Fundamenta Informaticae nnn (2015) 1–48
DOI 10.3233/FI-2015-0000
IOS Press

Petri Nets with Structured Data

Eric Badouel
INRIA Rennes
eric.badouel@inria.fr

Loïc Hélouët [C]
INRIA Rennes
loic.helouet@inria.fr

Christophe Morvan
Université Paris-Est
christophe.morvan@u-pem.fr

Abstract. This paper considers Structured Data Nets (StDN): a Petri net extension that describes open systems with data. The objective of this language is to serve as a formal basis for the analysis of systems that use data, accept inputs from their environment, and implement complex workflows. In StDNs, tokens are structured documents. Each transition is attached to a query, guarded by patterns, (logical assertions on the contents of its preset) and transforms tokens. We define StDNs and their semantics. We then consider their formal properties: coverability of a marking, termination and soundness of transactions. Unrestricted StDNs are Turing complete, so coverability, termination and soundness are undecidable for StDNs. However, using an order on documents, and putting reasonable restrictions both on the expressiveness of patterns and queries and on the documents, we show that StDNs are well-structured transition systems, for which coverability, termination and soundness are decidable. We then show the expressive power of StDN on a case study, and compare StDNs and their decidable subclasses with other types of high-level nets and other formalisms adapted to data-centric approaches or to workflows design.

Keywords: Petri nets, Well-Quasi Orders, Structured Data

1. Introduction

Web services and business processes are now widely used systems. Many solutions exist to design such systems, but their formal verification remains difficult due to the tight connection of workflows with

Address for correspondence: INRIA Rennes Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France
[C]Corresponding author

data [1, 2, 3]. For instance, in an online shop, one faces situations where a workflow depends on data (*if the age of the client is greater than* 50*, then propose service* S), and conversely data depend on the way a workflow is executed (*return an offer with the minimal price proposed among the 5 first values returned by sub-contractors*). These systems have to be open: they must accept user inputs and manage multiple concurrent interactions. Openness also raises robustness issues: a system must avoid undesired interferences among distinct requests (a payment for some item should not trigger delivery of another unpaid item) and be robust for all inputs, including erroneous or malicious ones. Processing of a case (a request from an external user) is made of several steps that follow the logics of some business process, driven by the data attached to the processed request. Last, a web-based system such as a commercial website usually manages its own data, like catalogs, clients database or stock, which contents influences the execution of transactions.

Thus, exact descriptions of systems such as commercial sites or web-based information systems lead naturally to infinite state models with unbounded data, counters, and control flows that depend on the value of data or counters, that can only be captured by Turing powerful formalisms for which verification problems are undecidable. As a consequence, one has to work with abstractions of these systems to apply automated analysis techniques. Coarse grain approximations can rely on finite discretizations of data or on bounds on the number of ongoing requests in a system. These straightforward techniques allow one to get back to the familiar models of finite state systems or (variants of) Petri nets for which verification techniques are well-studied and decidable (model-checking for automata, coverability and reachability techniques for Petri nets). However, such bounded discretization that completely abstracts from data is usually too imprecise.

This paper introduces *Structured Data nets* (StDN), a variant of Petri nets where tokens are structured documents, and transitions transform these documents. A token represents a piece of information that either belongs to a database associated with the system, or is attached to some ongoing request. Each transition of an StDN is attached a query used to transform data. It is guarded by patterns expressing constraints on tokens in its input places. When firing a transition, the corresponding input documents are consumed and transformed by queries into new documents in its output places. new requests are introduced in the system using a designated input transition that non-deterministically produces new documents corresponding to the request. Termination of a request is symbolized by the consumption of a document by a designated output transition. We define structured documents as trees whose nodes carry information given by lists of attributes/values (*à la* XML). We show that considering documents of bounded depth labeled by well-quasi ordered values, one can provide a well-quasi ordering on documents. We define StDNs, their semantics, and consider their formal properties, such as coverability of a marking, termination and soundness of transactions. In their full generality, StDNs are Turing complete, so all these properties are undecidable. However, we prove that as soon as StDN manipulate well-quasi ordered documents, and meet some reasonable restrictions on the expressive power of patterns and queries (monotony with respect to ordering), StDNs are well-structured transition systems. If a well-structured StDN meets, in addition, effectiveness requirements (namely one can effectively compare markings, and effectively compute a finite representation for the set of predecessors of a marking), then coverability of a marking is decidable. As a consequence, termination and soundness are also decidable. All these properties hold for a single initial marking of a net, but can be extended to handle symbolically unbounded sets of initial markings satisfying constraints defined by a pattern. Even if some information systems can not be represented by well-structured StDNs, this decidable setting lies at a reasonable level of abstraction: it does not fix an *a priori* bound on the number of ongoing requests nor impose bounded

data domains.

There exist several Petri net variants to model business processes and transactions: workflow nets [4] or Jackson Nets [5], for example. However these models focus on processing a single case, and do not consider data. Furthermore, our model is not the first extension of Petri nets that handles data or complex types attached to tokens. Colored Petri nets [6] is a Turing powerful kind of Petri nets where tokens have colors ranging on unbounded sets, and transitions transform these colors. Nets that manipulate explicitly structured data such as PrT-Nets [7] or XML nets [8] have also been proposed. Considering decidability for nets with complex tokens and extended flow relations, there are variants of Petri nets with decidable coverability *e.g.* Petri nets with tokens carrying data [1], and nested nets [9]. Several subclasses of Generalized self-modifying nets, *i.e.*, nets with standard tokens and markings but polynomial flow relations also have decidable coverability [10]. These classes include transfer and reset Petri nets. Outside of the Petri net community, several models have also been proposed to specify systems with data and business processes: Active XML (AXML) [11], business artifacts [12], Guard-Stage-Milestones [13], ORC [14], BPEL [15], session systems [16], TPRS [2], Guarded attributed grammars [17] and others. One can also mention initiatives to model web-services in the π-calculus community [18, 19, 20, 21]. Section 6 performs an in-depth comparison of our work with several of these models, and with several extensions of Petri nets. In particular it compares decidable subclasses of StDNs with net variants for which coverability is decidable.

This paper is organized as follows: Section 2 introduces the basic elements of our model, namely documents and tree patterns. It then shows how documents can be ordered. Section 3 defines Structured Data Nets, and their semantics. Then it considers formal properties of this model, and in particular coverability of a marking, termination, and soundness of transactions. Section 4 illustrates the use of Structured Data nets on a case study, namely a travel agency. Section 5 compares StDNs with other high-level Petri nets extensions, and section 6 lists other models for web-based systems, business processes and systems handling data. Section 7 concludes this work and gives future lines of research.

2. Trees, Documents and their Ordering

2.1. Preliminaries

In this section, we introduce basic notations that will be used in the paper, and the key concepts of Well Quasi Orders and Well-Structured Transition Systems.

We denote by $\wp(A)$ the set of all subsets of A, and by $\mathcal{M}(A)$ the multisets with elements in set A. Every $X \in \mathcal{M}(A)$ is a map $X : A \to \mathbb{N}$, where $X(a)$ gives the multiplicity of element $a \in A$ in X. For a pair of multisets M_1, M_2, we denote by $M_1 \uplus M_2$ the multiset that contains $M_1(a) + M_2(a)$ occurrences of $a \in A$. We further let $\mathcal{M}_f(A)$ define the set of *finite multisets*, i.e., the subset of $\mathcal{M}(A)$ which contains the multisets X such that $X(a) \neq 0$ for only a finite number of elements $a \in A$.

For a given set X of elements, a relation R on X is a subset of $X \times X$. We denote by R^* the reflexive and transitive closure of R, that is, $(x, y) \in R^*$ iff $x = y$, or there exists $e_1, \dots, e_n \in X$ such that $(x, e_1) \in R$, $(e_n, y) \in R$, and $(e_i, e_{i+1}) \in R$ for every $i \in 1..n-1$. Two elements x, y of X are said incomparable (with respect to R) is $(x, y) \notin R$ and $(y, x) \notin R$. An *antichain* of R is a set of pairwise incomparable elements.

In the rest of the document, we will define Structured Data Nets, and show that for some subclasses of the model, the usual techniques to check coverability apply. Structured Data Nets will manipulate

semi-structured documents a la XML, that can be represented as labeled trees. A *tree* $T = (V, E, \text{root}_T)$ consists of a set V of vertices with a distinguished vertex, $\text{root}_T \in V$, called the *root* of the tree, together with a set of *edges* $E \subseteq V \times (V \setminus \{\text{root}_T\})$, such that: $i)$ every vertex $v \in V \setminus \{\text{root}_T\}$ has a unique predecessor, i.e. a vertex v' such that $(v', v) \in E$, and root_T has no predecessor, $ii)$ the set of edges is *acyclic*, $iii)$ for every vertex $v \in V \setminus \{\text{root}_T\}$ there exists a unique path from the root to v, i.e. a finite sequence $v_0, \ldots, v_n$ such that $v_0 = \text{root}_T$, $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq n$ and $v_n = v$. A node v is a *parent* of a node v' in a tree iff $(v, v') \in E$. A node v is an *ancestor* of a node v' iff $(v, v') \in E^*$. A tree is *labeled* in A if it comes equipped with a labeling function $\lambda : V \to A$. The *depth* of a tree T is the maximal length of a sequence of consecutive edges in T.

A binary relation $\leq$ over a set of elements X is a *well quasi order* (wqo) iff

- it is a quasi order: $\leq$ is reflexive and transitive (but not necessarily antisymmetric).
- it is *well-founded*: any infinite sequence $x_1, \ldots, x_n, \ldots$ contains two elements x_i and x_j such that $i < j$ and $x_i < x_j$.

Equivalently, a quasi order is a wqo if it contains no infinite strictly decreasing sequences nor infinite antichains. Let $\uparrow x = \{y \mid x \leq y\}$ denote the *upward closure* of an element x. Upward closure easily extends to sets of elements, i.e., $\uparrow X = \bigcup\limits_{x \in X} \uparrow x$. A set X is *upward closed* if $\uparrow X = X$. Any upward closed set in a wqo has a *finite* basis (a finite set $B(X) \subseteq X$ such that $\bigcup_{x \in B(X)} \uparrow x = X$). This property ensures the existence of a finite representation for infinite upward closed sets of elements.

A *transition system* is a tuple $(X, \rightarrow, x_0)$, where X is a set of configurations, $\rightarrow \subseteq X \times X$ is a *move* relation depicting how a system can move from one configuration to another, and x_0 is the initial configuration. A *well-structured transition system* (WSTS) is a tuple $(X, \leq, \rightarrow, x_0)$ where $(X, \rightarrow, x_0)$ is a transition system, $(X, \leq)$ is a wqo, and relations $\leq$, and $\rightarrow$ are *compatible*, that is $x \leq y$ and $x \rightarrow x'$ implies that there exists a sequence of moves $y \rightarrow y_1 \rightarrow \cdots \rightarrow y_k \rightarrow y'$ such that $x' \leq y'$. WSTS have good properties with respect to coverability. The *coverability* question is defined as follows: for a given pair of initial configuration x_0 and target configuration y, can one reach a configuration y' greater than y starting from x_0 ? With some additional properties, coverability in WSTS can be checked using a fixpoint algorithm that computes a basis for the set of all possible predecessors of upward closed sets of configurations [22, 23]. For an upward closed set X of configurations of a WSTS, we denote by $pre(X)$ the set of predecessors of elements of X by $\rightarrow$, i.e., $pre(X) = \{y \mid \exists x \in X, y \rightarrow x\}$. Coverability can then be rephrased as: does $x_0 \in Pre^*(\uparrow y)$? This definition does not suffice to provide an algorithm to check coverability, as $Pre^*(\uparrow y)$ is not necessarily finite. In addition to well-structuredness, a key property to guarantee is that one can represent sets of predecessors of upward closed sets with a finite basis, and compute this basis effectively. A WSTS has an *effective pred-basis* iff for any $x \in X$, there exists an algorithm that can compute a finite basis $PredB(x)$ for $Pre(\uparrow x)$.

A standard set saturation algorithm to check coverability of a configuration y from an initial configuration x_0 is as follows. The algorithm starts with a finite basis B_0 for the upward closed set $Y_0 = \uparrow y$. At each step i, the algorithm, computes a basis $B_i = B_{i-1} \cup PredB(\uparrow B_{i-1})$. A configuration y is coverable from x_0 iff there exists an index k and a element $b \in B_k$ such that $b \leq x_0$. The algorithm stops at step m if there exists $b \in B_m$ with $b \leq x_0$, or if $\uparrow B_{m+1} = \uparrow B_m$. In the latter case, B_m is a basis for $Pre^*(\uparrow y)$. It was proved in [22, 23] that this algorithm is correct and that it terminates for *effective* WSTS where effectiveness means that *i)* the comparison relation $\leq$ is effective and *ii)* (backward-effectiveness) the WSTS has an effective pred-basis.

2.2. Documents and Tree Patterns

Our model of net is a variant of Petri nets manipulating structured data. These data are encoded as trees, and queried using tree patterns and queries. Tokens of Structured Data Nets are *documents* represented by finite trees whose nodes are labeled with attribute/value pairs, i.e. by a finite set of equations of the form $a = v$ where tag a denotes a data field or an attribute and v its associated value. This is captured by the notion of *tag system*: a tag system lists possible attributes, and for each attribute, a domain of legal values. For instance, we can define a tag system with two attributes $name$ and age, whose domains are respectively strings of at most 255 characters and positive integers. More formally, a *tag system* $\tau = (\Sigma, \mathbb{D})$ consists of a set Σ of tags and a set $\mathbb{D}$ indexed by Σ such that for every $\sigma \in \Sigma$, the set $\mathbb{D}_\sigma$ of possible values for attribute σ is non-empty. A *valuation* associated with a tag system $\tau = (\Sigma, \mathbb{D})$ is a partial function $\nu : \Sigma \to \mathbb{D}$ whose domain of definition, denoted $\text{tag}(\nu)$, is finite and such that $\forall \sigma \in \text{tag}(\nu), \nu(\sigma) \in \mathbb{D}_\sigma$. We denote by Val_τ the valuations associated to tag system τ.

Definition 2.1. (Document)
A *document* $D \in \text{Doc}_\tau$ associated with a tag system τ is a finite tree labeled by valuations in Val_τ.

We denote by Doc_τ the set of all documents with tag system τ. If v is the node of a document, we let $\text{tag}(v)$ be a shorthand for $\text{tag}(\lambda(v))$ and let $v \cdot \sigma$ denote $\lambda(v)(\sigma)$ when $\sigma \in \text{tag}(v)$. We use *tree patterns* to address boolean properties of trees. A tree pattern is also a labeled finite tree, whose edges are partitioned into parent edges and ancestor edges, and whose nodes are labeled by constraints. A *constraint* with tag system $\tau = (\Sigma, \mathbb{D})$ is defined by a partial function $C : \Sigma \to \wp(\mathbb{D})$ whose domain, denoted $\text{tag}(C)$, is finite and such that $\forall \sigma \in \text{tag}(C), C(\sigma) \subseteq \mathbb{D}_\sigma$. We denote by Cons_τ the set of constraints with tag system τ. For convenience, we will often define constraints in an (in-)equational way. For instance if $\mathbb{D}_\sigma$ is the set of integers then $5 \leq \sigma \leq 20$ constrains the value of σ to lay within the set of integers ranging between 5 and 20. We also write $\sigma = ?$ to state that σ can take any value in $\mathbb{D}_\sigma$.

Definition 2.2. (Tree Pattern)
Let τ be a tag system. A *tree pattern* over τ is a tuple $P = (V, \text{Pred}, \text{Anc}, \lambda)$, where V is a set of nodes containing a particular root node root_P, $\text{Pred}, \text{Anc} \subseteq V \times V$ are disjoint set of edges and $(V, \text{Pred} \cup \text{Anc}, \text{root}_P, \lambda)$ is a finite tree labeled by constraints in Cons_τ.

Edges of Pre in a pattern represent pairs of nodes (v, v') in documents where v is the parent of v'. Edges of Anc in a pattern represent pairs of nodes (v, v') in documents where v is an ancestor of v'. For a fixed tag system τ, we denote by Pat_τ the set of patterns over τ. As for documents, we let $\text{tag}(v)$, for v a node of a tree pattern, be an abbreviation for $\text{tag}(\lambda(v))$ and let $v \cdot \sigma$ denote $\lambda(v)(\sigma)$ when $\sigma \in \text{tag}(v)$. We further use $v \cdot \sigma = ?$ as a shorthand for $v \cdot \sigma = \mathbb{D}_\sigma$ which means that v must carry the tag σ but the value of this tag is not constrained. This situation should not be confused with $\sigma \notin \text{tag}(v)$ which does not constrain node v to carry tag σ (see Figure 1 for an illustration).

We adopt the following graphical convention: Patterns will be represented as trees. Single edges in these trees will denote the parent relation, and double edges the ancestor relation. Nodes will be represented as sets of constraint equations. We will also denote by $* = \{\}$ the empty constraint: nodes tagged with $*$ in a pattern are attached a constraint function that is undefined for every $\sigma \in \Sigma$, i.e. they can be mapped to document nodes that carry any set of tags and any valuation.

Figure 1 is an example of tree pattern. It describes the set of trees which have at least five nodes v_0, v_1, v_2, v_3, and v_4 with the following properties. v_0 is the root of the tree, v_1 is not a leaf node (i.e.

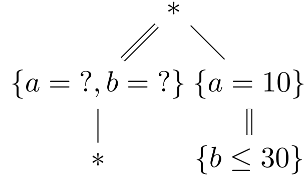


Figure 1. A tree Pattern

it has at least one successor node v_2) and it carries tags a and b (tag$(v_1) \supseteq \{a, b\}$) with no particular constraints on their values: $\lambda(v_1)(a) = \mathbb{D}_a, \lambda(v_1)(b) = \mathbb{D}_b$. Node v_3 is an immediate successor of the root, it carries tag a (tag$(v_2) \supseteq \{a\}$) and the value attached to tag a is 10. Node v_4 is some successor node of v_3 tagged by b and the value attached to b is lower than 30. Pattern satisfaction is formally defined as follows:

Definition 2.3. (Pattern Satisfaction)
A document $D = (V_D, E_D, \mathrm{root}_D, \lambda)$ satisfies a tree pattern $P = (V_P, \mathrm{Pred}_P, \mathrm{Anc}_P, \lambda_P)$, denoted $D \models P$, when there exists an injective map $h : V_P \to V_D$ such that:

1. $h(\mathrm{root}_P) = \mathrm{root}_D$,
2. $\forall v \in V_P \quad \mathrm{tag}(v) \subseteq \mathrm{tag}(h(v))$,
3. $\forall v \in V_P \; \forall \sigma \in \mathrm{tag}(v) \quad h(v) \cdot \sigma \in v \cdot \sigma$,
4. $\forall (v, v') \in \mathrm{Pred}_P,\ (h(v), h(v')) \in E_D$,
5. $\forall (v, v') \in \mathrm{Anc}_P,\ (h(v), h(v')) \in E_D^*$ (where E_D^* is the reflexive and transitive closure of E_D), and
6. $\forall (v, v') \notin (\mathrm{Pred}_P \cup \mathrm{Anc}_P)^*, (h(v), h(v')) \notin E_D^*$

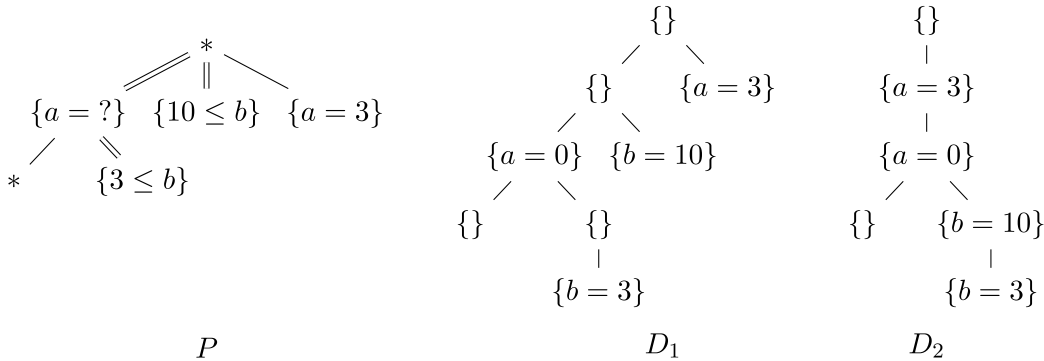


Figure 2. A tree pattern P and two documents D_1, D_2

Remark 2.4. Let us comment on Condition 6 of Def. 2.3. This requirement means that if two nodes do not belong to a single path in a pattern, then their image must not appear in a single path in the documents

that satisfy the pattern. This definition of pattern matching is called *injective* matching. Non-injective pattern matching has also been proposed in the litterature, but injective patterns are more expressive than their non-injective counterpart [24]. For instance, non-injective patterns cannot express that a document contains a node with two children tagged a. Another advantage of working with injective patterns is that one can check more efficiently that a document satisfies a pattern. We refer readers to [24] for more information on injective patterns and on the complexity of related algorithms. We hence define Structured Data Nets using an injective notion of pattern satisfaction. In the rest of the paper, this will also simplify notations and proofs, as documents satisfying a pattern can be obtained form the structure of the pattern by choosing appropriate values for constrained nodes and adding nodes to the pattern between nodes in ancestor relation, or by adding totally new subtrees. To illustrate Def. 2.3, consider the pattern P and the documents D_1, D_2 in Figure 2. Document D_1 satisfies pattern P and document D_2 does not satisfy P even though one can find an injective map h that satisfies Conditions 1 to 5 of Def. 2.3. This map however does not satisfy Condition 6 since it forces all immediate successors of the root of the pattern to match with nodes that belong to the same path.

According to [25], tree patterns are already a reasonable subset of the XPATH standard [26], even if they do not embed its whole power. Tree patterns are hence a good tradeoff between syntax simplicity and expressiveness to query documents in StDNs. Requiring pattern matching to hold at the root of a document is not a limitation. Indeed, for a pattern P with root v, one can design a new pattern P' that has an additional node v' such that $(v', v) \in \mathrm{Anc}$ and $\lambda_{P'}(v') = \{\}$. Then, P' holds at the root of a document D iff P holds *at some node* of D. On the other hand, using a child relation in patterns implies that matching is not a simple embedding relation (in the usual sense used for graphs): at some places, an edge-preserving embedding is required. This will have an impact to define an ordering on documents together with a consistent notion of pattern monotony, as shown in next section.

2.3. Ordering Trees

We do not distinguish between isomorphic trees, i.e. when there exists a bijection $\varphi : V_T \to V_{T'}$ between their respective sets of vertices such that $(v, v') \in E_T \iff (\varphi(v), \varphi(v')) \in E_{T'}$ (and thus also $\varphi(\mathrm{root}_T) = \mathrm{root}_{T'}$), and $\lambda(v) = \lambda(\varphi(v))$.

If $(A, \leq)$ is an ordered set (resp. a quasi ordered set, i.e. $\leq$ is a reflexive and transitive relation) then the set of trees labelled in A can be ordered (resp. quasi ordered) by setting $T_1 \leq T_2$ for any pair of trees $T_1 = (V_1, E_1, \mathrm{root}_1, \lambda_1)$, $T_2 = (V_2, E_2, \mathrm{root}_2, \lambda_2)$, when there exists an injective map $f : V_1 \to V_2$ such that:

1. $f(\mathrm{root}_1) = \mathrm{root}_2$,
2. $(v, v') \in E_1 \implies (f(v), f(v')) \in E_2$, and
3. $\forall v \in V_1, \lambda_1(v) \leq \lambda(f(v))$.

Hence $T_1 \leq T_2$ if T_2 can be obtained from T_1 by adding new edges and/or replacing existing labels by greater ones. For instance, given an order relation, $\leq_\sigma$, on $\mathbb{D}_\sigma$ and a subset of tags, $\Sigma' \subseteq \Sigma$, one obtains a quasi order on Doc_τ associated with the quasi order on valuations Val_τ given by:

$$\nu \leq_{\Sigma'} \nu' \iff \mathrm{tag}(\nu) \cap \Sigma' \subseteq \mathrm{tag}(\nu') \wedge \forall \sigma \in \mathrm{tag}(\nu) \cap \Sigma', \nu(\sigma) \leq_\sigma \nu'(\sigma)$$

Thus, in restriction to tags in Σ', valuation ν' has a larger domain and associates greater values to tags for which both ν and ν' are defined (see Figure 3 for an illustration). Note that $\Sigma' \subseteq \Sigma'' \implies \leq_{\Sigma''} \subseteq \leq_{\Sigma'}$.

Definition 2.5. (Monotony)
A pattern P is *monotonous* if, for any pair of documents (D_1, D_2), $D_1 \leq_P D_2$ and $D_1 \models P$ implies $D_2 \models P$ where $\leq_P \stackrel{\text{def}}{=} \leq_{\Sigma_P}$ is the order associated with the set Σ_P of tags occurring in P.

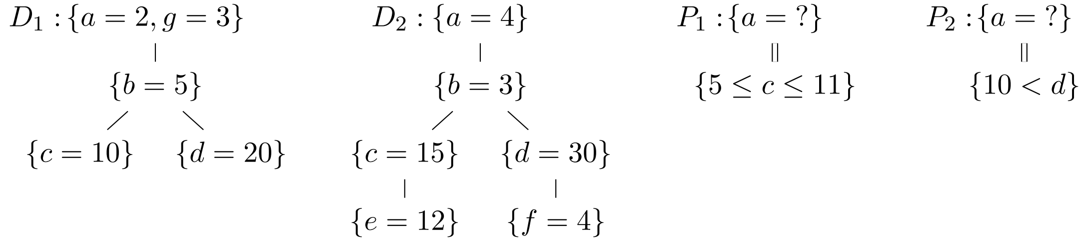


Figure 3. **Documents and patterns:** Assume all the domains $\mathbb{D}_\sigma$ are given by the set $\mathbb{N}$ of natural numbers with their usual ordering, then $D_1 \leq_{\{a,c,d\}} D_2$. Pattern P_1 is not monotonous since $D_1 \leq_{\{a,c\}} D_2$, $D_1 \models P_1$ and $D_2 \not\models P_1$. Pattern P_2 is monotonous.

As illustrated in Figure 3, a pattern that imposes upper bounds on attribute values is not monotonous. Finding a wqo on structured data can serve to finitely represent collections of data of arbitrary sizes, or to allow symbolic manipulations on families of trees. However, in contrast with Kruskal's theorem, which states that tree *embedding* is a well quasi order on the set of finite trees, the set $(\mathrm{Doc}_\tau, \leq_{\Sigma'})$ is in general not a wqo even if the set of tags is finite and their domains are finite or well quasi ordered. In fact, $(\mathrm{Doc}_\tau, \leq_{\Sigma'})$ is a *strict rooted inclusion*. Very often, tree comparison is defined as Kruskal's tree embedding, that only requires existence of a mapping that maps vertices in parent relation onto vertices in ancestor relation. We can not use this comparison relation on documents, as the ancestor relation is not precise enough to account for structural differences in databases or capture the notion of subfield. As a counterpart strict rooted inclusion allows sets of pairwise incomparable elements of arbitrary sizes (as shown in Figure 4).

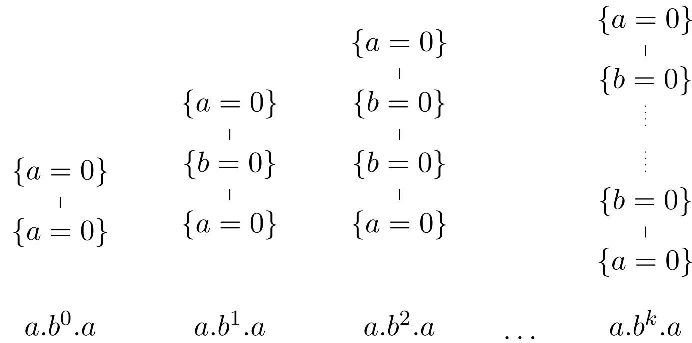


Figure 4. Let us consider tag system $\tau = (\{a, b\}, \mathbb{D})$, with $\mathbb{D}_a = \mathbb{D}_b = \{0\}$ and the trees shown above, denoted $a.b^k.a$, whose roots v_0, tagged a with $\lambda(v_0)(a) = 0$, are followed by a sequence $v_1, \dots, v_k$ of nodes tagged b with value $\lambda(v_i)(b) = 0$, and end with a node v_{k+1} tagged a, with $\lambda(v_{k+1})(a) = 0$. The set of trees $\{a.b^k.a \mid k \in \mathbb{N}\}$ consists of pairwise incomparable elements for $\leq_{\{a,b\}}$, hence they form an infinite antichain, whereas they form a chain for standard tree embedding.

This problem can be avoided by restricting to trees of bounded depth. Let us denote by $\mathrm{Doc}_{\tau,\leq n}$ the set of documents whose depth is less than or equal to n. In order for $(\mathrm{Doc}_{\tau,\leq n}, \leq_\Sigma)$ to be a wqo one must also assume that the set of tags, Σ, is finite. If it is not the case, the family of trees reduced to their root and all labelled with distinct tag would constitute an infinite antichain.

Proposition 2.6. Let $\tau = (\Sigma, \mathbb{D})$ a tag system where Σ is a finite set, $\Sigma' \subseteq \Sigma$, and $n \in \mathbb{N}$. If, for all $\sigma \in \Sigma'$, $(\mathbb{D}_\sigma, \leq_\sigma)$ is a wqo then $(\mathrm{Doc}_{\tau,\leq n}, \leq_{\Sigma'})$ is a wqo.

Proof:
First, note that since two documents that only differ on tags that do not belong to Σ' are equivalent for the equivalence relation induced by the quasi order $\leq_{\Sigma'}$, one can assume without loss of generality that $\Sigma' = \Sigma$. We know by [27] that the set of graphs $\mathcal{G}^n_\Sigma$, of bounded depth labelled by well quasi ordered tags, and ordered by strict subgraph inclusion $\leq$ is a well quasi order. Therefore the same result holds for trees of bounded depth labelled by wqo, ordered by rooted strict subgraph inclusion $\leq^r$. Indeed one has $T \leq^r T' \iff \overline{T} \leq \overline{T'}$ where $\overline{T}$ is obtained from T by adding a node labelled with a new symbol and by adding an edge from this node to the root of T. This additional node is the root of $\overline{T}$ and any strict labelled-graph embedding from $\overline{T}$ to $\overline{T'}$ necessarily relates their roots (because of their common label which does not appear elsewhere) and therefore also their unique successor nodes, i.e. the roots of T and T'. So it remains to prove that the order relation $\nu \leq_\Sigma \nu' \iff \mathrm{tag}(\nu) \subseteq \mathrm{tag}(\nu') \quad \wedge \quad \forall \sigma \in \mathrm{tag}(\nu) \quad \nu(\sigma) \leq_\sigma \nu'(\sigma)$ on valuations Val_τ is a wqo. This order relation can be expressed as: $\nu \leq_\Sigma \nu' \iff \forall \sigma \in \Sigma' \quad \nu(\sigma) \leq^\perp_\sigma \nu'(\sigma)$ where a valuation is viewed as a function $\nu : \Sigma \to \mathbb{D} \cup \{\perp\}$ where $\perp$ is a new element added to each of the sets $\mathbb{D}_\sigma$ as a least element ($x \leq^\perp_\sigma y \iff x = \perp \vee x \leq_\sigma y$) and by letting $\nu(\sigma) = \perp \iff \sigma \notin \mathrm{tag}(\nu)$. Then $(\mathbb{D}_\sigma \cup \{\perp\}, \leq^\perp_\sigma)$ is a wqo for every $\sigma \in \Sigma'$. As the Cartesian product of a *finite* family of wqos is a wqo, we have that $(\mathrm{Val}_\tau, \leq_\Sigma)$ is a wqo. □

Working with well-quasi ordered sets of documents has many advantages. It allows to manipulate infinite (upward closed) sets of documents represented by their basis.

Proposition 2.7. One can effectively compute a basis $\mathrm{BSat(P)}$ of the set $\hat{P} = \{D \in \mathrm{Doc}_{\tau,\leq n} \mid D \models P\}$ of documents, with depth at most n, satisfying a monotonous pattern P, if the order $\leq_P$ associated with the pattern is a wqo.

Proof:
Let $D \in \hat{P}$ be a document that satisfies P, and $h : V_P \to V_D$ be an injective map that witnesses $D \models P$ according to Def. 2.3, denoted as $h : D \models P$. Note that this map may not be unique. We then define the *h-reduction* of document D as the document $red(h, D)$ obtained by the following transformations on D: *(i)* we remove all nodes v such that the subtree rooted at this node lies outside the image of h, and for every remaining node v: *(ii)* we restrict its valuation to the set of tags $\mathrm{tag}(v')$ if $v = h(v')$, and *(iii)* we replace its valuation by the empty valuation (i.e., we remove all its tags) if it does not belong to the image of h. Then one has $h : red(h, D) \models P$ by Def. 2.3 and $red(h, D) \leq_P D$ by definition of $\leq_P$. Thus the set $\mathrm{BSat(P)}$ of minimal elements of $\{red(h, D) \mid D \in \mathrm{Doc}_{\tau,\leq n} \quad \wedge \quad h : D \models P\}$ is a basis of $\hat{P}$. As P is a monotonous pattern (see Def. 2.5), $\hat{P}$ is upward-closed. It thus has a finite basis —because $\leq_P$ is a wqo. $\mathrm{BSat(P)}$ is thus a finite set because its elements are all pairwise incomparable and form a basis of $\hat{P}$. Let us now detail how to compute $\mathrm{BSat(P)}$. The elements of $\mathrm{BSat(P)}$ are documents D

with depth at most n associated with a map $h : D \models P$ such that for each node of D that belongs to the image of h, i.e., $v = h(v')$ for some node v' of P, one has $\mathrm{tag}(v) = \mathrm{tag}(v')$ and $v \cdot \sigma$ is a minimal element for the upward closed set of values satisfying $v' \cdot \sigma$. Each node v of D that does not belong to the image of h lies on a path between two nodes that belong to the image of h and has an empty set of tags. By Remark 2.4 and in view of Condition (6) in Def. 2.3, documents in $\mathrm{BSat(P)}$ are derived from the pattern by the rewrite system given below. We encode a pattern P with an expression $\llbracket P \rrbracket$ where $\llbracket P \rrbracket = C \big\{ \llbracket P_1 \rrbracket, \cdots, \llbracket P_n \rrbracket \big\}$ if C is the constraint attached to the root node and $P_1, \ldots, P_n$ are the patterns rooted at the successor nodes of the root of P. Note that there is no ordering on $P_1, \ldots, P_n$, that describe subtrees in an unordered setting. The expression $\big\{ \llbracket P_1 \rrbracket, \cdots, \llbracket P_n \rrbracket \big\}$ hence designates a set of sub-expressions, and we can safely write expressions of the form $C \big\{ \llbracket P_1 \rrbracket, \cdots, \llbracket P_n \rrbracket \cup \llbracket P'_1 \rrbracket, \cdots, \llbracket P'_{n'} \rrbracket \big\}$. We further refine this encoding by associating an integer k to every node of the pattern. We hence refine our notation and consider constraints of the form (C, k). In (C, k), number k indicates the maximal number of nodes that can be inserted in the path from $h(v')$ to $h(v)$ where v' is the predecessor of v in the pattern. More precisely, we let $k = 0$ if v is the root of the pattern or if the edge from v' to v is a predecessor edge. We let $k = n - (d_1 + d_2 + 1)$ when the edge from v' to v is an ancestor edge, d_1 is the depth of v' in the pattern (the length of the path from the root to v'), and d_2 is the depth of the subtree of P rooted at node v. For instance the pattern in Fig. 2 is encoded by expression:

$$(*, 0) \big\{ (\{a = ?\}, 2) \big\{ (*, 0) \big\{ \ \big\}, (\{3 \leq b\}, 2) \big\{ \ \big\} \big\}, (\{10 \leq b\}, 3) \big\{ \ \big\}, (\{a{=}3\}, 0) \big\{ \ \big\} \big\}$$

If E is such an expression, or any subexpression thereof, we let $\min(E)$ denote the least integer k such that (C, k) occurs in E and $E - 1$ denote the expression obtained by replacing each subexpression (C, k) of E by $(C, k-1)$ assuming that $\min(E) \geq 1$. We have two kinds of rules which we apply in prefix order, i.e. we always apply reductions of a predecessor node prior to the reduction of a node. The first rule schema is

$$(C, k) \big\{ \mathrm{expr} \big\} \to V \big\{ \mathrm{expr} \big\}$$

where V is a "minimal" valuation compatible with constraint C, i.e. such that $\mathrm{tag}(V) = \mathrm{tag}(C)$ and for every $\sigma \in \mathrm{tag}(C)$, $V \cdot \sigma$ is a minimal element of $C(\sigma)$. Application of that rule explicitly creates the node of the document associated with the given node of the pattern. The second rule schema corresponds to the insertion of an intermediate node, thus associated with an empty valuation (represented by * symbol):

$$V \big\{ \{ (C_i, k_i) \big\{ E_i \big\} \mid 1 \leq i \leq K \} \big\} \to V \big\{ * \{ (C_i, k_i - 1) \{ E_i - 1 \} \mid i \in I \} \cup \{ (C_j, k_j) \{ E_j \} \mid j \notin I \} \big\}$$

where V is a valuation and $I \subseteq \{1, \ldots, K\}$ is associated with a subset of successor nodes of the given node in the pattern such that $k_i \geq 1$ and $\min(E_i) \geq 1$ for every $i \in I$. Rewriting ends after a certain number of steps with an expression defining a document of $\mathrm{BSat(P)}$. The key idea behind the rewriting is that an expression defines a reduced tree that embeds P, obtained by replacing a constraint by the mimimal value satisfying it. Inserting an unconstrained node in an expression amounts to defining a larger tree by inserting a new node between images of nodes that are in ancestor relation in the pattern, as soon as this transformation does not creates a tree of depth greater than n. An embedding relation remains a valid embedding after this insertion. The obtained tree is still reduced, as new nodes are inserted between images of nodes of P. Now, the set of nodes appearing in a tree of $Bsat(P)$ is bounded (it is a subset of trees of depth at most n with the same number of leaves as P), and one can show that any tree in the set of minimal reduced trees with k nodes satisfying P can be obtained by insertion of a node in a tree from the set of minimal reduced trees with $k - 1$ nodes. In rewritings, non-deterministic choice of an insertion position and of children of the new node ensures that all insertion possibilities

are considered. Hence, non-deterministic application of these rewriting rules generates all reduced trees, that embed P, carry minimal values allowing satisfaction of constraints attached to nodes of P, and have depth at most n. □

3. Structured Data Nets

3.1. Definitions and semantics

We use Structured Data Nets, StDNs for short, to model complex workflows with data, such as interactions within a webstore. Throughout the paper, we will use the term "transaction" to denote a workflow that starts with an input of a user to the system, and ends when the input is completely processed, and a result is returned to the user. This is a slight abuse of the term "transaction", that is frequently understood as an interaction that meets ACID properties (Atomicity, Consistency, Isolation, Durability). StDNs is a variant of Petri nets where tokens are documents, and transitions transform these documents. Each document is a piece of information that either belongs to a database owned by the system, or is part of the values computed to answer some ongoing request, in which case documents are attached a unique identifier associated with that request. Since the value of a token is a structured document, the current state of a transaction is a distributed document: The value of all the tokens associated with the transaction's identifier. Documents follow a workflow, are transformed by transitions of the nets, and aggregate data collected in the system. During its execution the workflow may create new documents or conversely assemble pieces of data referring to the same transaction. This allows for the execution of several parallel threads assembling data for the same request. Note also that during the execution of a workflow, some information from the case (client's name, ...) can be stored in the system for later use.

For convenience, we distinguish two particular transitions that are used to initiate and terminate transactions. A transition t_{in}, with no incoming place, delivers to the input place p_{in} a token representing a new transaction. A transition t_{out}, with no outgoing place, unconditionally consumes any token from the output place p_{out}. The consumption of a token by t_{out} represents the ending of the corresponding transaction, and the value of the token is the result that is returned to the caller, who initiated the transaction.

A frequent ACID property required in transactional systems is to ensure *isolation of transactions:* execution of two concurrent transactions results in a system state that would be obtained if transactions were executed one after the other. This property is rather strong, and we want to ensure a weaker notion of isolation, that ensures that two different transactions cannot mix. For instance, paying for ordered items should not trigger delivery of someone else's items in another transaction. This property can be ensured using session numbers. We hence assign an identifier to every token. More precisely, a token is a pair $T = (D, id)$ where D is a document, the value of the token, and $id \in \mathbb{N}$ is an identifier. When $id = 0$, it indicates that the data D is part of the local database of the system. Otherwise, $id \neq 0$ provides the identifier of the transaction that D belongs to. Thus identifiers of transactions are always positive integers. In many cases, however, we do not need to know the identifier of each individual transaction. A mechanism to distinguish transactions suffices. For instance such a mechanism is used in [16] where sessions are identified with specific components of the current configuration of the system. In Structured Data Nets, tokens' identifiers induce a partition on the set of tokens. Knowing the partition suffices for many purposes. Formalisms such as BPEL [28] use a more elaborated mechanism, called *correlations*, to filter and group messages sharing commonalities.

In short, StDNs are Petri nets whose input arcs are constrained by *patterns*, and whose output arcs are associated with *queries*. More precisely, each input arc (p,t) for $p \in {}^{\bullet}t$ is attached a *guard* given by a tree pattern $\langle p,t\rangle$. A useful pattern, denoted P_{tt} (tt stands for "true") throughout the document, contains a unique node v_0 with an empty set of constraints. This pattern is thus satisfied by any document and any input arc with $\langle p,t\rangle = P_{tt}$ simply checks existence of a document in place p. Transition t is enabled in a marking M if in every of its input place $p \in {}^{\bullet}t$ one can find a token $T_p = (D_p, id_p) \in M(p)$ such that $D_p \models \langle p,t\rangle$ and all non-zero identifiers id_p coincide. The latter condition ensures that all the pieces of information, but those belonging to the local database, are pieces of data that belong to the same transaction. When t fires, these tokens are removed from the current marking and some new tokens are added to every output place $p \in t^{\bullet}$. For that purpose, each output arc (t,p) for $p \in t^{\bullet}$ is attached a query $\langle t,p\rangle$ that describes how to compute the value of the token(s) to add in place $p \in t^{\bullet}$ from the vector of input documents $(D_p)_{p\in {}^{\bullet}t}$ which enabled the firing of the transition. Newly created documents are attached the common identifier found for transition firing or identifier 0 if they are stored in the local database of the system. Queries can produce multisets of tokens.

Definition 3.1. (Query)
An n-ary *query* $Q : (\mathrm{Doc}_\tau)^n \rightarrow \wp(\mathcal{M}_f(\mathrm{Doc}_\tau))$ is a function that non-deterministically produces a finite multiset of documents from a vector of documents given as input. We denote by $\mathrm{Im}(Q)$ the *image* of function Q. A query is *simple* when it non-deterministically returns a unique document: $\mathrm{Im}(Q) \subseteq \wp(\mathrm{Doc}_\tau)$. A query is *deterministic* if it returns a unique multiset of documents: $\mathrm{Im}(Q) \subseteq \mathcal{M}_f(\mathrm{Doc}_\tau)$.

We denote by $\mathcal{Q}_\tau$ the set of queries that manipulate documents from Doc_τ. We furthermore assume that queries are always effective functions. We do not fix a particular syntax for queries: they can be implemented with standard query languages such as XQuery [29], as tree transductions,... A query always returns a result, but this result can be an empty multiset. Non-simple queries can be used to produce several documents, or several copies of a same document. Let us illustrate this situation with a car insurance broking system. Figure 5 depicts a marking before firing of transition t. Place p_{cars} contains structured documents depicting cars and their prices. In the represented marking, p_{cars} contains a single description of a car, with identifier 1235. Place p_{comp} is a local database. A document in p_{comp} lists several insurance companies. The patterns $\langle p_{cars}, t\rangle = P_{tt}$ and $\langle p_{comp}, t\rangle = P_{tt}$ attached to input flows of transition t simply check the existence of documents in their respective places.

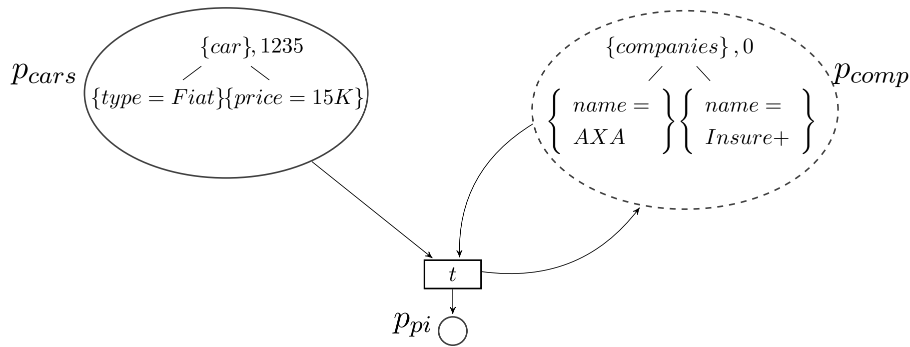


Figure 5. A StDN depicting part of a broking system for car insurance.

The place p_{pi} is the starting point to ask pro-forma invoices to companies. For a chosen car, transition

t creates one structured document in place p_{pi} per insurance company that appears in the database, by application of query $\langle t, p_{pi} \rangle$ attached to flow arc from t to place p_{pi}. Hence, $\langle t, p_{pi} \rangle$ is not simple. To leave the database unchanged by transition t, we attach to flow from t to place p_{comp} a query $\langle t, p_{comp} \rangle$ that simply copies the document used from place p_{comp}: Assuming an ordering on arguments where document from p_{cars} precedes the document from p_{comp}, we let $\langle t, p_{comp} \rangle(d_1, d_2) = d_2$. The marking obtained by firing transition t is depicted in Fig. 6.

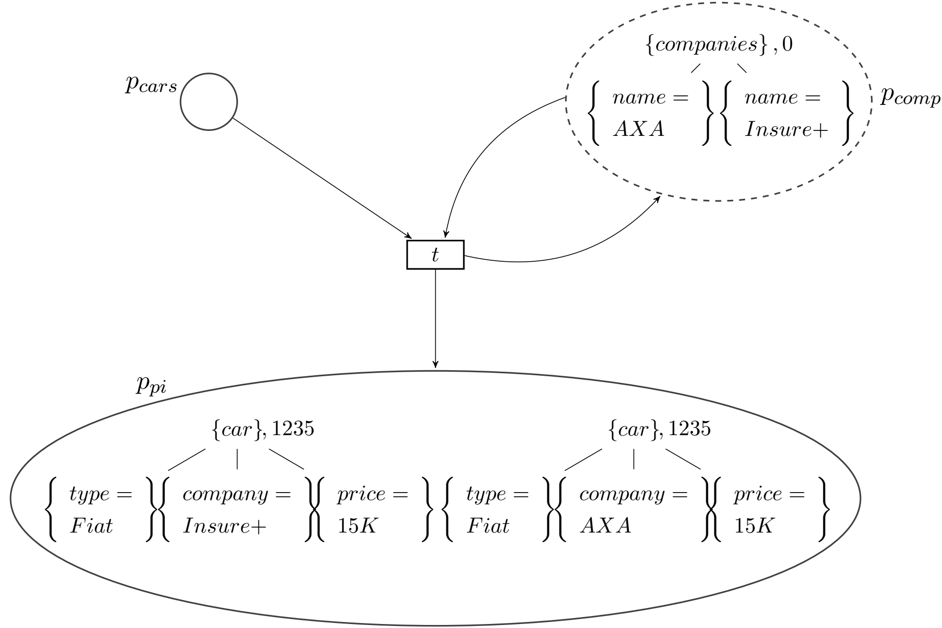


Figure 6. Marking reached by firing transition t in marking shown in Fig. 5.

Non-deterministic queries can be used to specify non-deterministic choices of the environment. This is illustrated by Figure 7, that models a part of an online shop in which a payment for some bought item needs to be granted by a bank. The marking represented in the left part of the figure contains an order awaiting for a clearance from a bank in place p. Transition BankDecision models this decision. The query $\langle \text{BankDecision}, p' \rangle$ attaches a new child to the document's root indicating bank's decision with a boolean valuation attached to tag $granted$. Hence, it non-deterministically returns the input document augmented with either a *true* or a *false* boolean tag. A possible result obtained after firing transition BankDecision is given in the right part of the figure.

As written earlier, we volontarily do not fix any specific query language. Our purpose is to define generic properties of nets. These properties depend on those of documents, queries specifications, and flow structure. But we aim at abstracting away the query language as much as possible. Several mechanisms have been proposed to query structured data. Standard query languages such as XQuery [29] and Xpath [26] use patterns to extract information from trees, and are usually described formally as tree pattern queries. The definition of structured data nets is as follows:

Definition 3.2. (Structured Data Net)
Let τ be a tag system. A *structured data net*, or StDN, is a structure $\mathcal{N} = (P, P_{DB}, T, F, \langle \cdot, \cdot \rangle)$ where P is a set of places, containing two particular places p_{in} and p_{out}, $P_{DB} \subseteq P$ is a subset of places corre-

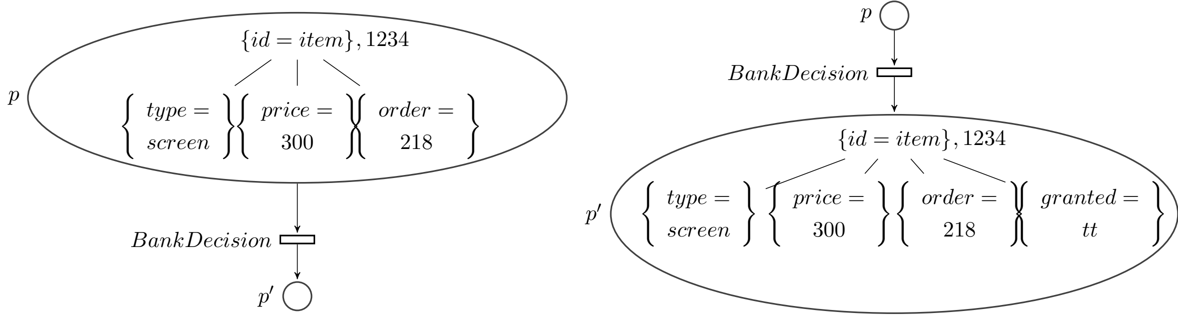


Figure 7. Modeling non-deterministic choices of the environment with non-deterministic queries.

sponding to the *local database* of the net, T is a set of transitions, containing two particular transitions t_{in} and t_{out} $F \subseteq P \times T \cup T \times P$ is a set of flow arcs, and map $\langle\cdot,\cdot\rangle : F \to \mathrm{Pat}_\tau \cup \mathcal{Q}_\tau$ associates each input arc $(p,t) \in F$ to a pattern $\langle p,t\rangle \in \mathrm{Pat}_\tau$ and each output arc $(t,p) \in F$ to a query $\langle t,p\rangle \in \mathcal{Q}_\tau$.

For any element of $x \in P \cup T$, we call the *preset* of x the set of its input elements, and define it as ${}^\bullet x = \{y \mid (y,x) \in F\}$. Similarly, the *postset* of x is the set of output elements from x, and is defined as $x^\bullet = \{y \mid (x,y) \in F\}$. The map $\langle\cdot,\cdot\rangle$ associates each input arc $(p,t) \in F$ to a pattern $\langle p,t\rangle \in \mathrm{Pat}_\tau$ and each output arc $(t,p) \in F$ to an n-ary query $\langle t,p\rangle \in \mathcal{Q}_\tau$ where $n = |{}^\bullet t|$ is the number of input places of t with a given enumeration on this set of places. We furthermore require that ${}^\bullet t_{in} = \emptyset$, $t_{in}^\bullet = \{p_{in}\}$,${}^\bullet p_{in} = \{t_{in}\}$, ${}^\bullet t_{out} = \{p_{out}\}$, $t_{out}^\bullet = \emptyset$, $p_{out}^\bullet = \{t_{out}\}$, and $\langle p_{out}, t_{out}\rangle = P_{tt}$ (the trivial true pattern matched by any document). Any transition such that ${}^\bullet t \cap P_{DB} \neq \emptyset$ has also input places in $P \setminus P_{DB}$ ensuring that a transition acts on the database only in the context of the processing of a particular transaction. Finally t_{in} is the unique transition with an empty preset, t_{out} is the unique transition with an empty postset, and any place in $P \setminus P_{DB}$ has non-empty preset and postset.

Figure 8 shows an example of Structured Data Net. We adopt the following graphical convention. Input and output transitions are represented by black rectangles. Standard transitions by white rectangles. Places from $P \setminus P_{DB}$ are represented by plain circles, and places from P_{DB} by dashed circles. We assume in this example that all queries are simple and all but $Q_{in} = \langle t_{in}, p_{in}\rangle$ are deterministic. Then input transition t_{in} creates non deterministically a new transaction by putting a token (D, id) in place p_{in} containing a document D (e.g. a form) together with a new identifier id. According to the shape of the token but also to the data contained in place $Data_1$ transitions t_5 and t_1 may be enabled. For instance t_1 may correspond to the nominal behaviour to handle a document while t_5 is used when the document is incomplete or ill-formed. In the latter case the document is immediately transferred to the output place p_{out}. In the former case the treatment is split by t_1 into two threads (concurrent actions t_2 and t_3) and the respective results are aggregated by transition t_4. Then the output transition t_{out} can withdraw a terminated transaction from the system as soon as place p_{out} contains a document, as $\langle p_{out}, t_{out}\rangle = P_{tt}$.

Definition 3.3. (Behaviour of StDNs)
A *token* $T = (D, id) \in \mathrm{Tok}_\tau$ is made of a document $D \in \mathrm{Doc}_\tau$ and a positive integer $id \in \mathbb{N}$. A *marking* $M : P \to \mathcal{M}_f(\mathrm{Tok}_\tau)$ assigns a finite multiset of tokens to each place such that for all $(D, id) \in M(p)$ one has $id = 0$ if and only if $p \in P_{DB}$. Transition $t \neq t_{in}$ is *enabled* in marking M and *firing* transition t in marking M leads to marking M', denoted as $M[t\rangle M'$, when

1. $\exists id \in \mathbb{N}, \forall p \in {}^\bullet t,\ \exists T_p = (D_p, id_p) \in M(p) \quad s.t. \quad D_p \models \langle p,t\rangle$, and $p \notin P_{DB} \Rightarrow id_p = id$,

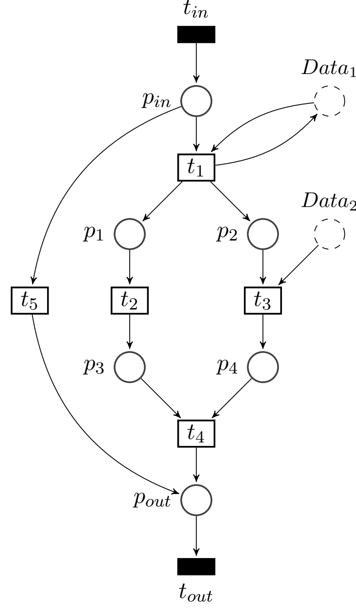


Figure 8. General shape of a Structured Data Net

2. $\forall p \in t^\bullet,\ \exists X_p \in \langle t, p\rangle \left((D_p)_{p \in {}^\bullet t} \right)$, where D_p is the document identified in place p in 1.

3. Let id, D_p and X_p be respectively defined from 1. and 2., then:

   - $\forall p \in t^\bullet \quad M'(p) = M''(p) \uplus \{(D, id_p) \mid D \in X_p\}$ where $id_p = id$ if $p \notin P_{DB}$ and $id_p = 0$ if $p \in P_{DB}$, and
   - $\forall p \notin t^\bullet \quad M'(p) = M''(p)$;

   where M'' is the marking given by:

$$M''(p) = \begin{cases} M(p) \text{ if } p \notin {}^\bullet t \\ M(p) \setminus \{(D_p, id_p)\} \text{ if } p \in {}^\bullet t \end{cases}$$

The behaviour of transition t_{in} is similar except that since it has no input place it is always enabled and no identifier results from the enabling condition. Given a sequence of markings $M_0[t_0\rangle M_1 \ldots M_n$ for a StDN, a *fresh identifier* is an integer $n > 0$ such that for every $i \in 0..n$, and every token (D, id) of M_i, $id \neq n$. To initiate a new case from any marking, t_{in} uses query $\langle t_{in}, p_{in}\rangle$ to create a new document D, attaches it a new fresh identifier id, and adds token (D, id) to input place p_{in}.

When conditions 1 in Definition 3.3 is met we say that transition t is *enabled* in marking M, denoted $M[t\rangle$. Note that the firing relation $M[t\rangle M'$ is non-deterministic due to the fact that first, one may find several token sets that satisfy the patterns associated with the input places of t, and second, the queries associated with the output places may also be non-deterministic. Marking M' is *reachable* from marking M when there exists a sequence of transition firings leading from M to M'. We denote by $\mathcal{R}(M)$ the

set of markings that are reachable from M. Let us comment on the requirement that tokens in a place of P_{DB} carry identifier 0. Identifier 0 is reserved for documents representing records in a local database of the represented system. These documents can be used during transition firings: this represents read or consumption of some recorded value. Conversely, new data can be appended to a database by creation of a token $(D, 0)$ in some place of P_{DB}. Last, when a transition consumes tokens from several places of $P \setminus P_{DB}$, these tokens represent distinct pieces of data for the same transaction, and hence have to carry the same identifier.

3.2. Undecidability

The main motivation for using formal notations and semantics is to derive automated tools to reason on models of systems. For transactional systems, one may want to check that a request with correct type is always processed in a finite amount of time, regardless of current data. Another issue is to guarantee that, for instance, a payment on an online store is always followed by the sending of the purchased item to the buyer. Last, one may want to check some simple business rules on transactions, confidentiality of some data, etc. In most cases, the properties to be checked do not deal with global states of the modeled system, but rather consider the status of one particular transaction in a limited environment. Most of these questions boil down to checking whether there exists a marking in which a place p contains some document D, regardless of the contents of other places. Therefore, the properties of interest for StDNs are closer to coverability properties than to reachability properties. In this section we formalize and address decidability of reachability, coverability, termination (whether all transactions terminate), and soundness (the question of whether all transactions terminate without leaving pending threads in the system). We can formalize reachability, coverability, termination and soundness as follows for an StDN with respect to a given initial marking M_0. We will assume w.l.o.g. that M_0 contains no transaction: $\forall p \in P \setminus P_{DB}, M_0(p) = \emptyset$. Indeed, transition t_{in} can non-deterministically introduce a new transaction at any time.

Reachability: For a given marking M, is M reachable from the initial marking: $M \in \mathcal{R}(M_0)$?

Coverability: For a given marking M, assuming an ordering $\leq$ on markings, is M smaller than some reachable marking: $\exists M' \in \mathcal{R}(M_0)\ \ s.t.\ \ M \leq M'$?

Termination of a transaction: Given a marking M such that a new transaction has just been created ($M(p_{in})$ contains a token (D, id) which is the only token with identifier id in M), *can one reach* a marking M' such that $M'(p_{out})$ contains a token (D', id)? This question can be reduced to a coverability question. Furthermore, it is a *weak* form of termination since it only guarantees that a transaction *may* terminate, and this termination may rely on information produced by other transactions. A stronger termination property expresses that for *every* marking reachable from such a marking M one can reach a marking M' such that $M'(p_{out})$ contains a token (D', id). Thus a system is terminating if *any* transaction can reach completion no matter how its execution started. Verifying strong termination requires forward accessibility techniques which are beyond the scope of this paper and which we intend to develop in extensions to this work. In this paper we thus restrict to *weak termination*.

Soundness: Termination is not sufficient. We also want to guarantee that in any marking, M', witnessing termination of a transaction, i.e., such that $M'(p_{out})$ contains a token (D', id), then no other token with identifier id –i.e., that takes part in the same transation – remains in M'. We say, in that case, that the transaction cleanly terminates: when it reaches the output place, no active thread associated

with it remains elsewhere in places of the net. The property that *every* transaction cleanly terminates is called *soundness*. We call *weak-soundness* the restriction of this property when one only assumes weak-termination: any transaction *may* terminate, and if it terminates it ends in a clean way, i.e., without leaving documents related to the terminated transaction in the system. This problem can be addressed using coverability techniques: one should be able to cover a marking witnessing termination, but markings with several tokens carrying a similar identifier in p_{out} and in other places must not be coverable.

Obviously, weak termination and weak soundness bring less guarantees than strong termination and soundness, as a net in which some transactions can deadlock can be weakly terminating or weakly sound. However, a StDN that does not meet weak soundness is ill-formed: it can terminate in configurations that leave unexploited documents in the system. Standard definitions of soundness are sometimes called ”proper termination” [30]. Another definition was proposed in [31] for workflow nets: it requires in addition to proper termination that no transition of the workflow is dead. Since we restrict ourselves to weak termination, and thus also to weak soundness, we omit in the following the adjective “weak” and assume that the weak forms of termination and soundness are intended unless explicitely stated. All questions above are undecidable if no restriction is imposed on the nature of documents or queries. We show in theorem 3.4 below that StDN can encode Turing machines, and that termination or coverability questions can be used to model the halting problem of a Turing Machine, which is known to be undecidable. In Section 3.3, we consider a class of StDNs that are effective well-structured transition systems, a property that guarantees the decision of coverability.

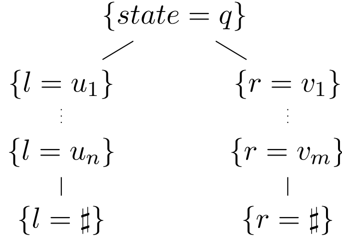
Theorem 3.4. (Undecidability)
Reachability, coverability, termination and soundness are undecidable problems for StDNs.

Proof:
We encode a Turing machine into an StDN. We recall that a Turing machine is made of an infinite bi-directional tape divided in both directions into an infinite number of consecutive cells and a finite state device that can read and write the cell being examined by a read/write head and that can also move that head along the tape in both direction. A cell contains a 0 or a 1, initially every cell has the default value 0. More precisely a Turing machine consists of a finite set of states Q with some initial state q_0 and a finite set of instructions of the form $[q, x, \omega, q']$ where q and q' are states, $x \in \{0, 1\}$ is the possible value of the cell, and $\omega \in \{0, 1, L, R\}$ is an operation that corresponds respectively to writing 0 or 1 in the current cell or moving the r/w-head to the left or to the right. A configuration is a triple $(q, u, v) \in Q \times \{0, 1\}^\omega \times \{0, 1\}^\omega$ made of a state $q \in Q$ and two infinite words coding respectively the contents of the left part of the tape, read from right-to-left, and the contents of the right part of the tape, read from left-to-right. The r/w-head is positioned on the first cell of the right-part of the tape. The transitions of the Turing machine are given as follows:

1. Writing a value $y \in \{0, 1\}$ on the current cell: $(q, u, x \cdot v) \xrightarrow{[q,x,y,q']} (q', u, y \cdot v)$.

2. Right move: $(q, u, x \cdot v) \xrightarrow{[q,x,R,q']} (q', x \cdot u, v)$.

3. Left move: $(q, y \cdot u, x \cdot v) \xrightarrow{[q,x,L,q']} (q', u, y \cdot x \cdot v)$.



A reachable configuration (q, u, v) contains only a finite number of non-null elements therefore one can encode a configuration with a tree as shown next where $\forall i > n, u_i = 0$ and $\forall i > m, v_i = 0$. We let $[q, u, v]$ denote this tree (even though the representation is not unique). In terms of this representation the moves of the Turing machine can be simulated with the rules:

1. Writing a value $y \in \{0, 1\}$ on the current cell: $[q, u, x \cdot v] \xrightarrow{[q,x,y,q']} [q', u, y \cdot v]$.
2. $[q, u, x \cdot v] \xrightarrow{[q,x,R,q']} [q', x \cdot u, v]$ and $[q, u, \sharp] \xrightarrow{[q,0,R,q']} [q', 0 \cdot u, \sharp]$.
3. $[q, y \cdot u, x \cdot v] \xrightarrow{[q,x,L,q']} [q', u, y \cdot x \cdot v]$ and $[q, \sharp, x \cdot v] \xrightarrow{[q,x,L,q']} [q', \sharp, 0 \cdot x \cdot v]$.

Each of these rules can straightforwardly be represented by a transition r with ${}^\bullet r = r^\bullet = \{p_{in}\}$ where pattern $\langle p_{in}, r\rangle$ describes those configurations that enable rule r and query $\langle r, p_{in}\rangle$ describes the effect of r on such a configuration. Pattern $\langle t_{in}, p_{in}\rangle = \{[q_0, \sharp, \sharp]\}$ produces the initial configuration. We complete the description of the StDN by adding one transition $halt_{q,x}$ from p_{in} to place p_{out} for each pair of state q and symbol x for which there is no move of the machine of the form $(q, x, -, -)$ where pattern $\langle p_{in}, halt_{q,x}\rangle$ tests that the state is q and the symbol read is x and query $\langle halt_{q,x}, p_{out}\rangle$ witnesses the halting of the Turing machine by creating a specific token, e.g. which document part is the empty configuration $[q_0, \sharp, \sharp]$, in the output place p_{out}. Then, a Turing Machine halts if and only if its StDN counterpart can reach a marking with one token $T = (D, id)$, with $D = [q_0, \sharp, \sharp]$ in p_{out}. Recalling that the halting problem for Turing Machines is undecidable, we immediately get that reachability of a marking of a StDN is undecidable. For this StDN reachability or coverability of the final marking with one token in p_{out} are equivalent to (weak) termination or (weak) soundness thus all these properties are undecidable. □

3.3. WQO Structured Data Nets

The result of Theorem 3.4 is not surprising, as reachability or coverability are usually undecidable for Petri nets with extended tokens like colored Petri nets. However, one may note several important issues from the encoding of a Turing machine. First, deterministic queries are sufficient for this encoding. Second, three distinct tags and finite domains of values are sufficient to encode a configuration of a Turing machine. An immediate question is whether one can rely on the structure of the data and on simple restrictions to obtain decidability results. A first obvious useful restriction is to bound the depth of documents manipulated by the system. This restriction is reasonable, as it is unlikely that documents depths grow arbitrarily during their lifetime in a system. Similarly, databases of arbitrary sizes can be represented as arbitrarily large sets of bounded depth documents in places of P_{DB}. Now, bounding the depth of documents is not sufficient to obtain decidabiliy results: One can indeed use configurations of Turing Machines as values for tags in bounded depth documents. We hence need restrictions on the domains of valuations too. By Proposition 2.6 a set of bounded depth documents is a wqo when the domains of the data fields attached to tree nodes are wqos. We can use this ordering to define the interesting class of *well quasi ordered StDNs*, and then see how ordering on documents extend to their markings.

Definition 3.5. (WQO StDN)
An StDN is well quasi ordered (is a *wqo StDN* for short), when

i) the domains of values used by document data fields are well quasi ordered (finite sets, integers, vectors of integers,...), with effective comparison (one can can effectively decide if $x \leq_\sigma y$), and

ii) there exists a bound on the depth of all documents appearing in $\mathcal{R}(M_0)$.

Let us comment on the restrictions in Definition 3.5. Assuming wqo values in documents still allows to work with infinite domains like integers. However, this restriction forbids to attach structured data such as queues of unbounded sizes to nodes. Within the context of transactional systems, this is not a severe limitation. Note also that checking whether $\mathcal{R}(M_0)$ contains only bounded depth documents can be reduced to the question of whether a Turing Machine has a bounded number of configurations, and is hence undecidable. However, this property is frequently met, and is not a severe limitation either: Most of transactional systems can be seen as protocols working with a finite number of data fields or using finite forms, in which a finite number of entries needs to be filled. Hence, applying a query usually does not increase too much the depth of a document. Even when large sets of facts are recorded in a XML document (think for instance of a patient record), these facts are not placed at growing depths (i.e., along a single path of the tree representing this document), but are rather independent subdocuments (i.e, they are placed on different branches of the tree), possibly with a creation date if these records need to be ordered. Hence, large documents often have big width but are still of bounded depth. One shall also note that the depth of standard structured documents is usually very low: The structure helps decomposing an entry into data fields, i.e. decomposing a concept into sub-concepts (a person is described as someone with a first name and last name) and it is recognized [32] that 99% of XML documents have depth smaller than 8, and that the average depth of XML documents is 4. Note also that the depth restriction does not mean finiteness of manipulated data: Trees of arbitrary width still comply with this restriction, and data values attached to nodes need not be chosen from finite domains. This allows for instance for the manipulation of XML documents containing arbitrary numbers of records. Still, as shown at the end of this section, considering well quasi ordered StDNs is not enough to obtain decidability.

Let us define the ordering relation on the set of markings induced by the ordering on documents, and thus ultimately by the ordering on the data values appearing in these documents. The powerset of an ordered or quasi ordered set $(A, \leq)$ is equipped with the quasi order $\leq$ where $X \leq Y$ when an injective map $h : X \to Y$ exists such that $\forall x \in X \quad x \leq h(x)$. For multisets $X, Y \in \mathcal{M}(A)$ we similarly let $X \leq Y \iff [\![X]\!] \leq [\![Y]\!]$ where $[\![X]\!] = \{(x, i) \mid x \in X \ \wedge\ 1 \leq i \leq X(x)\}$ denotes the set of occurrences of X. Markings are compared component-wise up to an injective renaming of the identifiers of transactions. More precisely, we let $M_1 \leq M_2$ when there exists an injective map $h : \mathbb{N} \to \mathbb{N}$ such that $h(0) = 0$, and for every place p and every $i \in \mathbb{N}$ one has $\pi_i(M_1(p)) \leq \pi_{h(i)}(M_2(p))$ where $\pi_i(M(p)) = \{D \mid (D, i) \in M(p)\}$ denotes the multiset of documents in $M(p)$ with identifier i. As the comparison between two markings is performed up to a renaming of transactions, the exact identifier of a token does not matter. The only concern is whether two tokens with the same (respectively with different) identifier(s) are mapped to tokens with the same (resp. with different) identifier(s). Hence, we can equivalently consider markings as partitions of a multiset[1] of pairs from $P \times \mathrm{Doc}_{\tau,\leq n}$. As a partition of a set X is a set of subsets of X, any quasi order on X extends (using twice the powerset extension) to a quasi order on the set of partitions of X. With this representation $M_1 \leq M_2$ when the two partitions are

[1] By partition of a multiset X we mean a partition of the set $[\![X]\!]$ of occurrences of X.

comparable for the extension to partitions of the ordering $\leq$ on $P \times \mathrm{Doc}_{\tau,\leq n}$ given by $(p, D) \leq (p', D')$ when $p = p'$ and $D \leq D'$.

Proposition 3.6. The set of markings over bounded depth documents whose data have well quasi ordered domains is a wqo.

Proof:
From proposition 2.6, we know that $(\mathrm{Doc}_{\tau,\leq n}, \leq)$ is a wqo. Since the set of places is finite, the ordering relation on $P \times \mathrm{Doc}_{\tau,\leq n}$ is also a wqo. Last, the product of two wqos forms a wqo [33], and we have seen that extending the ordering to multisets and then to partitions also yields a wqo. Hence, the ordering on markings over documents of bounded depth is a wqo. $\square$

An immediate followup to well quasi orderedness is to set restrictions to obtain well-structured transition systems (WSTS) and reuse existing results to check coverability. An n-ary query Q is said to be *monotonous* when $(\forall i \in \{1, \ldots, n\} \quad D_i \leq D'_i) \implies Q(D_1, \ldots, D_n) \leq Q(D'_1, \ldots, D'_n)$.

Proposition 3.7. Let $\mathcal{N}$ be a wqo StDN with monotonous patterns and queries, and let M_1, M'_1, M_2 be markings of $\mathcal{N}$, then $M_1[t\rangle M'_1$ and $M_1 \leq M_2$ implies $\exists M'_2, M_2[t\rangle M'_2 \ \wedge \ M'_1 \leq M'_2$.

Proof:
According to Definition 3.3 we distinguish the initial transition t_{in}, which is responsible for the creation of new identifiers, from the other transitions.
If $t = t_{in}$**:** The transition t_{in} is not guarded, and results in a non-deterministic creation of new documents $D_1, \ldots, D_k$ with a fresh identity id in place p_{in}, namely $M'_1(p_{in}) = M_1(p_{in}) \uplus \{(D_1, id)) \cup \cdots \cup (D_k, id)\}$, and $M'_1(p) = M_1(p)$ for every $p \neq p_{in}$. Then, one can find a fresh integer id' that is not used in M_2 so that $M_2[t_{in}\rangle M'_2$ where $M'_2(p_{in}) = M_2(p_{in}) \uplus \{(D_1, id')) \cup \cdots \cup (D_k, id')\}$, and $M'_2(p) = M_2(p)$ for every $p \neq p_{in}$. As $M_1 \leq M_2$, there exists an injective map h such that for every place p and every $x \in Dom(h)$, $\pi_x(M_1(p)) \leq \pi_{h(x)}(M_2(p))$. We extend this map by letting $h(id) = id'$ to get $\pi_{id}(M'_1(p)) \leq \pi_{id'}(M'_2(p))$ and thus $M'_1 \leq M'_2$.
General case ($t \in T \setminus \{t_{in}\}$**):** This transition is enabled when all the patterns $P_1 = \langle p_1, t\rangle, \ldots, P_k = \langle p_1, t\rangle$ attached to flows from places $p_1, \ldots, p_k$ in ${}^\bullet t$ to t are satisfied by some documents $D_1, \ldots, D_k$, with the same identifier id for documents located in places ${}^\bullet t \setminus P_{DB}$, and with identifier 0 for documents from ${}^\bullet t \cap P_{DB}$. Upon firing, t consumes $D_1, \ldots D_k$ from ${}^\bullet t$, and outputs a set of newly created documents $D'_1, \ldots D'_{k'}$ with identifier id in places of $t^\bullet \setminus P_{DB}$, and with identifier 0 in places of $t^\bullet \cap P_{DB}$ where $\left\{D'_1, \ldots D'_{k'}\right\} = \cup_{p \in t^\bullet} X_p$ for some $X_p \in \langle t, p\rangle(D_1, \ldots, D_k)$. As $M_1 \leq M_2$, there exists an injective mapping $h : \mathbb{N} \to \mathbb{N}$ from identifiers in M_1 to identifiers in M_2, such that for every identifier x and every place p, $\pi_x(M_1(p)) \leq \pi_{h(x)}(M_2(p))$. This also yields, for each identifier x and each place p a map $\varphi_{p,x} : \pi_x(M_1(p)) \to \pi_{h(x)}(M_2(p))$, such that for every document D_i in $M_1(p)$ we have $D_i \leq \varphi_{p,x} D_i$. Let us denote by $\varphi = \bigcup \varphi_{p,x}$ the union of all these maps for $p \in P$, and x an identifier used in M_1.

Since guards are monotonous and $D_i \leq \varphi(D_i)$, one has $\varphi(D_i) \models P_i$. From the monotony of patterns, and as h preserves equality of identifiers, we have that t is also enabled from M_2. From the monotony of queries we deduce that for every place $p \in {}^\bullet t$, there exists $X'_p \in \langle t, p\rangle(\varphi(D_1), \ldots, \varphi(D_k))$ with $X_p \leq X'_p$. Thus, there exists a marking M'_2 such that $M_2[t\rangle M'_2$ with

$$\begin{cases} M_2'(p) = M_2(p) \text{ if } p \text{ is not in } {}^\bullet t, \\ M_2'(p) = M_2(p) \setminus \{(\varphi(D_p), 0)\} \uplus (X_p' \times \{0\}), \text{ if } p \in P_{DB} \cap {}^\bullet t, \text{ and} \\ M_2'(p) = M_2(p) \setminus \{(\varphi(D_p), id')\} \uplus (X_p' \times \{id'\}), \text{ if } p \in (P \setminus P_{DB}) \cap {}^\bullet t, \text{ where } id' = h(id) \end{cases}$$

Let us now prove that $M_1' \leq M_2'$. We can design a set of injective maps $\varphi'_{p,x} : \pi_x(M_1'(p)) \rightarrow \pi_{h(x)}(M_2'(p))$ witnessing $M_1' \leq M_2'$. For every (D_i, x) that is not consumed by firing of t, we define $\varphi'_{p,x}(D_i) = \varphi_{p,x}(D_i)$, as the documents that were not consumed remain unchanged and hence comparable in both markings. Then, for each newly created document D_i' in X_p, as $X_p \leq X_p'$, we necessarily have a token (D_i', id) in $M_1'(p)$, a document D_j' in X_p' such that $D_i' \leq D_j'$, and a token of the form $(D_j', h(id))$ in $M_2'(p)$ (this includes the case when $p \in P_{DB}$ and hence $id = 0$). Hence we can set $\varphi'_{p,x}(D_i') = D_j'$, with the property that $D \leq \varphi'_{p,x}(D)$ for every newly created document D_i' with identifier x in place p. Hence, we can keep the same map h, and yet obtain the property that for every identifier x and every place p, $\pi_x(M_1'(p)) \leq \pi_{h(x)}(M_2'(p))$, which witnesses $M_1' \leq M_2'$. □

By Proposition 3.7 wqo StDNs with monotonous patterns and queries are WSTS. The standard backward algorithm of section 2.1 to decide coverability can be adapted to wqo StDNs as follows: For a given set of markings X, we let $pre(X) = \{M \mid \exists t \in T, M' \in X, M[t\rangle M'\}$. The coverability algorithm starts from set $X_0 = \{M\}$, that is a basis for all markings greater than M. Then, it iteratively computes a basis $X_{i+1} = X_i \cup predB(\uparrow X_i)$ for the sets of markings from which a marking in $\uparrow M$ can be reached in a finite number of steps. The algorithm stops when a fixed-point is reached, or as soon as a marking $M' \in X_i$ is found such that $M' \leq M_0$, indicating that there exists a sequence of transitions from M_0 to a marking greater than M. Now, wqo StDNs with monotonous patterns and queries are not necessarily *effective* WSTS. Indeed, one needs an effective algorithm to build the pred-basis $predB(\uparrow X_i)$. We will say that a StDN is *backward-effective* if, given a marking M, one can effectively compute a basis for $predB(\uparrow M)$.

Corollary 3.8. (Coverability)
Coverability is decidable for backward-effective wqo-StDN with monotonous patterns and queries.

Proof:
A consequence of Proposition 3.7 is that wqo-StDN with monotonous patterns and queries are WSTS. Backward-effectiveness allows to compute a basis for $predB(\uparrow M)$, and hence $predB(\uparrow X_i)$, which guarantees effectiveness of each step of the backward coverability algorithm. It remains to show that the comparison among markings is effective. For any pair of documents $D_1, D_2 \in \mathrm{Doc}_\tau$, one can effectively check for the existence of a mapping from D_1 to D_2, and compare the values of paired data fields, as we have assumed that the domains of these data-fields are effective wqos. Then finding an identity preserving mapping among contents of places (finite multisets) is also effective. □

In the rest of the document, we will refer to backward-effective wqo-StDN with monotonous patterns and queries as *effective* StDNs. Backward effectiveness means that from a basis for an upward closed set of markings X one can effectively build a finite representation of the set of predecessors of $\uparrow X$. This means being able to find the minimal data needed to fire some transition and reach $\uparrow X$. This property is easily met if the effect of a transition on a place is to aggregate finite amount of data collected from its input places (for instance the sum of positive integers collected in forms), or to append a new branch to a document (in this case, the consumed documents are subtrees of documents appearing $\uparrow X$).

Let us now show that this result on coverability allows to prove more properties, and in particular to address termination, soundness, and coverability for sets of initial markings described symbolically. More precisely, we want to guarantee a property such as coverability or termination for any initial document satisfying a given pattern P, i.e., any document in $\hat{P}$. From a practical point of view, this question resumes to considering that the set of documents that can be generated by query $\langle t_{in}, p_{in} \rangle$ is exactly $\hat{P}$. Note that in general, the set of documents generated by $\langle t_{in}, p_{in} \rangle$ needs not satisfy a single pattern P. It may be the case that this set of initial cases is not upward closed (for instance, a query can generate documents which nodes carry only odd integer values). However, a significant number of real systems take as inputs forms without complex contraints on their data fields, which are upward closed set of documents. Hence, considering upward closed sets of initial cases matching a pattern makes sense. Theorem 3.9 addresses coverability, termination and soundness when sets of initial documents are described by monotonous patterns. The coverability problem for the set of initial cases induced by P can be rephrased as follows: assuming $\mathrm{Im}(\langle t_{in}, p_{in} \rangle) = \hat{P}$, and given a marking M to cover, is there a marking M' greater than M in $\mathcal{R}(M_0)$ for every initial marking M_0 such that $M_0(p_{in}) \in \hat{P}$? The termination (resp. soundness) problems extends similarly to the set of markings containing a document that satisfies P.

Theorem 3.9. (Weak) termination and (weak) soundness are decidable for effective STDNs. Coverability for symbolic set of initial cases defined by a monotonous pattern are decidable for effective STDNs.

Proof:
Let M_0 be a marking such that $M_0(p_{in})$ contains a token (D_0, id) produced by transition t_{in} (for some identifier id). Let $D_\perp$ denote the least document (reduced to an untagged root). The **termination** of case (D_0, id) is equivalent to the coverability of the marking M_{end} such that $M_{end}(p_{out}) = (D_\perp, id)$ (and where all other places are empty) by some marking reachable from M_0. We recall that order on markings is based on the partition induced by the identifier and thus the exact value of the identifier does not matter. Decidability of **soundness** also stems from decidability of coverability. An StDN is sound if it terminates and whenever place p_{out} contains a token, one cannot find another place containing a token with the same identifier, i.e. for each place $p \in P \setminus \{p_{out}\}$ the marking M_p with token $(D_\perp, id)$ in both places p_{out} and p and with no other tokens in other places is not coverable from the initial marking.

Coverability, termination and soundness have solutions for a single given initial marking, i.e. for a particular chosen case. We would like to consider whether a given marking M can be covered when starting from *every* possible input to the system. We suppose that the set of results output by query $\langle t_{in}, p_{in} \rangle$ is the **symbolic set of documents** from $\mathrm{Doc}_{\tau, \leq n}$ that satisfy a particular monotonous pattern P.

Then, as P is monotonous, we can reuse the construction of Proposition 2.7 to build a basis $Bsat(P)$ for all documents satisfying P. Noticing that $\mathcal{R}(M) \leq \mathcal{R}(M')$ when $M \leq M'$ for wqo StDNs with monotonous queries and patterns, coverability can be verified for all cases initiated by $\langle t_{in}, p_{in} \rangle$ if it can be proved for all initial markings M_0 such that $M_0(p_{in})$ belongs to $BSat(P) \times \{id\}$ for any arbitrary identifier id (again with a chosen database contents if needed). Let us denote by $BM_{0,P}$ this set of markings. Note that it is sufficient to compute once the fixed-point X_m returned by the set-saturation algorithm that builds a basis for $Pre^*(\uparrow M)$ and then compare this set with elements in $BM_{0,P}$. Coverability for all cases satisfying pattern P is guaranteed iff for every initial marking M_0 in $BM_{0,P}$, there exists $x \in X_m$ such that $x \leq M_0$. This result obviously extends to termination and soundness. □

The above decidability results do not extend to reachability:

Theorem 3.10. (Undecidability of reachability)
Reachability is undecidable, even for effective StDNs.

Proof:
It is known that reachability is undecidable for reset Petri nets [10], and a StDN can easily simulate a reset Petri net. In section 5, we even prove that effective StDNs can simulate the more general class of Reset Post-G nets. To model a Reset Petri net $\mathcal{N}$, we create a StDN $\mathcal{N}'$, with only three places: p_{in} and p_{out} and a database place p_{one}, that will contain a single token. $\mathcal{N}'$ also contains usual transitions t_{in}, t_{out}, and one transition per transition of $\mathcal{N}$, the reset net to simulate. We also use a transition t_{one} from p_{in}, with ${}^\bullet t = \{p_{in}, p_{one}\}$ and $\langle p_{in}, t\rangle = \langle p_{one}, t\rangle = P_{tt}$. This transition will be used once to select one particular instance of a document representing marking M_0 of the reset net. At any time, the contents of place p_{out} contains at most one document, and encodes the current marking of $\mathcal{N}$. A marking is encoded as a document D with a root node and a child labeled p for every place p of the reset Petri net $\mathcal{N}$, such that $\nu(p) = n$ indicates that place p of the reset net contains n tokens (see Section 5, Figure 20). Represented this way, the set of tokens in some place p can be reset to 0, incremented or decremented by monotonous queries. Enabledness of a transition t of $\mathcal{N}$ can be encoded by a pattern P_t that tests the existence of a desired number of token in some place p, i.e. they are trees composed of a root, and one child node with constraint of the form $p_i \geq n_i$ per place in ${}^\bullet t$. We hence set $\langle p_{out}, t\rangle = P_t$. For every transition t, a monotonous query $\langle t, p_{out}\rangle$ can be used to increment or decrement the value of a particular node tagged by p, encoding consumption or creation of tokens. Such a query can even set the value of tag p to 0, simulating a reset arc. These queries are monotonous, and transitions using this kind of queries are also backward effective. Last, $\langle t_{in}, p_{in}\rangle$ is the query that produces document D_0 representing marking M_0 of the reset nets. Hence $\mathcal{N}'$ is an effective net simulating $\mathcal{N}$ (only firings of t_{in} and t_{out} are added to runs of $\mathcal{N}$). The general shape of the net is depicted Section 5, Figure 21. Undecidability of the reachability problem for reset nets [10] concludes the proof. □

This result should not be seen as a severe limitation: Many properties of transactional systems are not expressed in terms of global states and do not need reachability. Moreover reachability is undecidable for many extensions of Petri nets. We show in Section 5 similar encodings to simulate other extensions of Petri nets such as Reset Post-G nets or Nested nets. The remaining questions to conclude with wqo-StDN is whether this class is decidable, and whether well quasi orderedness of a net suffices to obtain decidable properties. Unfortunately, both questions have negative answers:

Proposition 3.11. Well quasi orderedness of an StDN is undecidable. Coverability, reachability and termination problems are undecidable for wqo StDNs.

Proof:
We design a wqo StDN that encodes a two counters machine. A two counters machine is given as a pair of counters C_1, C_2 holding non-negative integers and a finite list of instructions $l_1, \dots l_n$ each of which, except the last one, is of one of the following forms: *i)* $l_i : inc(C_\ell)$ meaning that we increment counter C_ℓ and then go to the following instruction, *ii)* $l_j :$ **if** $(C_\ell = 0)$, l_k **else** $dec(C_\ell)$, $l_{k'}$ indicating that if counter C_ℓ is null we must proceed to instruction l_k otherwise we decrement this counter and go to instruction $l_{k'}$. The machine halts when it reaches the last instruction $l_n : Halt$. A configuration of

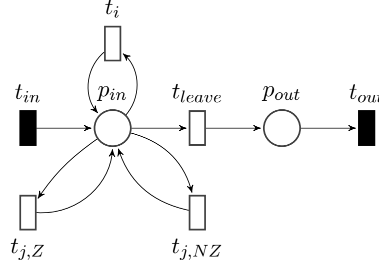


Figure 9. Encoding a counter Machine with wqo Structured Data Nets

a counter machine is given by the value of its counters, and the current instruction line. The machine usually starts at instruction 0, with counters set to 0. It is well-known that one cannot decide if a counter machine halts. For any counter machine, we can define an StDN (represented in Figure 9) that encodes the moves of the machine.

First, we can encode a counter machine configuration as a document with three nodes: a root, and its left and right children. The root is tagged by an instruction number from $l_1, \ldots, l_n$, the left and right children are tagged by c_1 and c_2 respectively with values given by non-negative integers. The corresponding documents are of bounded depth with values from wqo domains. For each instruction of the form l_i : $inc(C_\ell)$, we design a transition t_i with ${}^\bullet t_i = t_i^\bullet = p_{in}$ such that $P_i = \langle p, t_i \rangle$ is the pattern reduced to a root whose tag has value l_i and $Q_i = \langle t_i, p \rangle$ is the query that transforms a document into a document with root l_{i+1}, and such that the value attached to the node with tag c_ℓ is incremented by one, and the other one is left unmodified. For each instruction of the form l_j : **if** $(C_\ell = 0)$, l_k **else** $dec(C_\ell)$, $l_{k'}$ we design two transitions $t_{j,Z}$ and $t_{j,NZ}$ such that $P_{j,Z} = \langle p_{in}, t_{j,Z} \rangle$ is a (non monotonous) pattern testing if the root of a document is labeled by l_j, and the value of node with tag c_ℓ is zero, $Q_{j,Z} = \langle t_{j,Z}, p_{in} \rangle$ is the query that transforms a document into a document with root l_k, and such that the values attached to child nodes remain unchanged, $P_{j,NZ} = \langle p_{in}, t_{j,NZ} \rangle$ is the pattern testing if the root of a document is labeled by l_j, and the value of node with tag c_ℓ is greater than zero, $Q_{j,NZ} = \langle t_{j,NZ}, p_{in} \rangle$ is the query that transforms a document into a document with root $l_{k'}$, and such that the value attached to node with tag c_ℓ is decremented by one and the value attached to the other child node remains unchanged. The initial configuration of the counter machine is created by query $Q_{in} = \langle t_{in}, p_{in} \rangle$ that produces a document with root labeled l_0 and two children nodes tagged respectively by c_1, c_2 with values 0. We set M_0 as an initial marking in which all places are empty. Transition t_{leave} moves the token from place p to p_{out} if the root tag has value l_n, i.e. the machine halted. Clearly, the counter machine terminates iff one can reach a configuration in which p_{out} is not empty. Thus one cannot decide termination, and similarly the reachability or coverability (of the marking with just one token in p_{out}).

Let us now prove that one can not decide whether a net is wqo. One can add a transition t_{nobnd} to the above net such that ${}^\bullet t_{nobnd} = t_{nobnd}^\bullet = p_{out}$, $\langle p_{out}, t_{nobnd} \rangle = tt$, and $\langle t_{nobnd}, p_{out} \rangle$ is a query that increases the depth of a document by 1, by inserting a children with some tag a between the root and its first child (hence creating successive incomparable documents). Then the counter machine terminates iff the corresponding StDN is not wqo. □

Even though well quasi orderedness of a net is undecidable, acceptable restrictions ensure this property. Queries that non-deterministically insert new integers, rationals or strings in existing forms are backward effective and monotonous, and do not increase the depth of documents beyond a certain limit.

In many systems, queries are used to extract data from a data-set, i.e., a list of records that can usually be represented by bounded depth documents. The result of data extraction is a list of records that can be again assembled as a bounded depth document. Other queries compute new values from data-sets (sums, means, etc.) and insert the results in a new document (a "form") of bounded depth and size. Such queries are often backward effective and monotonous. So, one can restrict to queries that produce only documents of bounded depth, which values domains are finite sets or wqo sets (such as integers) without harming too much the expressiveness of the model.

Let us conclude this section by considering *safety* properties of effective StDNs. Verifying whether all runs of the system avoid markings where a given place p contains a given document D_{bad} representing an undesirable property boils down to reachability and is thus undecidable. If however we replace the document to be avoided by an upward closed set X_{bad} of documents the question becomes decidable since it reduces to coverability: "Is $M_0 \geq M'$ for some $M' \in Pre^*(\uparrow M_{bad})$?", where $M_{bad} = \{M \mid M(p) \in Basis(X_{bad}) \times \{id\}\}$ for an arbitrary identifier id, should receive a negative answer.

A similar and more interesting question can be asked for *sets of initial cases* satisfying a monotonous pattern P. Namely: "Does there exists a marking M_0 such that $M_0(p_{in})$ contains a token of the form (D, id) where document D satisfies pattern P, and a run starting from M_0 and ending in a marking M such that $M(p)$ contains a token (x_{bad}, id) with $x_{bad} \in X_{bad}$?". The question can be formally written as: "Does $\uparrow (X_{P,id}) \cap Pre^*(\uparrow M_{bad}) = \emptyset$?" where $X_{P,id}$ is a set of initial markings such that $M_0(p_{in}) = (x, id)$ with $x \in Bsat(P)$. For effective StDNs, one can compute a basis B_{bad} for $Pre^*(\uparrow M_{bad})$ and $X_{P,id}$ has also a basis $B_0 = \{M \mid M(p_{in}) = (x, id) \wedge x \in X_{P,id} \wedge M(p) = \emptyset \text{ for } p \neq p_{in}\}$).

However, difficulties arise due to the fact that the safety question does not ask for inclusion of an upward closed sets of markings into another one, which could be solved by comparing their basis, but rather checks emptiness of their intersection. A marking in the intersection of $\uparrow B_0$ and $\uparrow B_{bad}$ is a marking in which p_{in} contains a token of the form (D, id), where $D \geq D'$ for some $D' \in Bsat(P)$ and $D \geq B'$ for some $B' \in B_{bad}$. As satisfaction of a pattern and ordering of documents imply root preserving embeddings, such a marking exists if and only if one can find a pair of documents (D', B') in $Bsat(P) \times B_{bad}$ whose roots allow the existence of such mappings. More precisely, let ν, ν' denote the respective valuations of $root_{D'}$ and $root_{B'}$, and let $CT_{D',B'}$ be the set of tags shared by both roots. There exists a common document in $\uparrow D' \cap \uparrow B'$ if and only if $\uparrow \nu(\sigma) \cap \uparrow \nu'(\sigma) \neq \emptyset$ for every $\sigma \in CT_{D',B'}$. On the example of Figure 10, one can see two documents B_1, B_2 whose roots carry a common tag b with integer valuation. The document B_3 in this figure is greater than both B_1 and B_2 and hence belongs to $\uparrow B_1 \cap \uparrow B_2$. We have made little assumptions on valuations, and even when every $\sigma \in CT_{D',B'}$ has valuation in some well quasi ordered set, nothing guarantees that $\uparrow \nu(\sigma) \cap \uparrow \nu'(\sigma) \neq \emptyset$ can be effectively checked. If all valuations for roots of documents appearing in $\uparrow (X_{P,id})$ and $Pre^*(\uparrow M_{bad})$ take values in wqos that are *intersection effective* (i.e, such that for every x one can effectively compute a basis for $\uparrow x \cap \uparrow x'$), then our safety problem has a solution. Integers with their natural ordering, labeled graphs with finite sets of labels ordered by embedding relations are examples of intersection effective wqos.

4. Case Study

Let us illustrate the main features and expressive power of Structured Data Nets on an example: a travel agency. We use this case study not only to highlight interesting features of the model, but also possible extensions, and discuss whether these extensions could hinder decidable classes. The workflow of a case

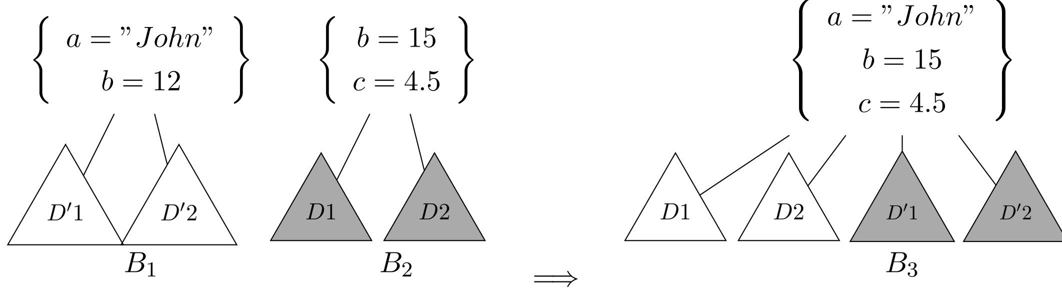


Figure 10. Merging two documents with compatible roots

in our travel agency example is the following: a client enters the system with a document indicating the origin of her travel, the city she wants to visit, and a maximum budget for this travel. Then the travel agency seeks concurrently for hotel and flight offers, and builds a set of proposals that fit the customer's budget. Once an offer has been selected, a payment is performed: the customer enters her bank details, and a clearance is asked to the bank. Upon positive answer from the bank, the travel documents are printed, reservations are done, and the case in considered as completed. On the contrary, if the bank does not accept payment, the travel is canceled.

We model more precisely this case study with a StDN. This StDN manipulates several types of documents. Let us start with three documents shown in Figure 11. The first one is the client's request. It is a document of depth 1 containing information on both origin and destination, the desired departure and return dates, and the maximal amount the client wishes to pay for the travel —including hotel and flight prices. As a simplification, we assume that both origin and destination may be connected by air travel. The second document is a flight request: It contains the same data fields as the client's request, but with document root labeled by tag $FlightRequest$. This type of document also contains an additional data field, i.e, a new node carrying tag $Status$, which valuation belongs to the set $\{\mathrm{searching}, \mathrm{booking}\}$, indicating respectively that a search is currently carried out for this request, and that the search was stopped to start the booking phase. The third document is a flight offer from a company. It contains origin and destination cities, a company name, a price, and several departure and return dates which combinations are proposed for the same price.

For convenience, we cut the whole structured data net depicting the workflow of the travel agency in several parts, and explain the contents of each piece separately. Places with identical names in different pictures represent the same place in the whole network. We start with the part of the StDN, depicted in Fig. 12, that processes the arrival of a client's request and initializes the workflow. Transition t_{in} non-deterministically creates a token (D, id) in place p_{in} where D is a document representing a travel request, as in Fig. 11. Hence, query $\langle t_{in}, p_{in}\rangle$ is a simple non-deterministic query. Transition *split* separates a client's request into flight and hotel requests. It consumes a Client demand, and applies as soon as place p_{in} contains such document. Hence we set $\langle p_{in}, \mathrm{split}\rangle = P_{tt}$. Firing transition *split* creates two documents: A flight request in place *FlightRequest* using query $\langle\mathrm{split}, \mathrm{FlightRequest}\rangle$, and an hotel request in place *HotelRequest* using query $\langle\mathrm{split}, \mathrm{HotelRequest}\rangle$. Both budgets attached to these documents must be lower than the overall travel budget. For simplicity we consider here that queries deterministically apply a 60% rate to the overall budget to compute a bound for flight and hotel prices. When a client request with identifier id is consumed by transition *split*, two documents with the

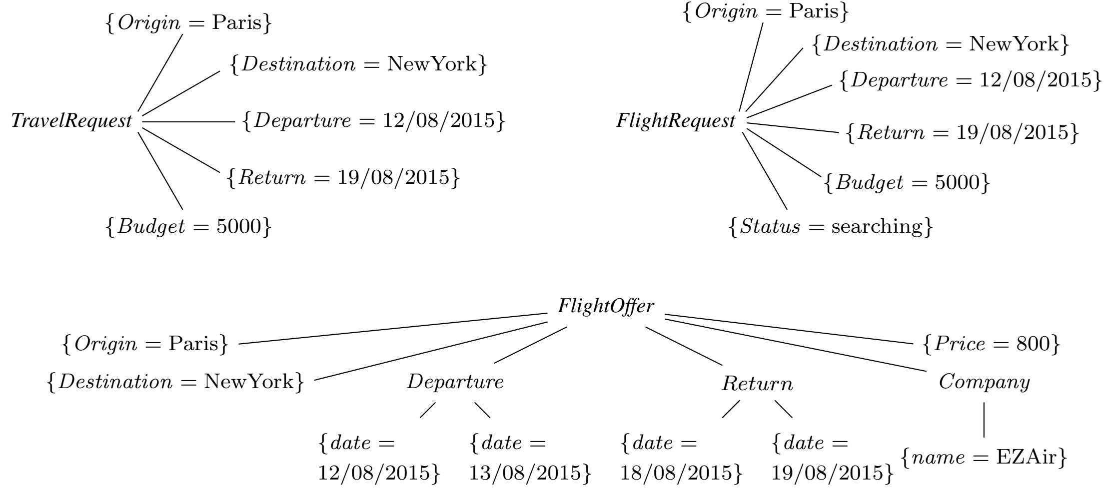


Figure 11. Documents used by the travel agency: Travel request, Flight request, and Flight offer —where a node labelled "*label*" is an abbreviation for a node with valuation $\{label = *\}$ where *label* is a tag with value domain $\{*\}$.

same identifier id are created. They initiate concurrent search processes.

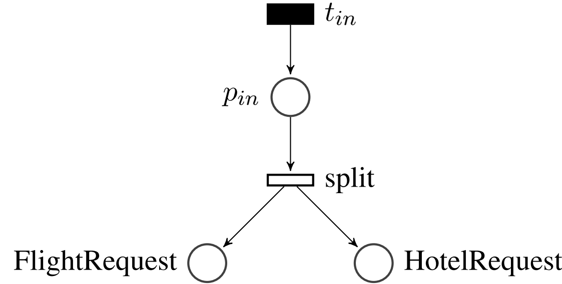


Figure 12. Splitting a travel request into searches for hotels and flights.

Workflow given in Figure 13-a) tells how to search flights that comply with client's wishes, initially introduced in place *FlightRequest*, regarding desired destination, dates and budget. Various offers of several companies are stored in place *FlightOffers*. Such an offer takes the form of a document as shown in Fig. 11. Since this place is a database place one does not have to consider identifiers attached to documents contained in this place in order to fire transitions. At initial stage of reservation, a flight request is a document (together with an identifier) whose *status* tag has value "searching". Transition *SearchFlight* is fired if there exists a flight request with status equal to "searching" in place *FlightRequest*, and a flight offer document in place *FlightOffers*. We hence set $\langle \text{FlightOffers}, \text{SearchFlight} \rangle = P_{tt}$, and $\langle \text{FlightRequest}, \text{SearchFlight} \rangle = P_{\text{Searching}}$, where $P_{\text{Searching}}$ is the pattern depicted in Figure 13-b). Transition *SearchFlight* simply reads a flight request and a flight offer, but does not consume them. Hence, the query attached to output flow from transition *SearchFlight* to place *FlightRequest* simply re-

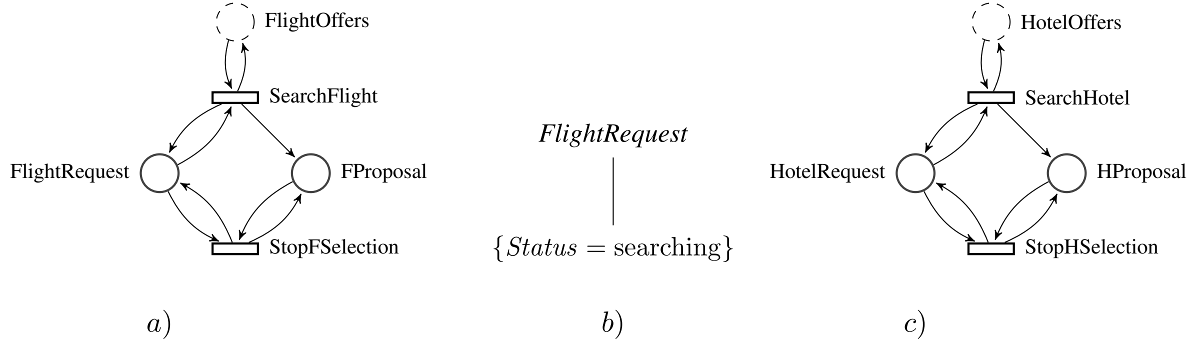


Figure 13. Searching for flights (Fig. a), pattern $P_{\text{Searching}}$ (Fig. b), and searching for Hotels (Fig. c).

turns the consumed flight request document to place *FlightRequest*. Similarly, the query attached to flow from transition *SearchFlight* to place *FlightOffer* returns the selected flight offer document unchanged to place *FlightOffers*. This can be easily defined by setting $\langle \text{SearchFlight}, \text{FlightRequest} \rangle (D_1, D_2) = D_1$ and $\langle \text{SearchFlight}, \text{FlightOffers} \rangle (D_1, D_2) = D_2$. The query attached to the output flow from transition *SearchFlight* to place *Fproposal* returns an empty set if the flight offer and requests are not compatible (inadequate date, destination or budget), and a new document that merges the request and offer information tagged with the same identifier as the flight request otherwise. This way, one can non-deterministically fill the contents of place *FProposal* with flight offers that comply with the client's wishes. One can notice that query $\langle \text{SearchFlight}, \text{Fproposal} \rangle$ is not monotonous, because applying this query with expensive flight offers may result in empty answers while using this query with a cheap offer produces an offer.

Note also that transition *SearchFlight* can fire an arbitrary number of times for the same client request. The search process can stop after an arbitrary number of steps as soon as at least one offer has been found. This is modeled by transition *StopSelection* that can fire if there exists at least one flight proposal document in place *FProposal* and a flight request that is still in the searching phase in place *FlightRequest*, with identical identifier.

We hence define $\langle \text{FProposal}, \text{StopSelection} \rangle = P_{tt}$ and $\langle \text{FlightRequest}, \text{StopSelection} \rangle = P_{\text{Searching}}$. One just needs to check existence of some flight proposal, and not consume it, so query $\langle \text{StopSelection}, \text{FProposal} \rangle$ simply copies the document selected from place *FProposal*. The status of the selected flight request changes after firing transition *StopSelection*, which is modeled by a query $\langle \text{StopSelection}, \text{FlightRequest} \rangle$ that changes the valuation of node with tag *Status* to "Booking" in the selected flight request. Hence, after firing *StopSelection*, we have the following properties:

- pattern $P_{\text{Searching}}$ does not hold for the modified flight request,
- transitions *SearchFlight* and *StopSelection* can no longer be fired with the modified request as input,
- place *FProposal* contains at least one valid offer carrying the same identifier as the modified flight request with status "booking".

Selection of hotels, depicted on Figure 13-c), follows the same principle, and fills place *HProposal*

starting from a place *HotelRequest* containing hotel requests, and a database place *HProposal* containing hotel offers.

Remark 4.1. Let us now comment on this implementation of database search.

- One can first notice that, since flight (resp. hotel) offers are not consumed by transition *SearchFlight* (resp. *SearchHotel*), the same offer can be used twice, and hence several identical proposals can be produced in place *FProposal* (resp. *HProposal*) for the same request. Second, the selection process for a particular flight (resp. hotel) request stops after an arbitrary number of firings of transition *SearchFlight* (resp. *SearchHotel*). Overall, this gives very little control on the obtained contents of proposal places, that can contain a single document, or conversely several documents but with redundancy. In some sense these operations are *lazy* counterparts of classical database selection processes. At the end of this section, we will describe a system which copies the whole content of a place in order to perform an exhaustive selection on this content.

- In the three parts of net represented so far, some places are read, and the consumed token is put back to its origin place without modification. This calls for a harmless extension of our model with read-arcs, which would avoid the definition of useless queries and output flows.

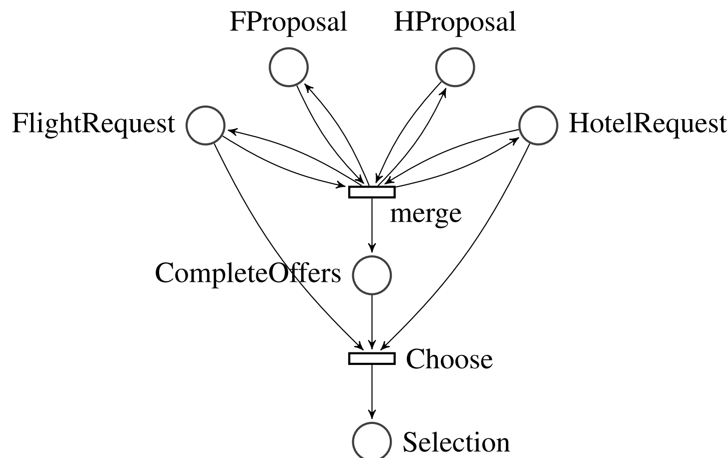


Figure 14. Selection of a flight and hotel offer fitting with a given budget.

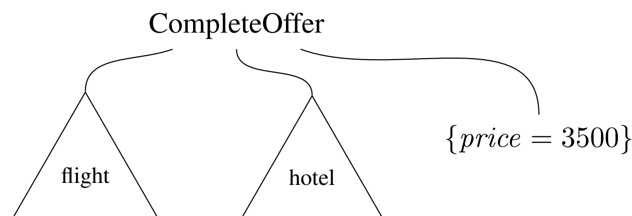


Figure 15. A complete offer document.

The next step depicted in Fig. 14 consists in selecting a pair of flight and hotel offers that fits the client's budget. This is implemented in the following way: Flight and hotels are merged non-deterministically to build complete offers, that fit the client's budget. The overall price of the offer is the sum of the flight offer's price and of the hotel offer's price. This information in merged into a single *complete offer* document such as the one depicted in Fig. 15. In this schema, triangles are just copies of the respective offers that were selected. The only new information added is the global price of the offer just computed. The workflow depicted in Fig. 14 behaves as follows: Transition *merge* fires when there exists an hotel request with status "booking", an hotel proposal, a flight request with status "booking", and a flight proposal that all carry the same identifier. Query $\langle \mathrm{merge}, \mathrm{CompleteOffers} \rangle$ computes the sum of flight and hotel proposals prices, and returns an empty set if the overall price is greater than $\frac{10}{6}$ times the flight request budget (recall that this budget is 60% of the overall budget). Otherwise, it returns a *complete offer* document containing the flight and hotel offers as children of a root node tagged *CompleteOffer*, plus an additonal node with tag *price* whose value is the overall computed price. Queries $\langle \mathrm{merge}, \mathrm{FlightRequest} \rangle$, $\langle \mathrm{merge}, \mathrm{FlightOffer} \rangle$, $\langle \mathrm{merge}, \mathrm{HotelRequest} \rangle$ $\langle \mathrm{merge}, \mathrm{HotelOffer} \rangle$ return the document used from their respective places to its original place. Transition *choose* consumes a complete offer, flight request and hotel request with identical identifier. This is a way to select one of the offers proposed to a client, and to stop all computations for these requests (no flight request/hotel request with the selected id appears in the net after firing *choose*. We hence only define $\langle \mathrm{FlightRequest}, \mathrm{Choose} \rangle = \langle \mathrm{HotelRequest}, \mathrm{Choose} \rangle$, $\langle \mathrm{CompleteOffer}, \mathrm{Choose} \rangle = P_{tt}$. Query $\langle \mathrm{Choose}, \mathrm{Selection} \rangle$ copies the complete offer selected in place *CompleteOffer* to place *Selection*.

The mechanisms used to build a selection in Fig. 13 and 14 allow to simulate pattern matching on the contents of several places and on valuations of several nodes. In the basic definition of StDN (Def. 3.2), transitions are guarded by *local* patterns attached to each input arc, and apply to one token per place. The constraints on data values used in patterns are also localized to a single node of a document (item 2 of Def. 2.3). However, following the schema of Fig. 13 and 14, one can first non-deterministically merge pairs of documents from distinct places (i.e. build a tree with a new root and the merged documents as subtrees) and then check truth value of some pattern on the merged contents to simulate a *global* constraint on the contents of several places. Similarly, one can copy tags and values from several nodes of several documents to a newly created single node carrying tags and values from several other nodes, and then use patterns to check locally a constraint on this node. Merging is a monotonous query, and is backward effective. For this reason, extending the pattern matching mechanisms to handle contents of several places, and valuations from several nodes does not harm decidability results demonstrated in this paper.

The last part of the workflow is rather standard, and consists in payment of the selected offer. Transition *BankInfo* selects a complete offer and adds bank information (client's name, bank name, account and credit card number) to the offer. As bank information can be added to a document as soon as an offer has been selected, we set $\langle \mathrm{Selection}, \mathrm{BankInfo} \rangle = P_{tt}$. Adding bank information is designed as a non-deterministic query $\langle \mathrm{BankInfo}, \text{Selection+Bank} \rangle$, that simply adds data fields to an existing document, and moves it to place *Selection+Bank*. We then model the bank's decision as a transition *BankDecision*. Again, decision of a bank regarding payment of a travel can occur as soon as bank information was entered, so, $\langle \text{Selection+Bank}, \mathrm{BankDecision} \rangle = P_{tt}$. Query $\langle \mathrm{BankDecision}, \mathrm{Decision} \rangle$ nondeterministically adds a node to the document with tag *Granted* and boolean valuation $\nu(Granted)$. Once bank's decision is obtained, the client's choices can be booked and printed if the payment was granted, or canceled otherwise. We set patterns $\langle \mathrm{Decision}, \mathrm{PrintTravel} \rangle = P_{granted=\mathrm{true}}$ (resp. $P_{granted=\mathrm{false}}$) where

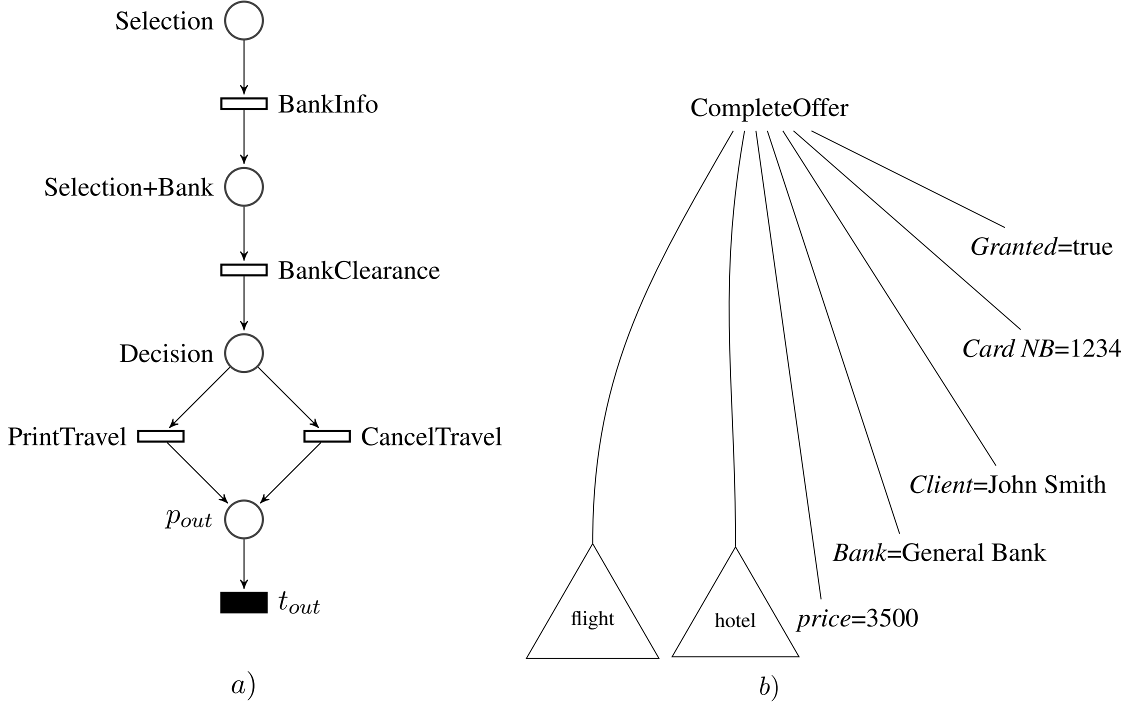


Figure 16. The workflow for payment and termination of a case $a)$, and a possible final document $b)$

$P_{granted=\text{true}}$ (resp. $P_{granted=\text{false}}$) is a pattern that checks existence of a node with tag *Granted* with value true (resp. false). Queries $\langle \text{PrintTravel}, p_{out}\rangle$ and $\langle \text{CancelTravel}, p_{out}\rangle$ simply copy a selected document from place *Decision* to place p_{out}. Hence, after bank's decision either transition *PrintTravel* or *CancelTravel* fires and moves the document to place p_{out}, from which the transaction can be terminated. This part of the workflow and a possible final document are described in Figure 16.

Note that our design of this case study is of bounded depth: All produced documents have depth at most 3. It is however not monotonous. Let us again consider the part of the workflow depicted in Fig. 13, and in particular the query $Q_{SF,FP} = \langle \text{SearchFlight}, \text{FProposal}\rangle$ that filters out flight offers that are too expensive. When checking that a flight offer is compatible with the request, one needs to compare the request's budget with the price attached to a flight offer. Assuming the price attached to an offer document D_O is compatible with the expected budget, a new document is created in place *Fproposal* (that collects eligible flights only). If we build a document D'_O with the same destination and dates as D_O, but with a higher proposed price, the flight offer is rejected: Query $Q_{S,FP}$ applied to D'_O returns an empty set, and thus adds no document to place *FProposal*. Assuming prices are given by integers one has $D_O \leq D'_O$, but $Q_{SF,FP}(D_O) \not\leq Q_{SF,FP}(D'_O)$. A way to overcome this problem is to assume an upper bound for prices, and use inverse order to compare them ($price_1 \leq price_2$ iff value $price_1$ is bigger or equal to value $price_2$). This way, prices are still wqos, and query $Q_{SF,FP}$ becomes monotonous. However, this *reverted order* trick cannot be used anymore if a pattern imposes lower bounds on prices somewhere else in the net.

Note also that case termination is not guaranteed in the model of travel agency. Indeed, if no offer fits the maximal budget of a client with respect to the hotel or flight, then a flight request document (respectively a hotel request document) can remain forever in place *FlightRequest* (resp. in place *HotelRequest*).

The system as it is designed is not sound either: A selection of offers that are compatible with a client's budget may remain in place *CompleteOffers* after one particular offer has been selected and paid.

Structured Data nets are Turing complete (as proved in Theorem 3.4), so one can encode complex programs and data structures using StDNs. In the rest of this section, we enumerate some syntactical features which are missing in our model that can ease the design of a specification. Furthermore, we show a few non-native operations that can be simulated with StDNs.

We have already discussed the possibility to add read arcs and global patterns without changing the expressive power or decidability results of the model. Other missing features are needed to allow standard operations in database systems. The first one is a *selection* operation over the contents of places. This would allow to design filtering mechanisms, and implement operations such as "select all flight offers which price is lower than 10 000 euros". The second database feature we consider is a *join* operation, that would allow to implement operations of the form : "merge all hotel and flight offer documents that agree on the destination". It is also very common in databases to select a record with the minimal/maximal value with respect to some criterion (like the value of some datafield). Another frequent operation is to select the p best records with respect to some criterion. As highlighted in Remark 4.1, selection operations can be simulated by StDNs, but in a lazy way that uses non-deterministic transitions, and gives no guarantee on termination of the selection, nor on exhaustivity of the returned solution, nor on uniqueness of returned documents.

In our case study, selection of (some) flight offers meeting the budget constraint set in a flight request is modeled by a piece of StDN in Fig. 13). It can easily be adapted to filter offers with respect to other criteria, such as an upper bound on flight price, or to select flight offers that have the same destination as some request. However, this simplistic selection process is implemented in a naive and somehow unsatisfactory way. We now show how to perform an exhaustive selection on a place by first copying the contents of this place, and then applying some filtering operation to this copy.

A solution to apply an operation to all tokens in a place is to count the number of documents appearing in that place. Figure 17 is a gadget StDN that copies the contents of some place P_1 into another place P_2 as soon as some transition *start* is fired. The copy is achieved after firing of transition *end*, that is enabled only when the whole copy operation has been performed. In this gadget, we use an additional intermediate place P_1' to save documents consumed from P_1 during the copy, and three places C_1, C_1', and C_2, that respectively count the number of documents in places P_1, P_1', and P_2. More precisely, place C_1 (resp. C_1', C_2) contains a single document composed of a root node with a tag c_1 (resp. c_2, c_1') whose value $\nu(c_i)$ (resp. $\nu(c_2), \nu(c_1')$) is the exact number of documents in places P_1 (resp. P_1', P_2). Slightly abusing our notations, we will write $\nu(C_i)$ to refer to the valuation attached to the single node of the single document contained in place C_i. To ease presentation, we adopt the following convention: we attach a pattern name to input flow arcs of the figure when the pattern is not trivial. Input flow arcs without label are attached the trivial pattern P_{tt}. Similarly, we decorate output flow arcs with query names. We also represent the value of each counter place C_i by an integer located in place C_i. We denote $C_{i=0}$ the pattern that checks that a document has a root node with tag c_i and value $\nu(c_i)$ equal to 0, by $C_{i>0}$ the pattern that checks that a document has a root node with tag c_i and value $\nu(c_i)$ greater than 0. These patterns are attached to an input flow (C_i, t) from place C_i to some transition t. We denote by C_{i++} (resp. C_{i--}) the query that increments (resp. decrements) the value of tag c_i in the counter document of place C_i. These queries are attached to some output flow arc (t, C_i). Last, we denote by id_{P_i} the query that copies the document with provenance P_i used as input argument by all queries attached to some transition. For instance, on the drawing of Fig. 17, we have $\langle C_1, \text{copy1} \rangle = C_{1>0}$, $\langle P_1, \text{copy1} \rangle = P_{tt}$,

$\langle \text{copy1}, C_1 \rangle = C_{1--}$, and $\nu(C_1') = 0$.

The net starts in a marking M with one token in place p_{init}, several tokens $(D_1, id_1), \dots (D_k, id_k)$ in place P_1, $\nu(C_1) = k$, $M(P_2) = M(P_1') = \emptyset$, $\nu(C_2) = \nu(C_1') = 0$. The principle of the copy gadget is as follows : transition $copy1$ can be fired when place p_{cs} contains a token (hence after firing of transition $start$), place P_1 contains a document, place C_1 contains a token with strictly positive valuation, and place C_1' contains a token. The effect of firing transition $copy1$ is to consume a document from P_1, copy it into P_2 and P_1', decrement $\nu(C_1)$, and increment $\nu(C_1')$ and $\nu(C_2)$. Transition $copy1$ is the only transition that can fire until $\nu(C_1) = 0$. As soon as $\nu(C_1) = 0$ (and hence $\nu(C_1') = \nu(C_2') = k$ and $M(P_1') = M(P_2) = \{(D_1, id_1) \dots, (D_k, id_k)\}$), transition $copy_1$ can not fire anymore, and the only transition that is firable is transition $back$ that moves a token from place p_{cs} to place p_{bs}. After firing of transition $back$, transition $back1$ can fire until $\nu(C_1') = 0$. The effect of firing $back1$ is to move a document from place P_1' to place P_1, decrement $\nu(C_1')$ and increment $\nu(C_1)$. As soon as $\nu(C_1') = 0$, transition $back1$ can not fire, and the only firable transition is end, which terminates the copy operation. At the end of the copy, we have $\nu(C_1) = \nu(C_2) = k$, $\nu(C_1') = 0$, $M(P_1) = M(P_2) = \{(D_1, id_1) \dots, (D_k, id_k)\}$ and $M(P_1') = \emptyset$.

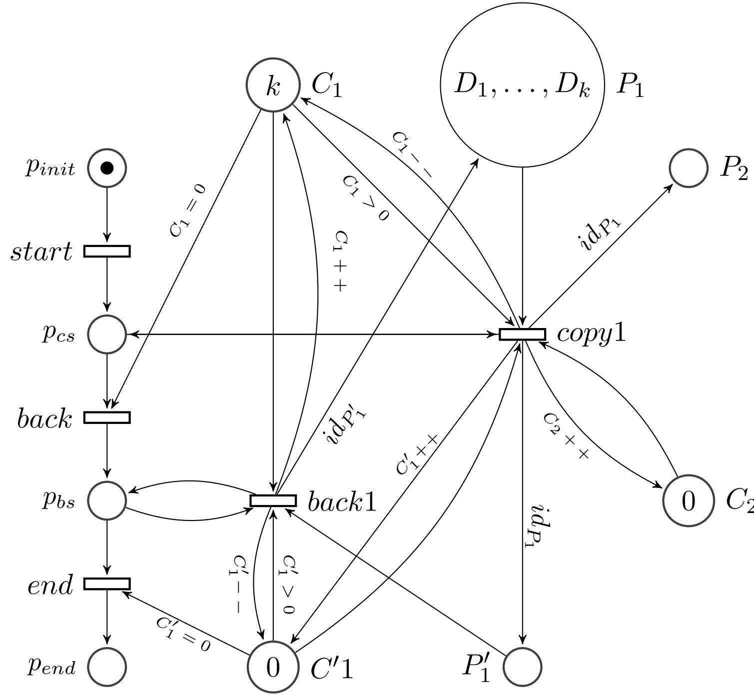


Figure 17. A copy gadget with StDNs

After a copy, one can easily apply a chosen operation to all copies of documents in P_2 (selection with respect to a pattern, merge of all documents with the contents of one place, ...). This can be done via an iterative process that consumes documents of P_2 and allows some transition to fire until $\nu(C_2) = 0$. As an illustration, let us get back to the travel agency example. Let us suppose that all flight offers have been copied to some place P_2, and that the number of flight offers in P_2 is maintained in some counter place C_2. Then, we can select one particular flight request, compare it to all tokens in P_2, and keep only

offers that are compatible with the flight request's budget. This can be encoded for instance an in the net of Fig. 18. It behaves as follows: a particular flight request is selected from the place containing all flight requests using transition *SelectOne*. This selection can occur only once, as transition *SelectOne* consumes a token from place p_{one}. Query $\langle \mathrm{SelectOne}, \mathrm{OneRequest} \rangle$ simply moves the selected flight request to place *OneRequest*. As soon as a particular flight request has been selected, and as long as a copied document remains in place P_2, transition *SearchFlight* can fire. As in the model of Fig. 13, $\langle \mathrm{SearchFlight}, \mathrm{FProposals} \rangle$ creates a new proposal in place *FProposals* if the selected offer's price is compatible with the selected request's budget (it adds an empty set of documents to the place otherwise). The major differences with the selection process described in Fig. 13 is that the iterative selection process is applied to a single flight request at a time, that *every* flight offer in place P_2 is consumed, and that the transition also updates the contents of the counter C_2 associated with place P_2. When place P_2 is empty, place *FProposals* contains one proposal for *each* offer that was compatible with the budget of the selected flight request.

Note that the copy gadget of Fig. 17 and the selection net of Fig. 18 are not monotonous, as they use patterns that check that a counter is equal to zero. The copy gadget of Fig. 17 can be easily simplified to transfer all documents from a place P_1 to a place P_2, to transfer only documents with chosen characteristics, to reset the whole contents of place P_1, etc. One can also design a gadget that erases all tokens carrying the same identifier as some designated transaction. Such reset mechanisms can be used for instance to enforce soundness of a system. Of course, implementing such gadget nets that address *all documents* of a place requires implementing counting mechanisms. A similar result can be achieved by extending patterns definition to allow negative patterns that would allow to check that a place *does not contain* some kind of document. Flows carrying such negative patterns would however play the same role as inhibitor arcs for Petri nets. Of course, using such features results in a loss of monotony for patterns, and hence in undecidability of coverability, termination and boundedness. One can design similar gadgets to select the maximal element from a set of records, the best p records out of n proposals, etc. These gadgets are very low-level, and of course, one shall not expect designers of a workflow system to specify at such a detail level. Hence, to be practical, StDNs should be equipped with a higher-level syntax which semantics would be the underlying low-level StDN obtained by composition and specialisation of gadgets. In the next section, we will show relations of StDNs with other extensions of Petri nets, and in particular with reset and nested nets. Interestingly, we will show that wqo StDNs suffice to simulate these nets, and hence that unbounded depth of documents, or non-monotonous patterns and queries are not mandatory to encode interesting extensions of Petri nets.

5. Comparison with high-level nets

Structured Data nets is an extension of standard Petri nets with data and queries. Other high-level Petri net extensions have been proposed for the design and orchestration of complex workflows. In this section, we compare StDNs with existing Petri net variants.

Several net variants have been proposed to model workflows, such as for instance workflow nets [4]. These nets model transactions with forks, joins, and concurrent subtasks. They contain two particular places representing input and output of the designed system. A data-centric variant of workflow nets called Jackson nets have been proposed in [5]. These nets are workflow nets which can be described by so-called Jackson types, i.e. expressions that define the structure of documents. Informally, a Jackson net

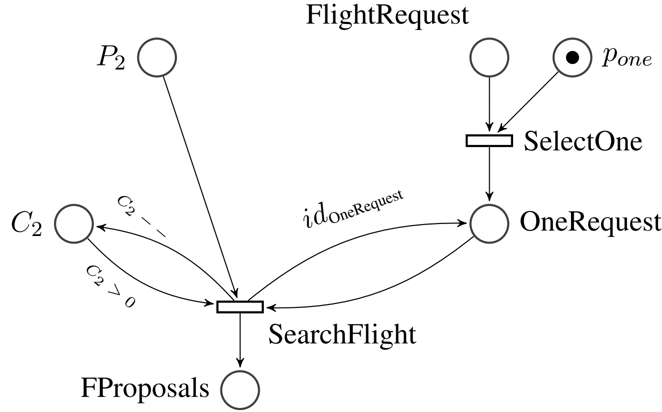


Figure 18. Selection of all flights compatible with a request after a copy.

describes in which order to fill the contents of a structured document with iterated or optional parts. This approach is data-centric in the sense that the way data is organized influences the workflow. However, contrarily to StDNs, in a Jackson nets the actual value of data in a document does not influence the workflow. Jackson and workflow nets have another drawback: They describe the execution of a single case, and can not be used to handle several transactions running concurrently. On the contrary, Structured Data nets are designed to allow some form of isolation of distinct transactions through identifiers. Though this is not the strict isolation in the sense of ACID properties, this differentiation among concurrent cases is an interesting property for transactional web-based systems).

Structured data nets are not the first extension of Petri nets handling tokens with complex types. Since StDNs are, in their full generality, Turing Powerful, they can simulate most of known Petri net extensions, as for instance Colored Petri nets [6]. Colored Petri nets can also be considered as Petri nets with data. However, it is well known that colors give a huge expressive power to nets, and can be used to encode arithmetic operations. It is hard to find a reasonable syntactic subclass of colored nets that is amenable to verification of simple properties. Yet, we are convinced that our model can certainly be defined using complex coloring mechanisms. Our nets are close in spirit to $PrT-$Nets [7], that modify structured data via manipulations that are guarded by First Order predicates. However, StDNs use guarding mechanisms that can not be encoded in FO. XML nets [8] is another variant of nets that manipulates and transforms structured data. Places of an XML net carry data and are constrained by DTDs to guarantee well-formedness of documents, while transitions perform data manipulations described with the (Xmanila) query language. Structured Data nets are close in spirit to XML nets, but keep XML transformations as abstract as possible. Translating XML nets to StDNs would mainly require to use Xmanila as query language, and to ensure that queries do not produce documents violating any DTD. Note also that this work on StDNs emphasizes on semantics, decidability and formal properties of the model. Consequently, the model was designed to allow for identifiable subclasses amenable to verification, which was not the major concern of former works on XML Nets.

In the rest of this section, we compare the expressive power of effective StDNs with several Petri net extensions. As we are interested by properties close to coverability, it is interesting to compare effective StDNs with classes of nets for which coverability is decidable. Such nets include Generalized

Self Modifying nets[10], nested nets [9], or Petri nets with token carrying data [1]. In this last extension, data is not really transformed through the workflow, but is mainly used to adapt the structure of flows of a affine nets at runtime. Let us first compare StDNs with the classes of Generalized Self Modifying nets depicted in [10] (GSM net for short).

Definition 5.1. (GSM Nets)
A *Generalized SM net* is a tuple $\mathcal{N} = (P, T, F, M_0)$ where P is a set of places, T a set of transitions, and M_0 an initial marking, the flow F is a polynomial function of place contents of the form $F(x, y) = \sum_{j \in J} \lambda_j p_{i_j}^{n_j}$, where $(x, y) \in P \times T \cup T \times P$, $\lambda_j, n_j \in \mathbb{N}$, and $p_{i_j} \in P$, and J is a finite set.

The major difference between GSM and Petri nets is that the flow relation in GSM is a polynomial function of place contents of the form $F(x, y) = \sum_{j \in J} \lambda_j p_{i_j}^{n_j}$, where $(x, y) \in P \times T \cup T \times P$, $\lambda_j, n_j \in \mathbb{N}$, and $p_{i_j} \in P$, and J is a finite set. For a marking M and a flow function $F(x, y) = \sum_{j \in J} \lambda_j p_{i_j}^{n_j}$, we denote by $val(F(x, y), M) = \sum_{j \in J} \lambda_j M(p_{i_j})^{n_j}$ the value obtained by replacing every variable p_{i_j} by $M(p_{i_j})$. Informally, a transition t can fire in marking M if for every place p, $M(p)$ is greater than $val(F(p, t), M)$. Similarly, the number of tokens put in a place p when firing transition t from M is $val(F(t, p), M)$. GSM nets can be separated into several subclasses with distinct properties, depending on characteristics of functions attached to their input and output flows. We list some of them, and refer to [10] for more classes and details on classes inclusion.

Definition 5.2. (GSM subclasses)
A GSM net is a

- ***Self Modifying net*** iff functions $F(p, t)$ and $F(t, p)$ use polynomials of degree at most 1.
- ***Post-G net*** iff only post arcs carry functions, and $F(p, t)$ is always an integer for every $p \in P, t \in T$. Output flow functions can be polynomials.
- ***Reset Post-G net*** iff either $F(p, t) = p$ (the whole contents of place p is consumed, this flow is a reset arc) or $F(p, t) \in \mathbb{N}$. Output flow functions can be polynomials.
- ***Transfer Post-G net*** iff it is a Reset postG nets such that if $F(p, t) = p$, then there exists a place p' such that $F(t, p') = p$, i.e. transition t transfers the contents of place p into place p'.
- ***Post-SM net*** iff it is a Self Modifying-net in which $F(p, t)$ is always an integer for every $p \in P, t \in T$. Output flow functions can be polynomials of degree at most 1.
- ***Reset Petri net*** iff it is a Reset Post-G Nets with standard output flow relation ($F(t, p) \in \mathbb{N}$).
- ***Transfer Petri nets*** iff it is a Transfer Post-G Nets that contains only normal arcs or transfer arcs.

As shown on the examples of Figure 19, GSM nets can encode nets with inhibitor arcs, which yields undecidability of termination, boundedness, coverability and reachability properties. However, several classes with decidable properties have been identified [10]. In the above list, Self Modifying nets is the only class with undecidable coverability, and all other mentioned classes are contained in the class of Reset Post-G Nets, for which coverability is decidable.

One can first notice that markings of GSM nets remain standard Petri net markings, that can be encoded very easily with bounded depth and WQO documents. Let $M : P \to \mathbb{N}$ be a marking. We can build a document D_M for this marking that has a root with any tag, and $|P|$ children nodes $n_1, \ldots n_{|P|}$ with tags $p_1, \ldots p_{|P|}$, and valuation $\nu(p_i) = M(p_i)$ attached to each node n_i. Figure 20-b) shows an example of such marking encoding. A similar encoding can be used and remains WQO for any kind of

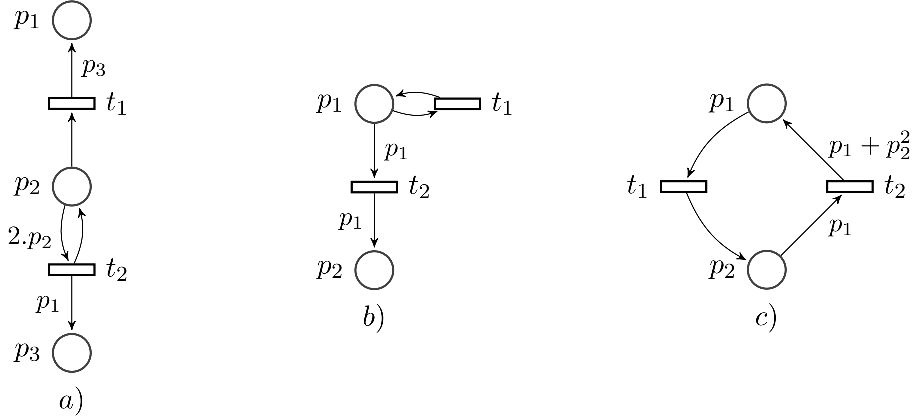


Figure 19. Examples of Generalized Self Modifying nets (borrowed from [10]). Net $a)$ is a self modifying net that contains an inhibitor arc: as soon as place p_2 contains a token, constraint $p_2 = 2.p_2$ can not be satisfied. Net $b)$ is a transfer Petri net : when transition t_2 fires, the contents of place p_1 is moved to place p_2. Net $c)$ is a reset post-G net: the whole contents of place p_1 is consumed when t_2 fires.

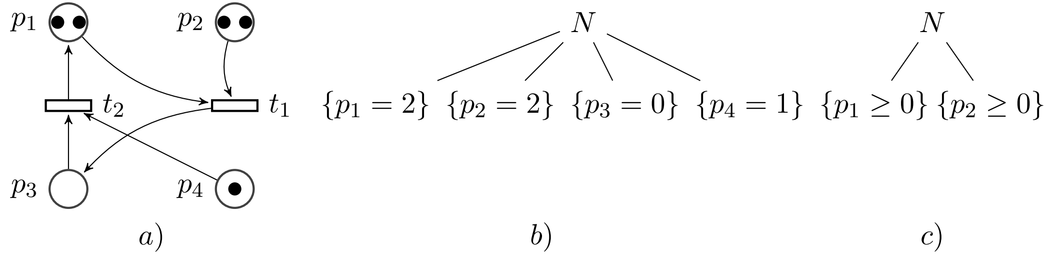


Figure 20. Encoding an integral marking of a net with a bounded document: Figure $a)$ is a Petri net, with marking $m(p_1) = 2, m(p_2) = 2, m(p_3) = 0, m(p_4) = 1$, and Figure $b)$ shows document D_m associated with this marking. Valuations attached to nodes copy the contents of places. Figure $c)$ is the pattern P_{t_1} checking on a marking document that the number of tokens in places of m allows firing of t_1.

net provided the marking associates a multiset of elements from a finite domain to places. The question is now whether firability of a transition t from a marking M is equivalent to satisfaction of some pattern P_t by D_M. Then, simulating the effect of a transition firing can be done by a query Q_t that updates deterministically the contents of D_M.

We will show later that for several interesting classes of extended Petri nets models, marking encoding and appropriate local patterns are sufficient to encode the semantics of a net with effective StDNs. For instance, Figure 20-$c)$ shows an example of pattern which satisfaction on a document D_M is equivalent to enabledness of transition t_1 in the Petri net of Figure 20-$a)$. However, it is already clear that for some classes of GSM nets, checking firability of a transition t needs to compare the contents of a place with polynomials over all place contents. This comparison can not be encoded by a simple pattern P_t, as patterns only check local constraints on nodes, i.e. constraints on individual place contents. We can show that no appropriate encoding with effective StDNs can solve this issue. Indeed, decidability properties of effective StDNs and SM nets suffice to prove the following result:

Proposition 5.3. There exist GSM nets that can not be encoded by effective StDNs.

Proof:
The proof is straightforward: coverability is decidable for effective WQO StDNs, and undecidable for (G)SM-nets [10]. □

A typical example of net that has no effective StDN counterpart is the net of Figure 19-a). This net is in fact a net with inhibitor arc : as soon as place p_2 contains a token, the constraint $m(p) > 2.m(p)$ is violated, and this transition can not fire. Note that guards attached to flows in StDNs rely on *existence* of a document satisfying some constraint on its valuations, not on *absence* of such document. One can not either model an inhibitor arcs (p, t) by defining an unsatisfiable pattern $\langle p, t\rangle$, as this would forbid firing of t regardless of whether p contains tokens or not. The translation of markings as described above shows that a place p is empty when $\nu(p) = 0$. One can hence check emptiness of a place p in marking M using a pattern $\langle p, t\rangle$ that checks existence of a node n_p in D_M carrying tag p and with valuation $\nu(p) = 0$. This pattern is obviously not monotonous. Fortunately, patterns of effective StDN suffice to encode many other intersting variants of Petri nets.

Proposition 5.4. Effective StDN can simulate Reset post-G Nets.

Proof:
We show this proposition by providing a construction for wqo StDN that can simulate a Reset Post-G nets with StDNs that. We have shown at the beginning of this section that any marking M of a reset post-G net can be described by a structured document D_M of bounded depth.

For a given Reset Post-G net $\mathcal{N} = (P, T, F, M_0)$, we build a StDN $\mathcal{N}' = (P', T', F', M'_0)$, where $P' = \{p_{init}, p_{one}, p_{out}\}$, $P'_{DB} = \{p_{one}\}$, $T' = T \cup \{t_{in}, t_{one}, t_{out}\}$, $M'_0(p_{out}) = (D_{M_0}, id)$ for some arbitrary identifier id, and $M'_0(p_{one})$ contains a single token which document is of any type (say • for instance). The flow relation F' is built as follows:

- F' contains flow (t_{in}, p_{in}), with associated query $\langle t_{in}, p_{in}\rangle$ that returns D_{M_0}.
- F' contains flows (p_{one}, t_{one}) and (p_{in}, t_{one}) which associated patterns are P_{tt}. F' also contains flow (t_{one}, p_{out}), with associated query $id_{p_{in}}$ (i.e, t moves one document from p_{in} to p_{out}. This ensures that transition t_{one} is fired only once in any execution of $\mathcal{N}'$ and copies one initial marking document D_{M_0}.
- F' contains flows (p_{out}, t) and (t, p_{out}) for every transition $t \in T$. Guards are set as follows: each guard $P_t =< p_{out}, t >$ is a pattern that checks existence of a node n_p with tag p and value $\nu(p) > 0$ if $F(p, t) = p$. If $F(p, t) \in \mathbb{N}$, then pattern P_t checks existence of a node n_p with tag p and value $\nu(n_p) \geq F(p, t)$. Then each flow (t, p_{out}) is associated with a query $\langle t, p_{out}\rangle$ that computes $D_{M'}$ from any input document D_M, where M' is the marking obtained after firing t from M. The query $\langle t, p_{out}\rangle$ computes $D_{M'}$ as a simple update of value $\nu(p)$ attached to each node n_p in D_M as follows. If $F(p, t) = p$ then $\nu'(p) = 0 + val(F(t, p), M)$. If $F(p, t) = x$ for some $x \in \mathbb{N}$, then $\nu'(p) = \nu(p) - x + val(F(t, p), m)$. Every query $\langle t, p_{out}\rangle$ is hence clearly deterministic, monotonous, and backward effective.

The structure of net $\mathcal{N}'$ is illustrated in Figure 21. Now it is obvious that at any time, net $\mathcal{N}'$ can fire transition t_{in}, but that only one document D_{M_0} produced by this transition can be moved from p_{in} to p_{out}. We now define a relation R from markings of $\mathcal{N}$ to marking of $\mathcal{N}'$ as follows: $M_{\mathcal{N}}\, R\, M_{\mathcal{N}'}$ iff $M_{\mathcal{N}'}$ is a marking that associates token $(D_{M_{\mathcal{N}}}, id)$ to place p_{out}, or $M_{\mathcal{N}} = M_0$ and $M_{\mathcal{N}'}(p_{out}) = \emptyset$. We can easily show that considering firings of t_{init} and t_{one} as unobservable moves, R is indeed a simulation relation: $M_0 R M'_0$, and for every $M_1\ R\ M'_1$ any move $M_1[t\rangle M_2$ of the initial net $\mathcal{N}$ can be simulated by a sequence $M'_1[t^*_{init}.t.t^*_{init}\rangle M'_2$ of moves of $\mathcal{N}'$. □

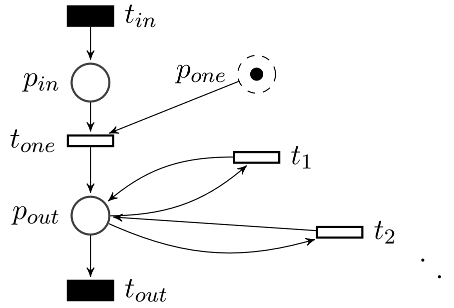


Figure 21. Encoding Reset Post-G-Nets

One can notice that the patterns used to encode a Reset post-G net with a StDN are monotonous, and that given a transition t and a marking M of the net N', one can easily recompute a basis for the set of possible markings preceding M. According to [10], coverability is decidable for Reset post-G nets, which is compatible with our results in section 3.3, and furthermore, reset Post-G nets subsume the classes of reset Petri Nets, transfer post-G nets, transfer nets, post-G nets, post-SM nets, and of course Petri nets. So the encoding of Proposition 5.4 works with all these classes of nets.

We can now compare effective StDNs with another high-level variant of nets with decidable coverability, namely Nested Petri nets [9]. Nested Petri nets are labeled nets that use Petri net markings as tokens. They are strictly more expressive than Petri and Reset nets. The semantics of nested nets is defined using several kinds of moves: *transfer moves*, which can create, transfer or delete a token (a net marking) from a place, independent *inner moves* of a net within a place, that simply consist in firing a transition from a marking contained in some place, *horizontal synchronization*, which synchronizes two inner moves of two nets contained in the same place, and *vertical synchronization*, that synchronizes an inner move of a net within a place with a transfer move of the higher-level structure. Both synchronization steps suppose that one transition carries some label a and the other one an adjacent label $\bar{a}$.

Definition 5.5. (Nested Petri nets)
A *nested Petri net structure* is an indexed set of nets $\mathcal{N}_i = (P_i, T_i, F_i, \lambda_i)$ where $\mathcal{N}_2, \ldots, \mathcal{N}_k$ are called *element nets*, and $\mathcal{N}_1$ is called a *system net*. P_i's and T_i's are disjoint sets of places and transitions, and every $F_i \subseteq P_i \times T_i \times E \cup T_i \times P_i \times E$ is a flow relation, where E denotes a set of *expressions*. Every transition t from element and system nets carry a label $\lambda(t)$ from a finite set L. Two labels from L are *adjacent* if one of them is some letter a, and the other one its complement $\bar{a}$. Element nets have markings over finite sets $S_2, \ldots, S_k$. We denote by $\mathcal{M}$ the set of markings of element nets. A marking of an element net $\mathcal{N}_i$ associates a multiset of elements from S_i to each place of P_i. A

marking of a nested net is a map that associates to places of its system net a multiset of markings from its element nets. Flows of (element and system) nested nets are labeled by expressions that sum variable names and constant from $S_2 \cup \cdots \cup S_k \cup \mathcal{M}$, i.e. expressions of the form $v_1 + \dots v_x + c_1 + \cdots + c_y$ where v_i's are variable names representing tokens, and c_j's are constant values. If the considered net is $\mathcal{N}_1$, constants take value in $\mathcal{M}$, and if the considered net is $\mathcal{N}_i$, with $i \in 2 \dots k$, constants take values in S_i. Last, it is required that expressions on input arcs do not contain twice the same variable name, and do not contain constants. In other words, expressions on input arcs are used to select some tokens (bind variables), and constants in output arcs are used to create new tokens.

Figure 22 shows an example of nested Petri net structure. $\mathcal{N}_1$ is a system net with three transitions labeled $\{a, b, c\}$, $\mathcal{N}_2$ and $\mathcal{N}_3$ are element nets with two transitions labeled respectively with $\{\bar{c}, d\}$ and $\{\bar{d}, e\}$. Markings of $\mathcal{N}_1$ associate multisets of markings of $\mathcal{N}_2$ and $\mathcal{N}_3$ to places of $P_1 = \{p_1, p_2, p_3\}$. Markings of $\mathcal{N}_2$ associate multisets of elements from $S_2 = \{s_1, s_2, s_3\}$ to places of $P_2 = \{p_4, p_5\}$, and markings of $\mathcal{N}_3$ associate multisets of elements from $S_3 = \{s_4\}$ to places of $P_3 = \{p_6, p_7\}$.

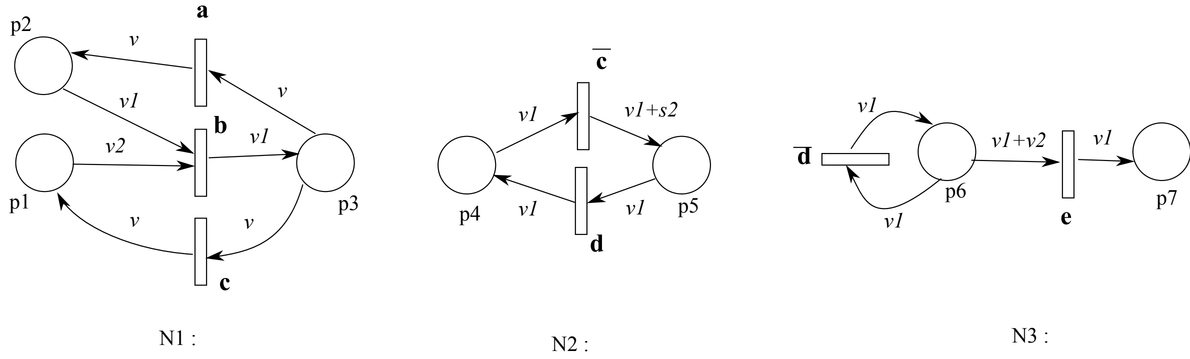


Figure 22. Nested net example

We can now give the semantics of Nested nets. A *binding* for a set of variables V is a map associating an element from $\mathcal{M} \cup S_2 \cup \dots S_k$ to every variable in V. A binding for a transition t is a binding for the set of all variables appearing in expressions labeling input arcs of t. For a binding b and an expression e, we denote by $b(e)$ the expression obtained by replacing every variable in e by its value. $b(e)$ can be interpreted as a multiset over $\mathcal{M}$ or any of the S_i's. A transition t of a net is firable from a marking M iff there exists a binding b for t such that for every place p with input flow $(p, t, e_{p,t})$, we have $M(p) \geq b(e)$.

Firing an *inner transition* t from a marking M contained in some place of the system net consists in finding a binding b for variables in expressions labeling input arcs of t that allows firing of t. Then these tokens are consumed to produce a temporary marking M_{tmp} ($M_{tmp}(p) = M(p) - b(e_{p,t})$ for every place in $^\bullet t$) and new tokens are produced in output places to produce a final marking M' such that $M'(p) = M_{tmp}(p) + b(e_{t,p})$ for every place in $t^\bullet$. We write $M \xrightarrow{a} M'$ when a move from M to M' using a transition carrying label a exists.

Transfer transitions behave similarly, but at the system net level, that is bindings take value in $\mathcal{M}$. Such transitions allow to create new element net markings in places of the system net, transfer or delete markings. Similarly, we write $M \xrightarrow{a} M'$ when a move from M to M' using a transition carrying label a exists.

An *horizontal synchronization* consists in finding a pair of markings M_1, M_2 in the same place of the system net such that $M_1 \xrightarrow{a} M_1'$ and $M_2 \xrightarrow{a'} M_2'$ such that a and a' are adjacent labels, and then applying the effect of both selected transitions to M_1 and M_2.

A *vertical synchronization* consists in finding a markings M_1 in some place of the system net such that an element net allows move $M_1 \xrightarrow{a} M_1'$, and a transition $M_2 \xrightarrow{a'} M_2'$ of the system net such that a and a' are adjacent labels, and then applying simultaneously the effect of selected transitions.

On the example of Figure 22, one can remark that the system net $\mathcal{N}_1$ can synchronize vertically with a transition from a marking of the element net $\mathcal{N}_2$ via labels $c, \bar{c}$, and that a pair of markings for nets $\mathcal{N}_2, \mathcal{N}_3$ can synchronize horizontally via letters $d, \bar{d}$.

Proposition 5.6. Effective StDNs can simulate nested Petri nets.

Proof:
Let $(\mathcal{N}_i)_{i\in 1..k}$ be a nested Petri net structure over an alphabet of labels L with initial marking M_0 (that associates a multiset of markings of nets $\mathcal{N}_2, \dots \mathcal{N}_k$ to places of $\mathcal{N}_1$). We use a net structure similar to the one used for the encoding of a Reset post-G nets, but with adapted marking documents and more transitions. We design a StDN $\mathcal{N} = (P', T', F', M_0')$, where $P' = \{p_{in}, p_{out}, p_{one}\}$, $T' = \{t_{in}, t_{out}, t_{one}\} \cup T_0 \times B$, where B considers the set of all possible bindings for variables appearing in element net structure, and T_0 is a set of transitions representing possible moves of the net structure. More precisely, we have: $T_0 = T_{inner} \cup T_{transfer} \cup T_{horiz} \cup T_{vert}$ with

$$\begin{array}{ll} T_{inner} = & \{t_k \mid t_k \in T_2, \dots T_k\} \\ T_{transfer} = & \{t_k \mid t_k \in T_1\} \\ T_{horiz} = & \{t_{t_1,t_2} \mid t_1, t_2 \in T_2, \dots T_k \wedge l(t_2) = \overline{l(t_1)}\} \\ T_{vert} = & \{t_{t_1,t_2} \mid t_1 \in T_1, t_2 \in T_2, \dots T_k \wedge l(t_2) = \overline{l(t_1)}\} \end{array} .$$

The flow of net $\mathcal{N}$ follows the same principle as for the encoding of reset post-G nets shown in proof of Proposition 5.4, and illustrated in Figure 21, but including $T_0 \times \{p_{out}\} \cup \{p_{out}\} \times T_0$ in the construction of the flow relation connected to place p_{out} instead of $T \times \{p_{out}\} \cup \{p_{out}\} \times T$. The main difference lays in the definition of markings for this net and of patterns and queries attached to flows from/to transitions of $T_{inner} \cup T_{transfer} \cup T_{horiz} \cup T_{vert}$.

Markings: For every marking M of a nested net, we compute a document D_M as follows : D_M has a root node n_0 carrying tag N_1, with $|P_1|$ children nodes $n_1, \dots n_{|P_1|}$ that carry respectively tags $p_1, \dots p_{|P_1|}$. The root and its children depict the system net level. Hence, every tree rooted at a node with tag p_q representing place p_q is an encoding of a set of net markings. Let $M(p_i) = M_1, \dots M_j$ be the set of net markings attached to place p_i in net $\mathcal{N}_1$. In document D_M, every node n_i has as many successors $n_i^1, \dots n_i^j$ as there are net markings in $M(p_i)$. Each marking is a marking from an element net, that will be again represented as a subtree. So, each node n_i^q of D_M is attached a tag from $N_2, \dots N_k$: the tree starting at node n_i^q carries a tag N_x if it depicts a marking of net N_x. It now remains to build a subtree rooted at n_i^q representing a marking of a standard Petri net. Then, every n_i^q carrying tag N_x has $|P_x|$ children $n_i^{q,1}, \dots n_i^{q,|P_x|}$, carrying tags $p_{1,x} \dots P_{|P_x|,x}$ representing places of net $\mathcal{N}_x$. Each node $n_i^{q,y}$ has $|S_x|$ successors, and each of these successors is attached a tag $s \in S_x$ and a valuation $\nu(s)$ that associates to tag s the number of tokens of type s that marking M_q associates to place y. Note that all documents obtained from markings are of bounded depth. Figure 23 shows a marking document for the nested net of Figure 22.

Patterns and Queries: It now remains to describe patterns and queries attached to transitions, that encode the semantics of nested nets. Let $t \in T_i$, ${}^{\bullet}t = p_1, \dots p_n$ and let the flow from p_i to t be labeled by an expression e_i of the form $v_1 + \dots + v_{x_i}$. For a binding $b : \{v_i\}_{i \in 1..x_i} \rightarrow S_i$, let $b(e_i)$ denote the multiset obtained by replacing every variable in e_i by its bound value in b, let $\mathrm{Im}(b)$ denote the set of symbols used by b, and $|b(e_i)|_j$ the number of occurrences of symbol $s_j \in S_i$ in multiset $b(e_i)$.

The constraint on flow from p_i to t is validated in marking M by binding b if $M(p_i) \geq b(e_i)$, i.e. place p_i contains more tokens of each kind than required by $b(e_i)$. Note that the number of possible bindings for element nets expressions is finite. Transition t is firable in M if there exists a binding b such that constraints of its input flows are validated in M by b. We let pattern $P_{t,inner,b}$ denote a pattern that checks existence of a marking of net of type N_i in some place of the nested net, for a fixed binding b. If ${}^{\bullet}t = p_1, \dots p_n$, this pattern is simply a tree with root n_0 with empty constraint, a children node n_1 with empty constraint too, representing any place of the system net (the place in which one is looking for some marking). Node n_1 must have a successor n_2 carrying tag N_i with n children nodes respectively with tags $p_1, \dots p_n$. Each of these n nodes has $k = |\mathrm{Im}(b)|$ successors. Each of these successors is tagged by a distinct constraint $\nu(s_j) \geq |b(e_i)|_j$. If this pattern is satisfied by a marking document D_M, then binding b allows firing transition t from M. Note that this pattern is monotonous.

We can now design a non-deterministic query $Q_{t,inner,b}$ that given a marking document D_M searches a mapping μ associating nodes of $P_{t,inner,b}$ to nodes of D_M and computes the effect of firing transition t from M with binding b. In some sense, μ chooses which marking (subtree of D_M) is updated. The effect of firing t with binding b results in a new marking M'. One computes $D_{M'}$ from D_M by changing the value $\nu(s_j)$ attached to nodes contained in the subtree representing an inner marking. These nodes are leaves of D_M in the image $\mu(P_{t,inner,b})$, or their neighbors. Let the input flow from p to t be labeled by some expression e_i, and the output flow from t to p be labeled by an expression e'_i (an expression of the form $v_1 + \dots + v_x + c_1 + \dots c_y$), and let n be a node representing the number of tokens of type s_j in place p in the inner marking subtree selected by μ. Then, in $D_{M'}$, node n is attached a new valuation ν' such that tags of n remain unchanged, but $\nu'(s_j) = \nu(s_j) - |b(e_i)|_j + |b(e'_i)|_j$.

One can similarly design patterns that check existence of a binding allowing a transfer transition, or of a pair of bindings for two transitions performing a vertical/horizontal synchronization. Bindings for transfer transitions apply to infinite sets of markings, but for such bindings, one needs not be exhaustive, and it is sufficient to check that there exists as many markings in input places of a system net transition t as there are variables in input flow expressions of t, which can be easily done on the document representation of a marking. Then queries attached to transfer transitions simply copy subtrees from the original marking document, and create new subtrees corresponding to constants (there exists only a finite number of such constants) in the output flows expressions.

As for the encoding of reset post-G nets, one can show that any firing of a transition t can be simulated by a sequence of transitions $t^*_{init}.t.t^*_{init}$ in the corresponding StDN, and find a simulation relation R from any marking M of the nested net structure $\mathcal{N}_1, \dots \mathcal{N}_k$ to all markings of N such that p_{out} contains document D_M. □

A remaining question is whether effective StDNs can encode Petri nets with tokens that carry data (PNTCD for short). Following the definition of [1], markings of PNTCDs are finite sequences of vectors in $\mathbb{N}^P \setminus 0^P$. Clearly, this kind of data structure can be represented very easily with bounded depth documents, which are hence well-quasi ordered. Transitions of PNTCD are defined as matrices subtractions, multiplication and additions, that are proved to be backward effective (PNTCD are WSTS with

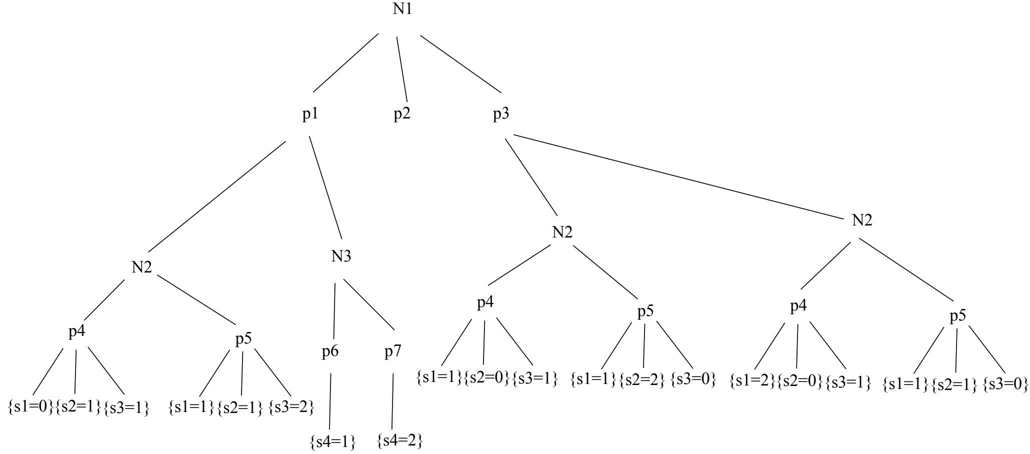


Figure 23. Encoding for the marking of a nested net as a structured document.

effective pred-basis). Though we do not provide an encoding of the matrices operations in this paper, we conjecture that PNTCD can be simulated with effective StDNs.

6. Related Work

Structured Data nets are a model for transactional systems with data, such as web-based applications. They emphasize workflows and data manipulations. Many formalisms outside the Petri net community have been proposed for the design and orchestration of complex workflows with data, like Active XML (AXML) [11], Business artifacts [12], Guard stage milestones [13], ORC [14],... One can also mention several initiatives to model web-services in the pi-calculus community (μ-se [19], CASPIS [20], COWS [21], to mention only a few). None of the above cited formalisms can be qualified of "open data-centric workflow model", and they all are confronted to undecidability issues.

Business artifacts [12] describe the logic of transactions for systems equipped with databases. A transaction carries variables, which are instantiated by values collected along the workflow or entered by the user. The workflow of a transaction has been defined both using automata with guards on variables values and on contents of database, or via logical rules defining the logic of a transaction. Verification of Business Artifacts has been proved feasible in a class of specifications where cyclic behaviors can reuse data in a restricted way [34]. In their original version, Business Artifacts consider sequential processing of cases, and can not define parallel threads. They have inspired another model called Guard Stage Milestones (GSM) [13] that allows some parallelism among tasks.

Session Systems is a model for transactional systems proposed in [16]. It allows for unbounded numbers of concurrent transactions among unbounded numbers of agents. Upon some restrictions on the behaviors of agents, verification of coverability properties and simple business rules are decidable. As in our case, this model attaches unique identifiers to transactions. However, this model emphasizes more on coordination among agents, and restricts the use of data to finite sets of variables with finite domains owned by agents.

Programming languages approaches. BPEL (Business Process Execution Language) and ORC are programming languages, that have been proposed to model complex workflows. BPEL [15] is an executable language, and has become the standard to design Business processes. A BPEL specification describes a set of independent communicating agents with a rich control structure. Coordination is achieved through message-passing. Interactions are grouped into sessions implicitly through *correlations*, which specify data values that uniquely identify a session—for instance, a purchase order number. ORC [35] is a programming language for the orchestration of services. It allows algorithmic manipulation of data, with an orchestration overlay to start new services and synchronize their results. ORC has better mechanisms to define workflows than BPEL, but lacks the notion of transaction identity that is essential to establish sessions among the participants in a service. ORC does not use structured data a priori, but can handle any kind of data type. Data manipulations are implemented through functions called *sites*. Data circulation is described by connecting inputs and outputs of sites via connectors. This allow to apply arithmetic operations to elements of a stream, to filter elements, or conveniently select the first value received from several streams.

Data-centric approaches. Several ingredient of our model are inspired from AXML. AXML [11] defines web services as a set of guarded rules that transform semi-structured documents described, for instance, in XML. However, it does not make workflows explicit, and does not have a native notion of transaction either. To implement a sequential workflow in an AXML specification, one has to explicitly integrate control states to documents, guards and rules, which can be cumbersome. A decidable subclass of AXML called "positive" AXML [36] has been identified. Rules in positive specifications can only append data to a document. This monotonicity allows to decide simple properties. For recursion free AXML, a simple logic (LTL-tree logic) has been shown decidable [37]. A different formalization of AXML has been proposed in Tree Pattern Rewrite Systems (TPRS) [2]. TPRS systems define rules, that can be applied to append data, synthesize or remove information from documents. Coverability of some configuration is decidable when a specification can only reach configurations in which documents are of bounded depth.

Process Algebras. A lot of efforts have also been devoted to services and transactional systems modeling in the π-calculus community. *Session types* [18] have been proposed as a formal model for web services, and have then been enhanced to capture various features such as multiple instantiations of identical agents [38] and nested sessions [39]. Session types have been used to determine whether an otherwise unconstrained set of processes adheres to a communication discipline specified by a session type, or to model and verify security issues such as information flows and access control problems. The expressive power of the whole π-calculus and session types do not allow for verification of reachability or coverability properties. However, [40] uses WSTS to show that a fragment of spatial logic is decidable for the fragment of well-typed π-calculus processes. The considered fragment can express safety properties. A solution to covering problems for π-calculus with bounded depth has been proposed in [3]. This work shows that for bounded depth π-calculus, a forward coverability algorithm (EEC) terminates, even if the bound is unknown.

Several variants of π-calculus have been proposed to model services. A variant of ORC and π-calculus is proposed by Lanese *et al.* [41]: processes communicate via streams, and choices of a process are implemented as external choices (if-then-else constructs can be implemented this way). This model has an interesting expressive power, as it allows to select values from (ordered) streams, minimal elements, etc. In particular, it allows for the selection of first arrived values (as in ORC), and this feature

can not be implemented in a simple way with our model. The counterpart of this expressiveness lies in the undecidability of non-trivial properties (in the sense of Rice's theorem). A multiparty session formalism called $\mu - se$ is proposed by [19]. As in StDNs, they avoid an explicit handling of sessions identities. Sessions in $\mu - se$ are located on sites, and allow participating processes to communicate in a private way. Additional communications are allowed among processes that are located on the same site. Arbitrary numbers of sessions can be created on a site, and a session can handle an arbitrary number of joined processes. A merging mechanism allows a process to enter a session at any point, and persistent sessions defining offered services can be specified. Communications are handled as usual in π-calculus. The *CASPIS* formalism [20] was influenced by the π-calculus and by ORC, and designed to orchestrate services. It provides pairwise sessions, modeled as service calls which create private names shared by the caller and callee of a session, and pipelining, i.e. a way for a service P to call another service Q whenever a new value is produced by P. Unlike the preceding π-calculus variants, CASPIS allows guarded sums (i.e. internal choices), and gives ways to terminate sessions before their final completion. *Conversation types* [42] is an extension of π-calculus that replaces channel based communications by context sensitive message based communications. A conversation is a behavioral type describing multi party interactions among processes. [42] provides typing mechanisms to ensure that conversations are implemented by processes in a compatible way, and that processes can never get stuck during concurrent transactions. A conversation allows for unbounded number of participants. The *COWS* approach (see for instance [21]) introduces a complete language for the orchestration of stateful services. It proposes correlation variables that implement correlations of messages as in BPEL, a wait operation to suspend processes for a chosen time and a kill operation, that terminates terms within a delimited scope. Though data is not a first class citizen in COWS, this formalism allows for data manipulation, and can use operations on semi-structured data to perform message correlations *à la* BPEL.

Let us now compare features of StDNs with features of other formalisms. Our model is data centric: it also allows complex manipulation of semi-structured data. It enables the specification of transactions, data storage, and workflows. The model, its expressiveness and decidable properties can be adapted by simply changing the pattern and query languages attached to flows. However, several convenient modeling features are missing. Some of them are purely syntactical: Database operation (filtering, join, etc.), global patterns, read arcs can be simulated by our model and could be seen as macros. However, the model still misses simple ways to end a transaction: There is no way to kill a transaction, and one has to wait for all document belonging to a failed transaction to be consumed by the final transition t_{out}. Mechanisms to reset all documents carrying some identifier can be implemented (as shown in section 4), but only with non-effective StDNs. A second missing aspect of StDN that was successfully implemented in π-calculus variants and in session systems is multiparty sessions. In StDNs, a transaction comes from the environment. External choices can be modeled by non-deterministic transitions, and a terminated case is erased, i.e, returned to the environment. However, a transaction is, in some sense, a point to point conversation between a StDN and its environment, there is no notion of session users, or groups contributing to a process.

7. Conclusion

This paper has defined an extension of Petri Nets whose transitions manipulate structured data via patterns and queries. In its full generality this model is Turing Powerful. However, under some restrictions on the nature of queries and on the shape of documents some interesting properties, such as coverability, are decidable. We believe that limiting data to structured documents of bounded depth with wqo labels is a sensible approach: many information systems use only strings, booleans, integers, but do not need domains that are not well quasi ordered such as real numbers with arbitrary precision.

Several improvements might be investigated. An important issue is to identify classes of data operations that allow StDNs to fall into decidable subclasses. Our coverability proof relies on backward effectiveness of transitions to obtain effective StDNs, i.e. obtain effective WSTS. This does not identify a particular class of queries. To be practical, we would like to identify classes of non-trivial monotonous queries that ensure effectiveness. Decidability results for positive active XML [36], for instance, use another form of monotonicity: documents can only grow. This is an adequate assumption in case management systems. Considering positive StDN could be a way to ensure effectiveness. Another improvement lies in pattern expressiveness. Currently, only individual constraints on data values are attached to nodes. One could, however, consider patterns with constraints of the form $v \cdot \sigma \leq v' \cdot \sigma'$, involving values of several nodes, sets of patterns requiring matching on several documents from a place or boolean combinations of patterns and see how these extensions affect the model properties. Another line of research concerns symbolic manipulation of upward closed sets of documents. So far, we only have studied coverability for symbolic set of initial cases, but we can imagine to define symbolic sets of initial markings, database contents, or target markings to cover. In a similar way, we would like to use adapted safety properties: as discussed in this paper, checking safety-like properties would probably require restricting documents to well quasi ordered documents over intersection effective domains. We also want to consider extensions of the model with some essential features for web services and transactional systems, for instance to allow cancellation of a transaction. Such a feature is currently not handled by our model, but is important, as an StDN might not be sound, even when it is effective. Another missing ingredient is the possibility to define multiparty transactions, these are essential to design, for instance chat systems or group communication.

So far, StDNs were not used on real case studies. Even if the formalism allows simulating any kind of program, it is obviously too low-level to be used as an engineering tool. One way to circumvent this drawback is to equip the language with a syntax adapted to users needs, containing macros (to query, copy, sort, or filter data), and to define the semantics of systems depicted with this syntax trough an equivalent StDN. Then a system defined this way can be verified if it falls in one of the decidable classes highlighted in this paper. Most of real systems will fall in undecidable classes. This however raises the issue of sound abstraction techniques from general classes of StDN to decidable subclasses to allow verification of StDNs for real-life systems. Abstraction is also a key issue to improve efficiency of coverability algorithms, which usually have high complexities.

Acknowledgements: We would like to thank anonymous reviewers who helped improving this work through their careful reading and useful suggestions.

References

[1] Lazic R, Newcomb T, Ouaknine J, Roscoe AW, Worrell J. Nets with Tokens which Carry Data. Fundam Inform. 2008;88(3):251–274.

[2] Genest B, Muscholl A, Wu Z. Verifying Recursive Active Documents with Positive Data Tree Rewriting. In: Proc. of FSTTCS 2010. vol. 8 of LIPIcs; 2010. p. 469–480.

[3] Wies T, Zufferey T, Henzinger TA. Forward Analysis of Depth-Bounded Processes. In: FOSSACS. vol. 6014 of LNCS. Springer; 2010. p. 94–108.

[4] van der Aalst WMP. The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems, and Computers. 1998;8(1):21–66.

[5] van Hee KM, Hidders J, Houben G, Paredaens J, Thiran P. On the relationship between workflow models and document types. Inf Syst. 2009;34(1):178–208.

[6] Jensen K. Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1, Second Edition. Monographs in Theoretical Computer Science. An EATCS Series; 1996.

[7] Genrich H. Predicate/Transition Nets. In: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986. vol. 254 of LNCS. Springer; 1986. p. 207–247.

[8] Lenz K, Oberweis A. Modeling Interorganizational Workflows with XML Nets. In: 34th Annual Hawaii International Conference on System Sciences (HICSS-34); 2001. .

[9] Lomazova IA, Schnoebelen P. Some Decidability Results for Nested Petri Nets. In: Perspectives of System Informatics. Springer; 1999. p. 208–220.

[10] Dufourd C, Finkel A, Schnoebelen P. Reset Nets Between Decidability and Undecidability. In: Proc. of ICALP'98. vol. 1443 of LNCS. Springer; 1998. p. 103–115.

[11] Abiteboul S, Benjelloun O, Manolescu I, Milo T, Weber R. Active XML: A Data-Centric Perspective on Web Services. In: BDA02; 2002. .

[12] Nigam A, Caswell NS. Business artifacts: An approach to operational specification. IBM Syst J. 2003 July;42:428–445. Available from: `http://dx.doi.org/10.1147/sj.423.0428`. doi:http://dx.doi.org/10.1147/sj.423.0428.

[13] Hull R, Damaggio E, Fournier F, Gupta M, Heath FT, Hobson S, et al. Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In: Proc. of WS-FM 2010. vol. 6551 of LNCS. Springer; 2011. p. 1–24.

[14] Kitchin D, Cook W, Misra J. A Language for Task Orchestration and Its Semantic Properties. In: CONCUR'06; 2006. p. 477–491.

[15] Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, et al.. Business Process Execution Language for Web Services (BPEL4WS). Version 1.1; 2003. Available from: `http://xml.coverpages.org/BPELv11-May052003Final.pdf`.

[16] Akshay S, Hélouet L, Mukund M. Sessions with an unbounded number of agents. In: ACSD'14. vol. 4281. IEEE; 2014. p. 166–175.

[17] Badouel E, Hélouët L, Kouamou GE, Morvan C. A Grammatical Approach to Data-centric Case Management in a Distributed Collaborative Environment. In: SAC'15. ACM; 2015. p. 1834–1839.

[18] Honda K, Yoshida N, Carbone M. Multiparty asynchronous session types. In: POPL. ACM; 2008. p. 273–284.

[19] Bruni R, Lanese I, Melgratti HC, Tuosto E. Multiparty Sessions in SOC. In: COORDINATION. vol. 5052 of LNCS. Springer; 2008. p. 67–82.

[20] Boreale M, Bruni R, De Nicola R, Loreti M. Sessions and Pipelines for Structured Service Programming. In: FMOODS. vol. 5051 of LNCS. Springer; 2008. p. 19–38.

[21] Pugliese R, Tiezzi F. A calculus for orchestration of Web services. J Applied Logic. 2012;10(1):2–31.

[22] Abdulla PA, Cerans K, Jonsson B, Tsay YK. General Decidability Theorems for Infinite-State Systems. In: Proc. of LICS'96. IEEE; 1996. p. 313–321.

[23] Finkel A, Schnoebelen P. Well-structured transition systems everywhere! Theor Comput Sci. 2001;256(1-2):63–92. Available from: http://dx.doi.org/10.1016/S0304-3975(00)00102-X. doi:10.1016/S0304-3975(00)00102-X.

[24] David C. Complexity of Data Tree Patterns over XML Documents. In: Mathematical Foundations of Computer Science. vol. 5162 of LNCS; 2008. p. 278–289.

[25] Miklau G, Suciu D. Containment and equivalence for a fragment of XPath. J ACM. 2004;51(1):2–45.

[26] World Wide Web Consortium. XML Path Language (Xpath). W3C; 1999. W3C Recommendation,http://www.w3.org/TR/xpath.

[27] Ding G. Subgraphs and well-quasi-ordering. In: Journal of Graph Theory. vol. 16(5); 1992. p. 489 – 502.

[28] OASIS. Web Services Business Process Execution Language. OASIS; 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[29] World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C; 1999. W3C Recommendation,http://www.w3.org/TR/xquery.

[30] Gostellow K, Cerf V, Estrin G, Volansky S. Proper Termination of Flow-of-control in Programs Involving Concurrent Processes. In: Application and Theory of Petri Nets ICATPN '97. vol. 7(11) of ACM Sigplan. Springer; 1972. p. 15–27.

[31] van der Aalst WMP. Verification of Workflow Nets. In: Application and Theory of Petri Nets 1997 ICATPN '97. vol. 1248 of LNCS. Springer; 1997. p. 407–426.

[32] Mlynkova I, Toman K, Pokorný J. Statistical Analysis of Real XML Data Collections. In: Proc. of International Conference on Management of Data'06. Tata McGraw-Hill; 2006. p. 15–26.

[33] Higman G. Ordering by divisibility in abstract algebras. Proc London Math Soc (3). 1952;2:326–336.

[34] Damaggio E, Deutsch A, Vianu V. Artifact systems with data dependencies and arithmetic. ACM Trans Database Syst. 2012;37(3):22.

[35] Misra J, Cook W. Computation Orchestration. Software and Systems Modeling. 2007;6(1):83–110.

[36] Abiteboul S, Benjelloun O, Milo T. Positive Active XML. In: Proc. of PODS'04. ACM; 2004. p. 35–45.

[37] Abiteboul S, Segoufin L, Vianu V. Static analysis of active XML systems. ACM Trans Database Syst. 2009;34(4).

[38] Deniélou P, Yoshida N. Dynamic multirole session types. In: POPL; 2011. p. 435–446.

[39] Demangeon R, Honda K. Nested Protocols in Session Types. In: CONCUR; 2012. p. 272–286.

[40] Acciai L, Boreale M. Deciding Safety Properties in Infinite-State Pi-Calculus via Behavioural Types. In: ICALP (2). vol. 5556 of LNCS. Springer; 2009. p. 31–42.

[41] Lanese I, Martins F, Vasconcelos VT, Ravara A. Disciplining Orchestration and Conversation in Service-Oriented Computing. In: SEFM. IEEE Computer Society; 2007. p. 305–314.

[42] Caires L, Torres Vieira H. Conversation types. Theor Comput Sci. 2010;411(51-52):4399–4440.