc{gen}_t$ is applied, the newly obtained database must conform to all constraints in $\mathcal{E}$. Consequently, only states that correctly assign new identifiers and fresh names matching Υ variables are kept as valid successors.

Let us now turn to the action TRANSF. It is independent of the specific transition, because it only needs to transfer the tokens generated in the previous phase from the temporary *NewID* relation to the proper output places, which are memorized inside tuples of *NewID* itself. Thus, the precondition of TRANSF needs only to check whether *FL* is true (this attests that a generation action has been applied in the previous computation step). In particular, ρ_N contains rule $FL \mapsto \textsc{transf}()$, and, when translating the $\nu - PN$ of Figure 1, TRANSF has the following shape:

$$\textsc{transf}() : \begin{Bmatrix} p_i(id,n) \rightsquigarrow \{p_i(id,n)\}, & \text{for } i \in \{1,\ldots,5\} \\ NewId(_,\mathsf{p_j},_,id,n) \rightsquigarrow \{p_j(id,n)\} & \text{for } j \in \{1,\ldots,5\} \end{Bmatrix}$$

Notice that since *FL* is not explicitly copied by TRANSF, it is implicitly toggled, making TRANSF not applicable anymore (it will be again applicable after a generation step).

5.3. From ν-PNs to DCDSs: Translation Algorithm

We now provide a systematic translation τ from arbitrary ν-PNs to DCDSs, which generalizes the intuitions discussed in Section 5.2.

Given a marked ν-PN $\overline{N} = (P,T,F,m_0)$, τ produces a DCDS $\tau(\overline{N}) = \{\mathcal{D}_N, \mathcal{P}_N\}$, whose execution semantics faithfully reproduces that of $\Gamma_{\overline{N}}$. Specifically, the data layer $\mathcal{D}_N = \{\mathcal{C}_N, \mathcal{R}_N, \mathcal{E}_N, \mathcal{I}_0^N\}$ is obtained by applying Algorithm 1. Here $\mathcal{E}_N$ is constituted by the following constraints:

- ε_1 and ε_2 ensure the uniqueness of new identifiers from *NewID*;
- ε_3 and ε_4 ensure the uniqueness of the fresh names stored in *NewID* (note that Υ_N contains all the fresh variables present in $\overline{N}$).

The initial state $\mathcal{I}_0$ is set depending to the initial marking m_0 of $\overline{N}$, taking into account the need of distinguishing tokens through the introduction of token identifiers in $\tau(\overline{N})$.

The process layer of $\tau(N)$ is $\mathcal{P}_N = \{\mathcal{F}_N, \mathcal{P}_N, \rho_N\}$ is constructed using Algorithm 2. The set of service calls $\mathcal{F}_N$ is constituted by a service call $genID(\mathsf{t},\mathsf{p},\mathsf{v},\mathsf{i})$ returning (new) token identifiers, and a service call $genFresh(\nu)$ returning (fresh) token names. For every transition $t \in T$ the algorithm defines a token generation action $\textsc{gen}_t$, and corresponding condition-action rule that ensures that $\textsc{gen}_t$ is executed with given parameters if and only if transition t can be fired by consuming the tokens corresponding to those parameters. The additional action TRANSF completes each $\textsc{gen}_t$ action by transporting the consumed tokens, together with the newly generated ones, to the right target places. Also TRANSF comes with a dedicated condition-action rule, just enforcing that between the application of two generation actions, TRANSF is the only applicable action.

We now elaborate on the faithfullness of such translation. Intuitively, the key property that our translation procedure should preserve is that verification of properties over the marked ν-PN of interest carries over the corresponding DCDS. To properly establish this correspondence, though, we need a way to understand $\mu\mathcal{L}_A^N$ properties over the input ν-PN as corresponding properties over the obtained DCDS. To do so, we resort to the $\mu\mathcal{L}_A$ logic to express properties over DCDSs, and introduce a translation function ξ that, given a $\mu\mathcal{L}_A^N$ property Φ, translates it into a corresponding $\mu\mathcal{L}_A$ property $\xi(\Phi)$. There are two crucial aspects that ξ must tackle:

Algorithm 1 Data Layer generator

1: $\mathcal{C}_N := Id \cup T \cup P \cup Var$
2: $\mathcal{R}_N := \{p_i/2 \mid p_i \in P\} \cup NewID/5 \cup FL/0$ ▷ $NewID(\mathsf{t},\mathsf{p},\mathsf{v},\mathsf{d},\mathsf{id})$
3: $\varepsilon_1 := \top$
4: **for all** $id \in \mathcal{C}_N$ **do**
5: $\quad \varepsilon_1 := \varepsilon_1 \wedge \Big(NewID(_,_,_,id,_) \wedge \bigvee_{p_i \in \mathcal{R}_N} p_i(id,_) \rightarrow \bot\Big)$
6: $\varepsilon_2 := \top$
7: **for all** $id, t, p, v, n, t', p', v', n' \in \mathcal{C}_N$ **do**
8: $\quad \varepsilon_2 := \varepsilon_2 \wedge \Big(NewID(t,p,v,id,n) \wedge NewID(t',p',v',id,n') \rightarrow t = t' \wedge p = p' \wedge v = v' \wedge n = n'\Big)$
9: $\varepsilon_3 := \top$
10: **for all** $n \in \mathcal{C}_N$ **do**
11: $\quad \varepsilon_3 := \varepsilon_3 \wedge \Big(\bigvee_{\mathsf{c} \in \Upsilon_N} NewID(_,_,\mathsf{c},_,n) \wedge \bigvee_{p_i \in \mathcal{R}_N} p_i(_,n) \rightarrow \bot\Big)$
12: $\varepsilon_4 := \top$
13: **for all** n_1, n_2 **do**
14: $\quad \varepsilon_4 := \varepsilon_4 \wedge \Big(\bigvee_{\nu_1,\nu_2 \in \Upsilon_N, \nu_1 \neq \nu_2} \big(NewID(_,_,\nu_1,_,n_1) \wedge NewID(_,_,\nu_2,_,n_2) \rightarrow n_1 \neq n_2\big)\Big)$
15: $\mathcal{E}_N := \{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \varepsilon_4\}$
16: **for all** $p \in P$ **do**
17: $\quad$ **if** $M_0(p) \neq \emptyset$ **then**
18: $\quad\quad$ **for all** $\mathsf{d} \in M_0(p)$ **do**
19: $\quad\quad\quad$ **for all** $\mathsf{id}^p_k \in X_p \subset \mathbb{N}$ **do**
20: $\quad\quad\quad\quad$ **if** $\forall i, j, .i \neq j$ we have $\mathsf{id}^p_i \neq \mathsf{id}^p_j$, and $\forall p, p'.X_p \cap X_{p'} = \emptyset$ **then**
21: $\quad\quad\quad\quad\quad \mathcal{I}_0 \leftarrow \{p(\mathsf{id}^p_k, \mathsf{d})\}$

- the local queries mentioned in Φ have to be suitably reformulated in $\xi(\Phi)$ as standard FO queries over the resulting DCDS;
- next-state operators in Φ must be doubled in $\xi(\Phi)$, since each execution step in the input marked ν-PN (corresponding to the firing of a transition) is mirrored as a sequence of two execution steps in the corresponding DCDS (one for the consumption of tokens and the generation of new tokens, and the other for transferring the tokens into the output places).

Technically, given a $\mu\mathcal{L}_A^N$ property Φ, the $\mu\mathcal{L}_A$ property $\xi(\Phi)$ is obtained by applying the following translation mechanism:

$$\xi(\Phi) = \begin{cases} \Phi & \text{if } \Phi \in \{true, x = y, Z\} \\ \forall i_1, ..., i_c, i_{c+1}. \bigwedge_{j \in \{1,..,c+1\}} p(i_j, _) \rightarrow \bigvee_{k,l \in \{1,..,c+1\}, k<l} i_k = i_l & \text{if } \Phi = [\#p \leq c] \\ \forall i_1, ..., i_c, i_{c+1}. \bigwedge_{j \in \{1,..,c+1\}} p(i_j, x) \rightarrow \bigvee_{k,l \in \{1,..,c+1\}, k<l} i_k = i_l & \text{if } \Phi = [\#p(x) \leq c] \\ \neg\xi(Psi) & \text{if } \Phi = \neg\Psi \\ \xi(\Phi_1) \vee \xi(\Phi_2) & \text{if } \Phi = \Phi_1 \vee \Phi_2 \\ \exists x.\xi(\Psi) & \text{if } \Phi = \exists x.\Psi \\ \langle - \rangle\langle - \rangle\xi(\Psi) & \text{if } \Phi = \langle - \rangle\Psi \\ \mu Z.\xi(\Psi) & \text{if } \Phi = \mu Z.\Psi \end{cases}$$

Observe in particular how local queries in $\mu\mathcal{L}_A^N$ are translated into FO counting queries over the current database, and the fact that each occurrence of $\langle - \rangle$ in Φ becomes a double occurrence $\langle - \rangle\langle - \rangle$ in $\xi(\Phi)$.

We are now in the position of formally assessing the connection between ν-PNs and DCDSs, by exploiting the translation functions τ and ξ.

Lemma 5.1. For every marked ν-PN $\overline{N}$ and closed $\mu\mathcal{L}_A^N$ formula Φ, $\overline{N} \models \Phi$ if and only if $\tau(\overline{N}) \models \xi(\Phi)$.

Proof. Let $\overline{N} = \langle P, T, F, m_0 \rangle$, and $\tau(\overline{N}) = \langle \mathcal{D}_N, \mathcal{P}_N \rangle$. Recall that for each transition $t \in T$ there are an action $\textsc{gen}_t$ and a condition-action rule $Q(\vec{x}) \mapsto \textsc{gen}_t$ in $\mathcal{P}_N$. In addition, also $\textsc{transf}$ is an action in $\mathcal{P}_N$.

Algorithm 2 Process Layer generator

1: $\mathcal{F}_N := \{genID(\mathsf{t},\mathsf{p},\mathsf{v},\mathsf{i}), genFresh(\nu)\}$
2: $\rho_N, \mathcal{A}_N := \emptyset$
3: **for all** $t \in T$ **do**
4: $\quad \phi := \neg FL;$
5: $\quad \vec{v}, \vec{id} := \emptyset$
6: $\quad$ **for all** $p_j \in {}^\bullet t$ and $v \in pre(t, p_j)$ **do**
7: $\quad\quad$ **for all** $m, k \in \{1, ..., |pre(t, p_j)|\}$ **do**
8: $\quad\quad\quad$ **if** $m \neq k$ **then**
9: $\quad\quad\quad\quad \vec{v} := \vec{v} \cup \{v\}$
10: $\quad\quad\quad\quad \vec{id} := \vec{id} \cup \{id^j_k\}$
11: $\quad\quad\quad\quad \phi := \phi \wedge p_j(id^j_k, v) \wedge id^j_k \neq id^j_m$
12: $\quad \rho_N := \rho_N \cup \{\phi \mapsto \textsc{gen}_t(\vec{v}, \vec{id})\}$
13: $\quad \textsc{gen}_t(\vec{id}, \vec{v}) := \emptyset;$
14: $\quad e_1 := true \rightsquigarrow FL$
15: $\quad \textsc{gen}_t(\vec{id}, \vec{v}) := \textsc{gen}_t(\vec{v}, \vec{id}) \cup e_1$
16: $\quad$ **for all** $p_j \in {}^\bullet t$ **do**
17: $\quad\quad I_{p_j} := \{1, ..., |pre(t, p_j)|\}$
18: $\quad\quad e_2 := p_j(id, n) \wedge \bigwedge_{i \in I_{p_j}} id \neq id^j_i \rightsquigarrow \{p_j(id, n)\}$
19: $\quad\quad \textsc{gen}_t(\vec{v}, \vec{id}) := \textsc{gen}_t(\vec{id}, \vec{v}) \cup e_2$
20: $\quad$ **for all** $p_k \in t^\bullet$ **do**
21: $\quad\quad$ **for all** $\mathsf{x} \in post(t, p_k) \setminus \Upsilon$ **do**
22: $\quad\quad\quad$ **for all** $\mathsf{num_x} \in \{1, ..., post(t, p_k)(var)\}$ **do**
23: $\quad\quad\quad\quad e_3 := true \rightsquigarrow \{NewID(\mathsf{t}, \mathsf{p_k}, \mathsf{x}, genID(\mathsf{t}, \mathsf{p_k}, \mathsf{x}, \mathsf{num_x}), v)\}$
24: $\quad\quad\quad\quad \textsc{gen}_t(\vec{v}, \vec{id}) := \textsc{gen}_t(\vec{v}, \vec{id}) \cup e_3$
25: $\quad\quad$ **for all** $\nu \in post(t, p_k) \cap \Upsilon$ **do**
26: $\quad\quad\quad e_4 := true \rightsquigarrow \{NewID(\mathsf{t}, \mathsf{p_k}, \nu, genID(\mathsf{t}, \mathsf{p_k}, \nu, 1), genName(\nu)\}$
27: $\quad\quad\quad \textsc{gen}_t(\vec{v}, \vec{id}) := \textsc{gen}_t(\vec{id}, \vec{v}) \cup e_4$
28: $\quad$ **for all** $p_i \in P \setminus {}^\bullet t$ **do**
29: $\quad\quad e_5 := p_i(id, n) \rightsquigarrow \{p_i(id, n)\}$
30: $\quad\quad \textsc{gen}_t(\vec{id}, \vec{v}) := \textsc{gen}_t(\vec{id}, \vec{v}) \cup e_5$
31: $\quad \mathcal{A}_N := \mathcal{A}_N \cup \textsc{gen}_t(\vec{id}, \vec{v})$
32: $\textsc{transf}() := \emptyset$
33: **for all** $p_i \in P$ **do**
34: $\quad e_6 := p_i(id, n) \rightsquigarrow \{p_i(id, n)\}$
35: $\quad \textsc{transf}() := \textsc{transf}() \cup e_6$
36: **for all** $p_j \in P$ **do**
37: $\quad e_7 := NewId(_, \mathsf{p_j}, _, id, n) \rightsquigarrow \{p_j(id, n)\}$
38: $\quad \textsc{transf}() := \textsc{transf}() \cup e_7$
39: $\mathcal{A}_N := \mathcal{A}_N \cup \textsc{transf}()$
40: $\rho_N := \rho_N \cup \{FL \mapsto \textsc{transf}\}$

The proof is a variation of the simpler proof given in [BHCDM14] for the comparison between P/T nets and (lossy) DCDSs. To compare the states of $\Gamma_{\overline{N}}$ with those of $\Lambda_{\tau(\overline{N})}$, we define the following *name cardinality-equivalence* relation: given a marking m in $\Gamma_{\overline{N}}$ and a state s in $\Lambda_{\tau(\overline{N})}$, we say that m is name cardinality-equivalent to s, written $m \approx s$, if, for each place $p_i \in P$ and name $a \in Id$, $m(p)(a) = n$ if and only if $db(s)$ contains n tuples $\langle _, a \rangle$ in relation p. By definition, $m_0 \approx s_0$. It can then be shown, inductively, that, given m in $\Gamma_{\overline{N}}$ and s in $\Lambda_{\tau(\overline{N})}$ such that $m \approx s$, for every transition $t \in T$ (and corresponding action gen_t in $\tau(\overline{N})$):

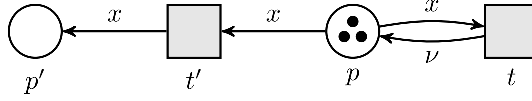

Fig. 7. A variation of the ν-PN in Figure 2, with a different initial marking, a new transition t', and a new place p'

- for every t, σ and m' s.t. $m[t,\sigma\rangle m'$, there exists a sequence $s \to s' \to s''$ in $\Lambda_{\tau(\overline{N})}$, where s' is produced by the application of $\textsc{gen}_t$ with parameter substitution σ', and where s'' is produced from s' by the application of $\textsc{transf}$, such that $m' \approx s''$;
- for every sequence $s \to s' \to s''$ produced by the application of action gen_t with parameter substitution σ, followed by the application of transf, there exists σ' and m' such that $m[t,\sigma'\rangle m'$ and $m' \approx s''$.

We now make two key observations. First, the two transition systems have the same structure, apart from the fact that each transition in $\Gamma_{\overline{N}}$ corresponds to a sequence of two transitions in $\Lambda_{\tau(\overline{N})}$, a feature that is correctly mirrored by ξ. Second, name cardinality-equivalence preserves the answers of local queries, i.e., given m and s such that $m \approx s$:

- for every number $c \in \mathbb{N}$ and every boolean $\mu\mathcal{L}_A^N$ query of the form $[\#p \leq c]$, we have that $[\#p \leq c]$ is true in m if and only if $\xi([\#p \leq c])$ is true in $db(s)$;
- for every number $c \in \mathbb{N}$, every open $\mu\mathcal{L}_A^N$ query of the form $[\#p(x) \leq c]$, and every substitution $\theta = [x/\mathsf{n}]$ ($\mathsf{n} \in Id$), formula $[\#p(x) \leq c]\theta$ is true in m if and only if $\xi([\#p(x) \leq c]\theta)$ is true in $db(s)$.

This concludes the proof. $\square$

We close this section by showing the full translation of a simple ν-PN into a corresponding DCDS.

Example 5.3. Let $\overline{N} = \langle P, T, F, m_0\rangle$ be a marked ν-PN, where: *(i)* $P = \{p, p'\}$, *(ii)* $T = \{t, t'\}$, *(iii)* $m_0 = \{p \mapsto [\bullet^3]\}$, and *(iv)* F is defined as follows:

$$F(e) = \begin{cases} x, & \text{if } e \in \{(p,t),(p,t'),(t',p')\}; \\ \nu, & \text{if } e = (t,p). \end{cases}$$

$\overline{N}$ is graphically depicted in Figure 7.

The application of τ to $\overline{N}$ produces the DCDS $\tau(\overline{N}) = \{\mathcal{D}_N, \mathcal{P}_N\}$. The data layer $\mathcal{D}_N = \{\mathcal{C}_N, \mathcal{R}_N, \mathcal{E}_N, \mathcal{I}_0^N\}$ is obtained from the application of Algorithm 1 to $\overline{N}$, which results in:

- $\mathcal{C}_N = \{\mathsf{t}, \mathsf{t}', \mathsf{p}, \mathsf{p}', \mathsf{x}, \nu\} \cup Id$;
- $\mathcal{R}_N = \{p/2, , p'/2, \mathit{NewID}/5, \mathit{FL}/0\}$;
- $\mathcal{I}_0 = \{p(1, \bullet)\}$;
- $\varepsilon_1 = \forall x.\mathit{NewID}(_,_,_,id,_) \wedge (p(id,_) \vee p'(id,_)) \to \bot$;
- $\varepsilon_2 = \forall id,t,p,v,n,t',p',v',n'.\mathit{NewID}(t,p,v,id,n) \wedge \mathit{NewID}(t',p',v',id,n') \to t = t' \wedge p = p' \wedge v = v' \wedge n = n'$;
- $\varepsilon_3 = \forall n.\mathit{NewID}(_,_,\nu,_,n) \wedge (p(_,n) \vee p'(_,n)) \to \bot$.

We do not report constraint ε_4, since it is not needed to handle this specific case, given the fact that $\overline{N}$ contains only one fresh name variable from Υ.

The process layer $\mathcal{P}_N = \{\mathcal{F}_N, \mathcal{P}_N, \rho_N\}$ is constructed using Algorithm 2, which produces in this case:

- $\mathcal{F}_N = \{\mathit{genID}(\mathsf{t},\mathsf{p},\mathsf{v},\mathsf{i}), \mathit{genFresh}(\nu)\}$;
- $\rho_N = \{p(id,x) \wedge \neg \mathit{FL} \mapsto \textsc{gen}_t(id,x)\} \cup \{p'(id,x) \wedge \neg \mathit{FL} \mapsto \textsc{gen}_{t'}(id,x)\} \cup \{\mathit{FL} \mapsto \textsc{transf}()\}$;
- $\mathcal{A}_N = \{\textsc{gen}_t(id,x), \textsc{gen}_{t'}(id,x), \textsc{transf}()\}$, where
  - $\textsc{gen}_t\,(id,x) = \left\{ \begin{array}{r} p(id',n) \wedge id' \neq id \rightsquigarrow \{p(id',n)\}, \\ p'(id',n) \rightsquigarrow \{p'(id',n)\}, \\ \mathit{true} \rightsquigarrow \{\mathit{NewID}(\mathsf{t},\mathsf{p},\nu,\mathit{genID}(\mathsf{t},\mathsf{p},\nu,1),\mathit{genName}(\nu))\}, \\ \mathit{true} \rightsquigarrow \{\mathit{FL}\} \end{array} \right\}$;
  - $\textsc{gen}_{t'}(id,x) = \left\{ \begin{array}{r} p(id',n) \wedge id' \neq id \rightsquigarrow \{p(id',n)\}, \\ p'(id',n) \rightsquigarrow \{p'(id',n)\}, \\ \mathit{true} \rightsquigarrow \{\mathit{NewID}(\mathsf{t},\mathsf{p}',\mathsf{x},\mathit{genID}(\mathsf{t},\mathsf{p}',x,1),x)\}, \\ \mathit{true} \rightsquigarrow \{\mathit{FL}\} \end{array} \right\}$;

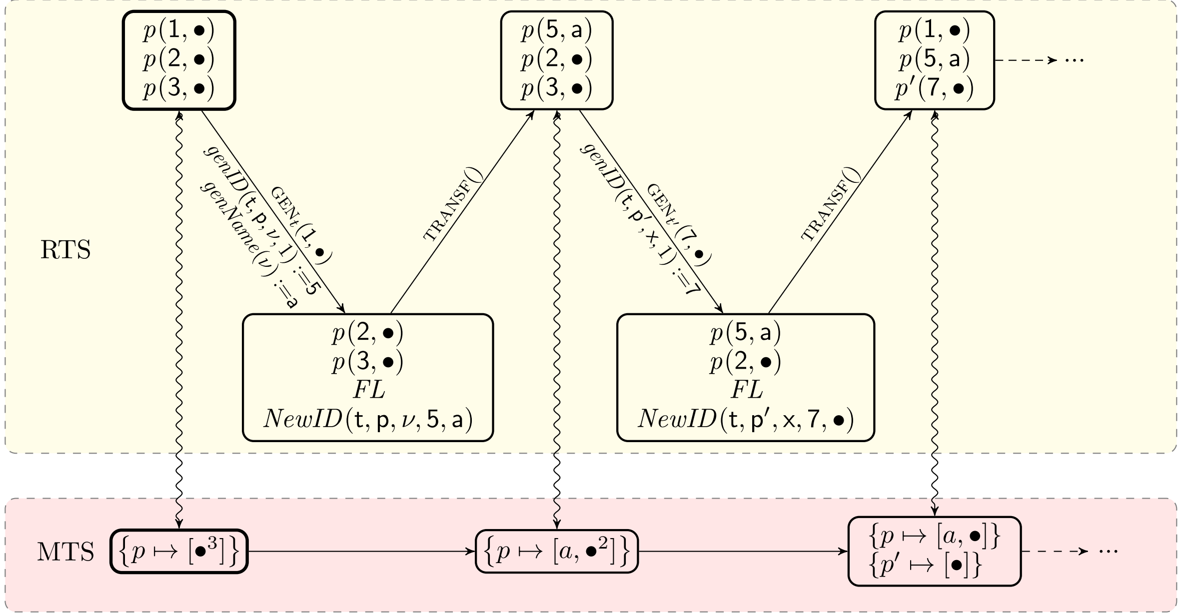

Fig. 8. Relationship between a (partial) run in the MTS of the marked ν-PN shown in Figure 7, and the corresponding run in the RTS of the DCDS obtained using our translation mechanism

– $\textsc{transf}() = \left\{ \begin{array}{r} NewId(_,\mathsf{p},_,id,n) \rightsquigarrow \{p(id,n)\} \\ NewId(_,\mathsf{p}',_,id,n) \rightsquigarrow \{p'(id,n)\} \\ p(id,n) \rightsquigarrow \{p(id,n)\} \\ p'(id,n) \rightsquigarrow \{p'(id,n)\} \end{array} \right\}$

The intuitive relationship between a run in the MTS of $\overline{N}$ and the corresponding run in the RTS of $\tau(\overline{N})$ is reported in Figure 8. ■

5.4. Proofs of Decidability

Lemma 5.1 is the key to prove the two decidability Theorems 5.1 and 5.2. We discuss the two proofs separately.

Proof of Theorem 5.1. From the proof of Lemma 5.1, we know that given a marked ν-PN $\overline{N}$, $\tau(\overline{N})$ faithfully reconstruct the execution semantics of $\overline{N}$. In particular, the states of $\Gamma_{\overline{N}}$ and those of $\Lambda_{\tau(\overline{N})}$ are connected by the name cardinality-equivalence relation. This directly implies that if $\overline{N}$ is run-bounded, then $\tau(\overline{N})$ is a run-bounded DCDS (in the sense defined in [BHCDG$^+$13]). The claim is then obtained by combining Lemma 5.1 with the result in [BHCDG$^+$13] that verification of $\mu\mathcal{L}_A$ properties over run-bounded DCDSs is decidable, and reducible to finite-state model checking.

Proof of Theorem 5.2. Let $\overline{N}$ be a ν-PN. By adopting the same line of reasoning of the proof for Theorem 5.1, we get that if $\overline{N}$ is state-bounded, then the DCDS $\tau(\overline{N})$ is state-bounded (in the sense defined in [BHCDG$^+$13]). It is easy to see that if the translation function ξ is applied to a $\mu\mathcal{L}_P^N$ formula Φ, the resulting formula $\xi(\Phi)$ is in $\mu\mathcal{L}_P$. The claim is then obtained by combining Lemma 5.1 with the result in [BHCDG$^+$13] that verification of $\mu\mathcal{L}_P$ properties over state-bounded DCDSs is decidable, and reducible to finite-state model checking.

6. Conclusion

We have studied the decidability boundaries related to the model checking of ν-PNs, i.e., Petri nets with (pure) name creation and management, against rich temporal properties specified using first-order variants

of the μ-calculus. To the best of our knowledge, this is the first attempt to study the challenging problem of formal verification of ν-PNs going beyond the investigation of general properties such as reachability, coverability, boundedness and termination. Decidability of verification is obtained for classes of ν-PNs that are still infinite-states, but whose runs or states contain a bounded amount of names (and a bounded amount of tokens in each marking). Notably, such semantic properties are decidable to check over ν-PNs, thanks to previous results [RVdFE11].

We have also shown that the ν-PN framework, and the corresponding model checking problems, can provide a solid basis for the analysis of concrete, novel settings in the area of workflow modeling and analysis. In particular, we have shown how the well-known paradigm of workflow nets [vdA97] can be enriched with explicit process instances and global resources, and how a suitably revised notion of soundness [vdAvHtH+11] can be formalized and checked with our approach.

The decidability results provided in this work are obtained via a translation from ν-PNs to the framework of DCDSs [BHCDG+13], developed to tackle the formal specification and verification of data-aware business processes working over full-fledged relational databases with constraints. In this light, our approach shows that interesting, novel results can be obtained by cross-fertilizing the research areas of formal methods for concurrent systems and that of foundations of data-aware processes, which have not been extensively related so far. In this article, we have mainly exploited the translation to transfer results from the setting of DCDSs to that of ν-PNs. However, we believe that the connection can also be fruitfully exploited in the opposite way, so as to carry interesting results for ν-PNs to the DCDS setting. For example, in [BHCDM14] it has been shown that the semantic property of state-boundedness is undecidable to check over DCDSs, even when their modeling capabilities are severely limited. Thanks to the decidability of state-boundedness for ν-PNs, and to the connection drawn in this work, we have implicitly isolated a class of DCDSs for which state-boundedness is decidable.

We plan to extend the approach presented in this work by foreseeing a systematic, bidirectional transfer of results between richer classes of colored Petri nets and DCDSs.

Acknowledgements

This work has been partially supported by the EU project *Optique* (FP7-IP-318338), and by the UNIBZ internal projects *KENDO* and *OnProm*.

References

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, CCS '97, pages 36–47, New York, NY, USA, 1997. ACM.

[AK95] Nikolay A. Anisimov and Maciej Koutny. On compositionality and petri nets in protocol engineering. In Piotr Dembinski and Marek Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 71–86. Chapman & Hall, 1995.

[BBKL12] Mikolaj Bojanczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. Towards nominal computation. In *Proc. of POPL*. ACM Press, 2012.

[BHCDG+13] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. of PODS*, pages 163–174. ACM, 2013.

[BHCDM14] Babak Bagheri Hariri, Diego Calvanese, Alin Deutsch, and Marco Montali. State boundedness in data-aware dynamic systems. In *Proc. of KR*, 2014.

[BHCM+13] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Description logic Knowledge and Action Bases. *J. of Artificial Intelligence Research*, 2013.

[BLP12] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. An abstraction technique for the verification of artifact-centric systems. In *Proc. of KR*, 2012.

[BST13] Mikolaj Bojanczyk, Luc Segoufin, and Szymon Torunczyk. Verification of database-driven systems via amalgamation. In *Proc. of PODS*, 2013.

[CDGM13] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data aware process analysis: A database theory perspective. In *Proc. of PODS*, 2013.

[CDGMP13] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. Verification and synthesis in description logic based dynamic systems. In *Proc. of RR*, volume 7994 of *LNCS*. Springer, 2013.

[CH80] A. Chandra and David Harel. Computable queries for relational database systems. *Journal of Computer and System Sciences*, 21, 1980.

[DW08] Gero Decker and Mathias Weske. Instance isolation analysis for service-oriented architectures. In *Proc. of SCC*, pages 249–256. IEEE Computer Society, 2008.

[Eme96] E. Allen Emerson. Model checking and the Mu-calculus. In *Proc. of the DIMACS Symposium on Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society Press, 1996.

[Esp94] Javier Esparza. On the decidability of model checking for several μ-calculi and petri nets. In *Proc. of CAAP*, volume 787 of *LNCS*, pages 115–129. Springer, 1994.

[GP02] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5), 2002.

[HMM13] Reiner Hüchting, Rupak Majumdar, and Roland Meyer. A theory of name boundedness. In *Proc. of CONCUR*, volume 8052 of *LNCS*, pages 182–196. Springer, 2013.

[JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[KR04] Michael Köhler and Heiko Rölke. Properties of object petri nets. In *Proc. of ICATPN*, volume 3099 of *LNCS*, pages 278–297. Springer, 2004.

[Lib04] Leonid Libkin. *Elements of Finite Model Theory*, volume 7360 of *LNCS*, chapter Fixed Point Logics and Complexity Classes. Springer, 2004.

[LNO+08] Ranko Lazic, Tom Newcomb, Joël Ouaknine, A. W. Roscoe, and James Worrell. Nets with tokens which carry data. *Fundam. Inform.*, 88(3):251–274, 2008.

[MR11] María Martos-Salgado and Fernando Rosa-Velardo. Dynamic soundness in resource-constrained workflow nets. In *Proc. of FMOODS/FORTE*, volume 6722 of *LNCS*, pages 259–273. Springer, 2011.

[Nee89] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 89–101. acm Press, Wokingham, 1989.

[Pet80] James L. Peterson. A note on colored petri nets. *Information Processing Letters, Vol.11, No.1*, 11:40–43, 1980.

[Rei13] Wolfgang Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[RVdFE08] Fernando Rosa-Velardo and David de Frutos-Escrig. Name creation vs. replication in petri net systems. *Fundam. Inform.*, 88(3):329–356, 2008.

[RVdFE11] Fernando Rosa-Velardo and David de Frutos-Escrig. Decidability and complexity of petri nets with unordered data. *Theor. Comput. Sci.*, 412(34):4439–4451, 2011.

[Var05] Moshe Y. Vardi. Model checking for database theoreticians. In *Proc. of ICDT*, volume 3363 of *LNCS*, pages 1–16. Springer, 2005.

[vdA97] Wil M. P. van der Aalst. Verification of workflow nets. In *Proc. of ICATPN*, volume 1248 of *LNCS*, pages 407–426. Springer, 1997.

[vdA05] Wil M. P. van der Aalst. Pi calculus versus petri nets: Let us eat humble pie rather than further inflate the pi hype. *BPTrends*, 3(5), 2005.

[vdAS11] Wil M. P. van der Aalst and Christian Stahl. *Modeling Business Processes - A Petri Net-Oriented Approach*. Cooperative Information Systems series. MIT Press, 2011.

[vdAvHtH+11] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe Thandar Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Asp. Comput.*, 23(3):333–363, 2011.

[vHSSV05] Kees M. van Hee, Alexander Serebrenik, Natalia Sidorova, and Marc Voorhoeve. Soundness of resource-constrained workflow nets. In *Proc. of ICATPN*, volume 3536 of *LNCS*, pages 250–267. Springer, 2005.

[vHSV06] Kees M. van Hee, Natalia Sidorova, and Marc Voorhoeve. Resource-constrained workflow nets. *Fundam. Inform.*, 71(2-3):243–257, 2006.

[Via09] Victor Vianu. Automatic verification of database-driven systems: a new frontier. In *Proc. of ICDT*, pages 1–13, 2009.

[WM11] Michael Westergaard and Fabrizio Maria Maggi. Modeling and verification of a protocol for operational support using coloured petri nets. In *Applications and Theory of Petri Nets - 32nd International Conference, PETRI NETS 2011, Newcastle, UK, June 20-24, 2011. Proceedings*, pages 169–188, 2011.