Soundness Verification of Data-Aware Process Models with Variable-to-Variable Conditions

Paolo Felli
Free University of Bozen-Bolzano
Bolzano, Italy

Massimiliano de Leoni
University of Padova
Padova, Italy

Marco Montali
Free University of Bozen-Bolzano
Bolzano, Italy

Abstract. Traditionally Business Process Modeling has only focused on the control-flow perspective, thus allowing process designers to specify the constraints on the activities of the process: the order and potential concurrency of their execution, their mutual exclusivity, the possibility of being repeated, etc. However, activities are executed by different resources, manipulate data objects and are constrained by the state of such objects. This requires that the traditional notion of soundness, typically introduced for control-flow-only models, is extended so as to consider data. Intuitively, a (data-aware) process model is sound if (1) it does not contain deadlocks, (2) no more activities are enabled when the process instance is marked as completed and finally (3) there are no parts of the model that cannot be executed. Although several data-aware notations have been introduced in the literature, not all of these are given a formal semantics. In this paper, we propose a technique for checking the data-aware soundness for a specific class of such integrated models, with a simple syntax and semantics, building on Data Petri Nets (DPNs). These are Petri nets enriched with case variables, where transitions are guarded by formulas that inspect and update such variables, and are of the form variable-operator-variable or variable-operator-constant. Even though DPNs are less expressive than Petri nets where data are carried by tokens, they elegantly capture business processes operating over simple case data, allowing to model complex data-aware decisions. We show that, if a DPN is data-aware sound, the Constraint Graph is a finite-state automaton; however, a finite-state Constraint Graph does not guarantee data-aware soundness, but provides a finite structure through which this property can be checked. Finally,

Address for correspondence: paolo.felli@unibz.it

we investigate further properties beyond data-aware soundness, such as the problem of verifying that an actor participating in the business process can unilaterally enforce data-aware soundness by restricting the possible executions of a bounded DPN, assuming this actor to be able to control the firing of some transitions and decide the value of some of the case variables whenever these are updated.

Keywords: Soundness of Process Models, Data Perspective, Data Petri Nets,

1. Introduction

Traditionally Business Process Modeling has only focused on the control-flow perspective, thus allowing process designers to specify the constraints on the activities of the process: the order and potential concurrency of their execution, their mutual exclusion, the possibility of being repeated, etc. However, activities are executed by different resources, manipulate data objects, and are constrained by the state of such objects. By overlooking these aspects, the resulting process models do not accurately describe the domain of interest. In the light of above, the process analysts and software vendors have shown an increasing interest in enriching the traditional control-flow perspective of processes with additional dimensions, and the Business Process Management (BPM) field has witnessed a shift from conceptual models solely focusing on the flow of activities, to a so-called multi-perspective, *data-aware* process modeling. This extends the flow of activities with additional information such as the resources performing these activities and the data that is produced and manipulated by their execution. In particular, the integration between (business) processes and data has been extensively studied, so as to gain a more holistic understanding of how processes induce evolutions on data [1, 2]. This is a crucial aspect because the data produced by the activity executions may influence how the single process executions will proceed. Indeed, data-aware models are typically equipped with decisions, which are points in the process from where several potential, alternative paths exist. Each path of a decision point is associated with an expression (or condition) on the data of the process, and if any expression is evaluated to true on the current data values, then the corresponding path may be taken. The incorporation of these decision within process models is gaining momentum, as also testified by the recent introduction and development of the Decision Model and Notation (DMN), an OMG standard [3], and by the large share of recent research on the analysis of decision points [4, 5], and their discovery within the realm of Process Mining [6, 7, 8].

The fundamental problem of verifying the correctness of business process models has been traditionally tackled by exclusively considering the control flow perspective. This means that correctness is assessed by only considering the ordering relations among activities present in the model. In this setting, one of the most investigated formal notions of correctness is that of *soundness*, originally introduced by van der Aalst in the context of workflow nets [9]. Intuitively, it requires the three good properties of "certainty of clean termination", "absence of deadlocks" and "absence of dead fragments" in the process. These ensure that whenever a process instance is being executed, it always has the possibility of reaching its completion and, if it does so, then no concurrent thread is still active in the process. Additionally, it requires that the process does not contain dead activities that are impossible to enact. Importantly, this is ascertained by considering each process instance in isolation, and has

the effect of requiring the resulting control-flow to be finite-state (that is, in Petri net terms, bounded).

In order to apply the same principles and practices to models in which the control-flow perspective is integrated with the data dimension, we need to develop methods and techniques able to ascertain the correctness of data-aware processes. As an example, a model, although sound according to the intuitive definition given above, might be equipped with a decision point where the disjunction of the expressions of the available execution paths is not tautological. In this case, it may happen that, for some execution of this process, the current data values at that decision point are such that no path is enabled, yielding a deadlock. Ot it may happen that a cyclic path is always enabled, but the termination of the process cannot be reached. It may also be that one of the paths is associated with an expression that can never be satisfied, consequently yielding a dead path.

Previous approaches for assessing correctness of data-aware processes typically focused on single decisions [4], or simply on the interplay between decisions and their immediate corresponding outgoing branches [5]. More recent efforts have tackled this locality problem by considering soundness of the overall, end-to-end process, but have mainly remained at the foundational level [10, 11] as they do not come with actual techniques to effectively carry out the verification of data-aware soundness.

The literature proposes a large repertoire of modeling notations for data and processes. Such models differ one from another in terms of their: *(i)* degree of formal semantics, *(ii)* representation of the data and process component, and of their linkage, *(iii)* sophistication and expressiveness of the resulting approach. To be able to ascertain soundness of data-aware process models, we need a model that employs Petri nets as the underlying control-flow process backbone, and whose execution semantics is formalized in full.

The literature flourishes of data-aware extensions of Petri nets at different levels of sophistication, ranging from weak extensions where tokens carry single data elements [12, 13] to more expressive models where tokens carry complex structures [14, 15] and/or are equipped with modeling constructs akin to (subclasses of) Colored Petri nets and relational databases [16, 17, 18].

In this work, we adopt for our investigation the simple, yet rich Petri net-based model of Data Petri nets (DPNs) [19, 8, 20]. A DPN consists of a Petri net equipped with a fixed set of read-write variables that are inspected and updated by transitions. Even though DPNs are less expressive than Petri net-based models where data are carried by tokens, they come with a threefold advantage. First, the classical notion of soundness can be naturally lifted to a corresponding notion of data-aware soundness for DPNs. Second, DPNs elegantly capture the interesting class of activity-centric business processes that operate over scalar case data, and that use decision models based on the DMN S-FEEL standard [3] to route the process depending on process data [11]. Third, as witnessed by [8], several process mining techniques use DPNs as modeling notation, e.g., to discover decision points and their logic, as well as to perform conformance checking.

Nonetheless verifying soundness of DPNs is undecidable in general, e.g. when case data can be updated using arithmetical operators. To overcome this, we isolate a decidable class of DPNs that employs both non-numerical and numerical domains, and is expressive enough to capture data-aware process models equipped with S-FEEL DMN decisions [3], such as those recently proposed in [5, 11]. However the soundness of these DPNs cannot be directly checked by analyzing the state space, as it is infinite due to data variables even when the number of reachable markings is finite.

To tame this infinity, in [20, 21] we have taken inspiration from the technique of predicate abstrac-

tion [22] and presented an effective technique that verifies soundness by computing an abstraction of the given net. In [20] we considered DPNs which are not able to express variable-to-variable conditions on transitions, but provided a general abstraction technique to check the data-aware soundness of these nets. The technique is based on constructing an abstract state space which can be faithfully and effectively inspected for soundness. In [21] we have generalized this to study the fundamental case where the evolution of the process depends on the comparison between the values carried by different variables through linear inequalities.

In this paper we illustrate the resulting framework, detailing our sound and complete verification technique. Importantly, our approach paves the way towards the verification of properties that go beyond data-aware soundness. Indeed, in this paper we enrich the previous results by also showing how an analogous technique, based on the same type of abstraction as in [21], can be extended to further properties that, unlike data-aware soundness, require to analyze the specific (abstract) branching structure induced by DPNs. Specifically, we consider (as notable example) the problem of verifying that an actor participating in the business process can unilaterally enforce data-aware soundness by restricting the possible executions of a bounded DPN, assuming this actor to be able to control the firing of some transitions and decide the value of some of the case variables whenever these are updated.

The paper is organized as follows. In Section 2 we discuss related work, and in Section 3 we provide the necessary background on DPNs, formalize their execution semantics and the property of data-aware soundness. In Section 4 we detail our verification technique. Finally, in Section 5 we extend our approach beyond data-aware soundness and comment on future work in Section 6.

2. Related Work

A large body of research exists to verify the soundness of process models. Most of these works only focus on the control flow [23], starting from the seminal work of van der Aalst et al. [9]. These works ignore the data decision perspective, which is a significant limitation as also acknowledged by Sadiq et al. [24]. Corradini et al. focus on verification of BPMN choreographies and the message exchanges among collaborating processes [25, 26], but the focus remains on the control flow.

In fact, some attempts exist to also incorporate data and decisions. Sidorova et al. proposed a conceptual extension of workflow nets [27], where transitions update data predicates, namely the entire guards of transitions, instead of single data variable. As testified by modern process modeling notations such as BPMN and DMN, however, correctly modeling the data perspective requires data variables and full-fledged guards and updates. Calvanese et al. [4] focus on single DMN tables to verify whether they are correct or contain inconsistent, missing or overlapping rules. The work in [5] follows a similar approach, by defining a notion of decision-aware soundness that is based on the individual properties of the DMN decision tables [3] associated to decision points, taken in isolation. These are decision-deadlock freedom and dead decision absence. Intuitively, the former requires that a table specifies an output value for every possible case input to a decision (i.e., for every possible current assignment of variables), and that each of such outputs enables at least one subsequent path. The latter requires that for each of these possible paths, an output of the decision table that would enable them exists. In comparison, our notion of data-aware soundness is a global property considering the behavior of the entire net.

Knuplesch et al. [28] propose a technique to verify properties expressed in LTL against models that incorporate the data perspective. Unfortunately, classical soundness (and its data-aware extension) cannot be expressed with LTL formulae, as the "possibility of clean termination" is branching in nature, and also assumes a finite-trace semantics.

Relevant literature on Petri nets is also dedicated to the problem of determining whether a net can be controlled so as to drive the system it describes into a desired state or, conversely, avoid certain undesired conditions. This setting is tightly related to that of Supervisory Control Theory (SCT) for Discrete Event Systems (DES) [29, 30], with practical applications ranging from concurrent systems to flexible manufacturing. In these approaches, some of the possible transitions of the net are deemed as controllable and can be disabled by a supervisor, coupled to form a closed-loop system. Classical approaches for SCT are based on languages, where the DES to be controlled is seen as a language generator modeled as a deterministic finite-state automaton, and the desired behavior is also a language. SCT employs language-based definition of supervisors which are of no direct applicability for Petri net models [29], which led to the development of alternative state-based control formulations [31] (also because the controllable sublanguage is not guaranteed to be a Petri net language). Our approach in Section 5 is also state-based, but tailored for the special case of DPNs where the typical control-flow perspective of SCT is merged with the data dimension, and for which standard techniques cannot be applied without first abstracting data while still allowing the controlled system to exhibit the desired behavior. Also, in our setting transitions are nondeterministic due to the associated operations on data, the control of transition labels is decoupled from the control of variables, and the desired behavior of the system is not expressed by forbidden states nor languages but rather captures data-aware soundness. However, unlike this literature, we are not interested in computing the minimally restrictive controllers for the given specifications. We limit ourselves to nets with a bounded control-flow, which allow us to exploit analogous abstractions and procedure as those used for checking soundness.

3. Data Petri nets

In this section we illustrate how data-aware processes are represented by DPNs [8]. We maintain the formalization introduced in previous work [20, 21], which provided a full account of the syntax and semantics of these nets, allowing to lift the standard notion of soundness to their richer, data-aware setting. While the DPNs in [20] are restricted to allow only transitions associated to conditions of the form variable-operator-constant, those in [21] allow for variable-to-variable comparisons as well. In this paper we consider the latter setting.

We first define the notion of domain for case *variables*, assuming an infinite universe of possible values $\mathcal{U}$. A domain is a couple $\mathcal{D} = \langle \Delta_{\mathcal{D}}, \Sigma_{\mathcal{D}} \rangle$ where $\Delta_{\mathcal{D}} \subseteq \mathcal{U}$ is a set of possible values and $\Sigma_{\mathcal{D}}$ is a finite set of binary predicates on $\Delta_{\mathcal{D}}$. We require these predicates to be effectively computable, and the theory associated (which includes all FO formulae that are true in $\mathcal{D}$) is so that the satisfiability of predicates is decidable.

We consider a fixed set of domains, and in particular the notable domains $\mathcal{D}_{\mathbb{R}} = \langle \mathbb{R}, \{<, >, =\} \rangle$, $\mathcal{D}_{\mathbb{Z}} = \langle \mathbb{Z}, \{<, >, =\} \rangle$, $\mathcal{D}_{bool} = \langle \{\texttt{true}, \texttt{false}\}, \{=\} \rangle$, $\mathcal{D}_{string} = \langle \mathbb{S}, \{=\} \rangle$ which, respectively, account for real numbers, integers, booleans, and strings ($\mathbb{S}$ denotes here the infinite set of all finite strings). Given $\mathcal{D}$, the special symbol $\perp \in \Delta_{\mathcal{D}}$ is used to denote an undefined value. Although our

approach can be applied to arbitrary domains, from now on we restrict to the ones above. We also assume that $\Sigma_{\mathcal{D}}$ is closed under negation, namely that for every predicate $\odot$ also its complement is included.

Consider a set V of variables. Given a variable $v \in V$ we write v^r or v^w to denote that the variable v is, respectively, read or written by an activity in the process, hence we consider two sets V^r and V^w defined as $V^r \doteq \{v^r \mid v \in V\}$ and $V^w \doteq \{v^w \mid v \in V\}$. Intuitively, since an activity of the process may require to read and/or update the value of variables, as formalized later, we use v^r (resp., v^w) to denote the variable v before (resp., after) an activity, represented here as a transition, is executed. For this reason, we also refer to them as read and written variables, respectively. To talk about the possible values of variables, we need to assign them a domain. If a variable v has domain $\mathcal{D} = \langle \Delta_{\mathcal{D}}, \Sigma_{\mathcal{D}} \rangle$, for brevity we denote by $v_{\mathcal{D}}$ the corresponding *typed variable* as shorthand to specify that v takes values from $\Delta_{\mathcal{D}}$. Variables provide the basic building block to define logical conditions on the possible evolutions of the process, depending on the value of read and written variables. We call such conditions *constraints*.

Definition 3.1. Given a set of typed variables V, a constraint is an expression of the form:

- $v_{\mathcal{D}} \odot k$, where $v \in (V^r \cup V^w)$, $k \in \Delta_{\mathcal{D}}$ and $\odot \in \Sigma_{\mathcal{D}}$; or
- $v_{1\mathcal{D}} \odot v_{2\mathcal{D}}$, where $v_1 \in (V^r \cup V^w)$, $v_2 \in V^r$ and $\odot \in \Sigma_{\mathcal{D}}$.

We denote by $\mathcal{C}_V$ the set of all possible constraints on V.

A constraint allows to compare a variable $v_{\mathcal{D}}$ with a constant (among those in the associated domain $\Delta_{\mathcal{D}}$) or with another variable with the same domain $\mathcal{D}$. Parentheses around constraints are used throughout the paper to enhance readability. As shorthand notation, we denote by $(v \odot k)$ a constraint in which a variable, either read or written, is compared with a constant, i.e., where $v \in (V^r \cup V^w)$, $\mathcal{D}$ is the domain of v and $k \in \Delta_{\mathcal{D}}$. When we need to specify not only the shape of a constraint but also whether a variable is read or written, we then use the notations $(v^r \odot k)$ and $(v^w \odot k)$. Analogously, for comparing variables between them, we use the notations $(v_1 \odot v_2)$, $(v_1^r \odot v_2^r)$ and $(v_1^w \odot v_2^r)$. When the right-hand side can be either a constant or a variable, we use the symbol x. Finally, given a constraint $(v \odot x)$, we denote by $\neg(v \odot x)$ the constraint in which $\odot$ is replaced by its negation. The set of variables read and written by a constraint c is denoted by $read(c)$ and $write(c)$, respectively.

We use constraints to formalize the conditions that we can associate to activities in the process: each activity in the process, modeled as a transition in our net representation, is associated to a constraint called guard. These allow to model conditions between the value of a variable and a constant (e.g., $\mathtt{a}^r > 0$), between the written value of a variable and the current value of another variable (e.g., $\mathtt{a}^w > \mathtt{b}^r$) or between the current value of two variables (e.g. $\mathtt{a}^r > \mathtt{b}^r$), with $\mathtt{a}, \mathtt{b} \in V$.

Conjunctions and disjunctions are not allowed for simplicity but without loss of generality: disjunctive guards can be mimicked by having multiple transitions from and to the same places, each having a disjunct as guard, whereas conjunctive guards can be modeled as "non-interruptible" sequences, i.e., a sequence of transitions whose initial firing requires a token in a special lock place (where the token is sent back as the entire sequence is completed). All such sequences share the same lock place. The simplification is only aimed at simplifying the technical details that follow.

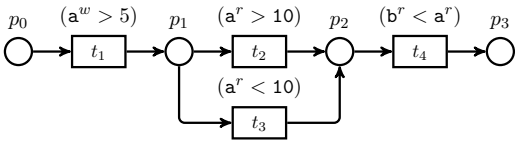

Figure 1. A DPN $\mathcal{N}$, where $M_I = \{p_0\}$ and $M_F = \{p_3\}$, $\mathtt{a}$ and $\mathtt{b}$ are integers and $\alpha_I(\mathtt{a}) = 0$ and $\alpha_I(\mathtt{b}) = 10$.

An assignment is a function $\beta : (V^r \cup V^w) \mapsto \mathcal{U}$, which assigns a value to read and written variables. Given β as above and a guard $c = (v \odot x)$, we say that c is true when variables are substituted as per β, written $c_{[\beta]} = \mathtt{true}$, iff x is a constant and $\odot(k, x)$ for $k = \beta(v)$ or x is a read variable and $\odot(k_1, k_2)$ for $k_1 = \beta(v)$ and $k_2 = \beta(x)$. In other words, a guard is satisfied by evaluating it after assigning values to read and written variables, as specified by β.

A state variable assignment, abbreviated hereafter as SV assignment, is a function $\alpha : V \mapsto \mathcal{U}$, which assigns values to each variable $v \in V$ so that $\alpha(v_{\mathcal{D}}) \in \Delta_{\mathcal{D}}$. The difference with an assignment is that while β may define a variable update when a transition is fired, a SV assignment α simply holds the current value of each and all variables. We can now formalize DPNs.

Definition 3.2. A Data Petri Net (DPN) $\mathcal{N} = (P, T, F, V, dom, \alpha_I, guard)$ is a Petri net (P, T, F) with additional components describing the additional perspectives of the process model:

- V is a finite set of process variables, and dom is a function assigning a domain $\mathcal{D}$ to each $v \in V$;
- α_I is the initial SV assignment;
- $guard : T \to \mathcal{C}_V$ returns a guard associated with the transition.

Given $t \in T$, as a shorthand we write $read(t) \doteq \{v \in V \mid v \in read(guard(t))\}$, and analogously $write(t)$. Moreover, we assume that a DPN is always associated with an arbitrary initial marking M_I and an arbitrary final marking M_F. When M_F is reached the execution of the process instance ends.

Example 3.3. Consider as en example the DPN in Figure 1. From the initial marking $M_I = \{p_0\}$ a transition t_1 updates the value of $\mathtt{a}$ to any integer greater of $\mathtt{5}$. Then, t_2 or t_3 may be executable depending on the current value assigned in t_1 being greater or smaller than $\mathtt{10}$. Similarly, t_4 can be executed only if the initial value of $\mathtt{b}$ is smaller than the current value of $\mathtt{a}$. The formal execution semantics of DPNs is given in Section 3.1, but one can easily verify that the only possible sequence of transition that reaches the final marking is t_1, t_2, t_4, as $\alpha_I(\mathtt{b}) = \mathtt{10}$. A simplistic analysis that disregards the possible SV assignments of variables at each step, and thus only considers the control-flow of the net, would instead erroneously conclude that there are no dead transitions and that it is always possible to reach the final marking avoiding deadlocks, i.e., that $\mathcal{N}$ is classically sound [9].

We introduce a slightly more complex process than the one modeled in Figure 1, which we use as running example (for this reason, the process is sometimes not reasonable for a real world scenario).

Example 3.4. The example, depicted in Figure 2, models a process in which a customer applies for a loan. It is a modification of the one in [21], and has evident issues. A credit request transition represents the activity of requesting a certain loan amount, hence writing the variable $reqd$ (with

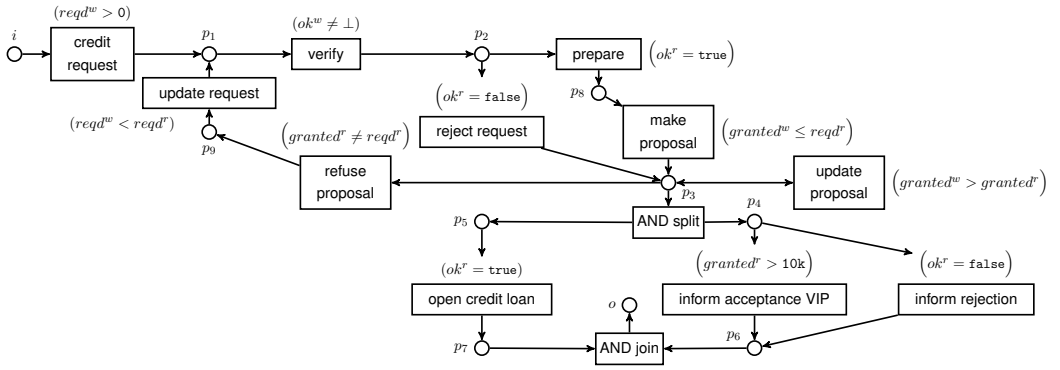

Figure 2. A modification of the example in [21]. $M_I = \{i\}$ and $M_F = \{o\}$ and $\alpha_I(reqd) = \alpha_I(granted) = 0$ and $\alpha_I(ok) = \bot$. The and-split and and-join transitions are considered to have some tautologically true guard.

domain $\mathcal{D}_{\mathbb{Z}}$). After this, a verify transition models the activity of performing some background checks, thus determining the value of a boolean variable ok. Depending on this outcome, two transitions can be fired: either prepare or reject request, so that a further transition make proposal can be only executed when the check was successful. This transition determines the amount that the bank is willing to lend, setting the integer variable $granted$ to a value that is smaller or equal to the requested amount. If the proposed amount is smaller than the requested amount, the transition refuse proposal can be fired. Then, by executing a further transition update request that updates $reqd$ to a smaller value than the current one, the procedure can be restarted. Alternatively, two parallel branches can be initiated: if the check represented by verify was successful, the transition open credit loan can be fired; at the same time, depending on the value of ok and $granted$, one of two transitions can be fired to notify the customer. These are inform acceptance VIP and inform rejection. A further case is possible: a transition update proposal can be fired to arbitrarily raise the proposed amount, even when the initial check determined a `false` value for variable ok.

This DPN is problematic: if verify sets ok to `false` then the and-join is never executable, and thus the final marking never reached. A similar case happens when $granted$ is smaller or equal to `10k`.

3.1. Execution Semantics

By considering the usual semantics for the underlying Petri net together with the guards associated to each of its transitions, we define the resulting execution semantics for DPNs in terms of possible states and possible evolutions from a state to the next. Let $\mathcal{N}$ as above be a DPN. Then the possibly infinite set of states of $\mathcal{N}$ is formed by all pairs (M, α) where $M \in \mathbb{B}(P)$[1], that is, M is the marking of the Petri net (P, T, F) and α is a SV assignment, defined as in the previous section.

In any state, zero or more transitions of a DPN may be able to fire. Firing a transition t updates the marking, reads the variables specified in $read(t)$ and selects a new value for those in $write(t)$. We

[1]The notation $\mathbb{B}(X)$ indicates the set of all multisets of elements of X. At times, we denote a marking M as a function such that, for each place p, $M(p)$ indicates the number of occurrences of p in M.

model this through a variable assignment β for the transition (cf. the previous section), which assigns a value to all and only those variables that are read or written. A pair (t, β) is called transition firing.

Definition 3.5. A DPN $\mathcal{N} = (P, T, F, V, dom, \alpha_I, guard)$ evolves from state (M, α) to state (M', α') through (t, β) with $guard(t) = c$, and we say that the transition firing is legal, iff:

- $\beta(v^r) = \alpha(v)$ if $v \in read(t)$: the variable assignment β assigns values as α for read variables;
- the new SV assignment α' is as α but updated as per β. It is computed as follows: for each $v \in V$, if $v \notin write(t)$ then the value of v is unchanged: $\alpha'(v) = \alpha(v)$; otherwise $\alpha'(v) = \beta(v^w)$;
- $c_{[\beta]} = \texttt{true}$: the guard is satisfied when we assign value to variables according to β;
- t is enabled and the new marking is M' (denoted $M[t\rangle M'$) according to the Petri net semantics.

We denote a legal transition firing by writing $(M, \alpha) \xrightarrow{t,\beta} (M', \alpha')$. We also extend this definition to sequences $\sigma = \langle (t^1, \beta^1), \ldots, (t^n, \beta^n) \rangle$ of n legal transition firings, called *traces*, and denote the corresponding run by $(M^0, \alpha^0) \xrightarrow{t^1,\beta^1} (M^1, \alpha^1) \xrightarrow{t^2,\beta^2} \ldots \xrightarrow{t^n,\beta^n} (M^n, \alpha^n)$ or equivalently by $(M^0, \alpha^0) \xrightarrow{\sigma} (M^n, \alpha^n)$. By restricting to the initial marking M_I of a DPN $\mathcal{N}$ together with the initial variable assignment α_I, we define the *runs of* $\mathcal{N}$ and *traces of* $\mathcal{N}$ as the set of runs and traces as above, of any length, such that $(M_I, \alpha_I) \xrightarrow{\sigma} (M, \alpha)$ for some marking M and SV assignment α.

For instance, referring to the simple DPN $\mathcal{N}$ in Figure 1, a possible run from of the initial state is $(\{p_0\}, \{\alpha_I(\texttt{a}) = 0, \alpha_I(\texttt{b}) = 10)\}) \xrightarrow{t_1,\{\beta(\texttt{a}^w)=7\}} (\{p_1\}, \{\alpha(\texttt{a}) = 7, \alpha(\texttt{b}) = 10)\})$.

Finally, recall that a Petri net (P, T, F) is unbounded when there exists a place $p \in P$ such that there exists no finite bound k so that $M(p) \leq k$ for all reachable markings M. The notion trivially extends to DPNs: a DPN is unbounded when there exists a place $p \in P$ so that there is no finite bound k such that $M(p) \leq k$ for all reachable states (M, α).

3.2. Data-aware soundness

We recall here the lifting of the standard notion of soundness [9] to the the data-aware setting of DPNs, as illustrated in [21]. The resulting notion is data-aware, as it requires not only to quantify over the reachable markings of the net, but also on the SV assignments for its case variables.

Given a DPN $\mathcal{N}$, in what follows we write $(M, \alpha) \xrightarrow{*} (M', \alpha')$ to mean that there exists a trace σ such that $(M, \alpha) \xrightarrow{\sigma} (M', \alpha')$ or that $(M, \alpha) = (M', \alpha')$. Also, given two markings M' and M'' of a DPN $\mathcal{N}$, we write $M'' \geq M'$ iff for all $p \in P$ of $\mathcal{N}$ we have $M''(p) \geq M'(p)$, and we write $M'' > M'$ iff $M'' \geq M'$ and there exists $p \in P$ s.t. $M''(p) > M'(p)$.

Definition 3.6. A DPN with initial marking M_I and final marking M_F is data-aware sound iff all the following properties hold. By denoting as $Reach_{\mathcal{N}}$ the set of reachable states of $\mathcal{N}$, namely the set $\{(M, \alpha) \mid (M_I, \alpha_I) \xrightarrow{*} (M, \alpha)\}$, these are:

P1. $\forall (M, \alpha) \in Reach_{\mathcal{N}}.\ \exists \alpha'.\ (M, \alpha) \xrightarrow{*} (M_F, \alpha')$

P2. $\forall (M, \alpha) \in Reach_{\mathcal{N}}.\ M \geq M_F \Rightarrow (M = M_F)$

P3. $\forall t \in T.\ \exists M_1, M_2, \alpha_1, \alpha_2, \beta.\ (M_1, \alpha_1) \in Reach_{\mathcal{N}}$ and $(M_1, \alpha_1) \xrightarrow{t,\beta} (M_2, \alpha_2)$

The first condition imposes, from any reachable state, that a state in which the first component is M_F is reachable. This corresponds to requiring that it is *always* possible to reach the final marking

by suitably choosing a continuation of the current run (i.e., legal transitions firings). The second condition captures that an output state is always reached in a "clean" way, i.e., without having tokens in the rest of the net. The third condition verifies the absence of dead transitions, where a transition is considered dead if there is no way of assigning the case variables, through the execution of the process, so as to eventually enable it. If only $P1$ and $P2$ hold for $\mathcal{N}$, but not $P3$, then the DPN is said to be data-aware weak sound. For instance, $P1$ is false for the DPN in Figure 1: when transition t_1 assigns a value not greater than `10` to `a` there exists no run from there which marks M_F.

4. Checking data-aware soundness of DPNs

In this section we show how to extend the general technique employed in [20] to our setting, namely to the case of more complex variable-to-variable conditions as in Definition 3.1. We show how the (possibly infinite) traces of the DPN, as defined in Section 3.1, can be abstracted through a special kind of state-transition structure which we call *constraint graph*. A constraint graph can be regarded as a finite *abstraction* of the original process with respect to soundness, as it isolates the source of infiniteness into the representation of the control-flow dimension of the original DPN, while abstracting the data that is manipulated by the process into a finite representation. As we are going to show, this implies that when the control-flow dimension of a DPN can be determined to be finite (that is, when the net is bounded), then the constraint graph is finite-state and it can thus be effectively analyzed to assess the data-aware soundness of the original process.

Given a constraint set C, we now define the procedure of computing the new constraint set C' resulting from the addition of a constraint c to C so that C' is uniquely determined, denoted $C' = C \oplus c$. This is shown in Algorithm 1, where we maintain the same notation as before, so that x, y, z can be either constants or read variables in V^r. It requires a `saturate` procedure that, given a set C' of constraints as input, returns the constraints in C' in addition to all the constraints logically implied by C' (using only variables and constants appearing in C', and assuming a lexicographic ordering that allows this operation to be deterministic). From Def. 3.1 it trivially follows that only finitely many constraints can be added with this procedure. When given an unsatisfiable constraint set, we assume `saturate` to return the same set as output. Hereafter, we assume a canonical ordering of variables and, hence, constraints. This allows us to efficiently compute the equivalence of two sets of constraints.

Referring to the algorithm, line 1 applies to constraints c where this imposes a condition of the current value of variables, i.e., with $write(c) = \emptyset$. In this case, the algorithm return the constraint set obtained by adding c to C. Line 2 applies to the case in which $write(c) \neq \emptyset$: the constraint c is added (line 3) whereas all previous constraints mentioning v are removed (line 5). After this, at lines 6-8 we replace each constraint of the form $(v^w \odot z)$ with one of the form $(v^r \odot z)$, expressing the fact that the new values assigned to variables is now taken as their current value. The test at line 8 makes sure that no constraint on the form $(v^r \odot v^r)$ is included in the set (this may otherwise happen for guards of the form $(v^w \odot v^r)$).

Let $\mathit{Const}_{\mathcal{N}}$ be the set of constants that appear in $\mathcal{N} = (P, T, F, V, dom, \alpha_I, guard)$. Formally, $\mathit{Const}_{\mathcal{N}} \doteq \{k \mid t \in T \wedge (v \odot k) = guard(t)\} \cup \{k \mid v \in V \wedge \alpha_I(v) = k\}$. Given a DPN $\mathcal{N}$, we denote by $\mathcal{C}_{\mathcal{N}}$ the set of all constraints $c \in \mathcal{C}_V$ satisfying one of the following two conditions: either $c = (v_1 \odot v_2)$ or, if $c = (v \odot k)$, that is k is a constant, then $k \in \mathit{Const}_{\mathcal{N}}$. Intuitively, $\mathcal{C}_{\mathcal{N}}$ is the set

Algorithm 1: Procedure for computing $C' \doteq C \oplus c$

```
1 if c = (v^r ⊙ x) then C' ← C ∪ {(v ⊙ x)}
2 if c = (v^w ⊙ x) then
3 |   C' ← saturate(C ∪ {(v^w ⊙ x)})
4 |   foreach c = (v^r ⊙ y) or c = (y ⊙ v^r) in C' do
5 |   |   C' ← C' \ {c}
6 |   foreach (v^w ⊙ z) in C' do
7 |   |   C' ← C' \ {(v^w ⊙ z)}
8 |   |   if z ≠ v^r then C' ← C' ∪ {(v^r ⊙ z)}
9 return saturate(C')
```

of all possible constraints that can be obtained by using only variables and constants that are in $\mathcal{N}$. Clearly, $\mathcal{C}_\mathcal{N}$ is finite.

Proposition 4.1. Given a DPN $\mathcal{N}$, if $C \cup \{c\} \subseteq \mathcal{C}_\mathcal{N}$ then $C' = C \oplus c$ is so that $C' \subseteq \mathcal{C}_\mathcal{N}$.

Since our objective is to verify only data-aware soundness, we define a *parsimonious* construction which attempts to minimize the number of states and transitions in the constraint graph. For this reason, we call such approach *lazy*, as opposed to the *eager* one in Section 5, when we will look at properties beyond data-aware soundness.

We consider a set of extra transition symbols τ_t, for $t \in T$. Each τ_t denotes the *silent* transition that corresponds to the explicit case-based reasoning hypothesis of assuming that $guard(t)$ does not hold in the current state. Given a set $E \subseteq T$, we define $\tau_E \doteq \{\tau_t \mid t \in E\}$.

Definition 4.2. Let $\mathcal{N} = (P, T, F, V, dom, \alpha_I, guard)$ be a DPN. Let $\mathcal{M}$ be the set of markings of $\mathcal{N}$, and M_I the initial marking. the lazy constraint graph $CG^{lazy}_\mathcal{N}$ of $\mathcal{N}$ is a tuple $\langle N, n_0, A\rangle$ with:

- $N \subseteq \mathcal{M} \times 2^{\mathcal{C}_\mathcal{N}}$ is a set of states of the graph, which we call *nodes* to distinguish them from the notion of states of the DPN. We equivalently use the notation n or (M, C) for its members, depending on the case; $n_0 = (M_I, C_0)$ is the initial node, with $C_0 = \bigcup_{v \in V} \{v = \alpha_I(v)\}$;
- $A \subset N \times (T \cup \tau_T) \times N$ is the set of arcs, which is defined with N by mutual induction:
  - a transition $((M, C), t, (M', C'))$ is in A iff
    - (i) $M[t\rangle M'$;
    - (ii) $C' = C \oplus guard(t)$ is satisfiable.
  - a transition $((M, C), \tau_t, (M, C''))$ is in A iff:
    - (i) $write(t) = \emptyset$;
    - (ii) $\exists M'$ s.t. $M[t\rangle M'$;
    - (iii) $C'' = C \oplus \neg guard(t)$ is satisfiable.

As define above, the set of nodes is the set of all possible couples in which the first component is a marking on the DPN $\mathcal{N}$ and the second is a constraint set on variables V and constants in $Const_\mathcal{N}$. The initial node is identified by the initial marking and the constraint set which simply encodes the initial SV assignment of the case variables. The transition relation between nodes (i.e., the arcs) admits two kinds of transitions. First, given a node (M, C), a new node (M', C') can be reached through a transition $t \in T$ of the DPN iff $((M, C), t, (M', C')) \in A$, which analogously to DPNs we denote as

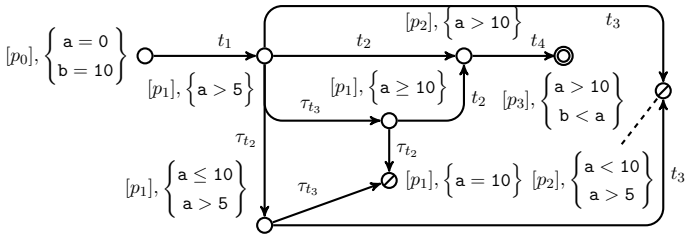

Figure 3. The lazy constraint graph $CG^{lazy}_{\mathcal{N}}$ for the DPN $\mathcal{N}$ in Figure 1. We list next to each node the couple (M, C), where a marking M is denoted as a set (since the DPN is 1-bounded). The constraint $(\mathtt{b} = 10)$ is not repeated in each node after the initial one, for brevity. We highlight nodes with the final markings by a double circle, and nodes that are dead-ends as forbidden signs. As anticipated, the DPN is clearly not data-aware sound.

$(M, C) \xrightarrow{t} (M', C')$. The conditions are as follows: (i) M' is the marking resulting from firing t in M according to the underlying Petri net semantics, and (ii) the constraint set C' obtained by adding the guard of t to the current set C, as defined by Algorithm 1, is satisfiable. Second, given a node (M, C), a new node (M', C') can be also reached through a silent transition τ_t iff $((M, C), \tau_t, (M', C')) \in A$, denoted $(M, C) \xrightarrow{\tau_t} (M', C')$. This case simulates the reasoning by case that is required to take into consideration every possible current SV assignment that does not enable a transition t. The conditions require that: (i) the transition t is only reading variables (since we are considering variables previously written); (ii) t is enabled in marking M of the DPN; (iii) the constraint set C'' obtained by adding the negation of the guard of t is satisfiable. We extend to constraint graphs the notions of trace and run, as well as that of reachability. Hence we say that a node (M, C) is reachable in $CG^{lazy}_{\mathcal{N}}$ iff $\{(M, C) \mid (M_I, C_0) \xrightarrow{*} (M, C)\}$. A simple example of lazy constraint graph $CG^{lazy}_{\mathcal{N}}$ for the DPN $\mathcal{N}$ in Fig. 1 is shown in Fig. 3. The DPN is not data-aware sound, since a can be assigned a value smaller or equal to 10. This corresponds, in $CG^{lazy}_{\mathcal{N}}$, to the nodes with no outgoing arcs.

Observation. This observation follows immediately by construction, and plays an important role in our approach: since the size of $\mathcal{C}_{\mathcal{N}}$ is finite, the state-space of $CG^{lazy}_{\mathcal{N}}$ is finite if and only if the DPN $\mathcal{N}$ is bounded, as the number of reachable markings is finite in this case.

Example 4.3. Consider the constraint graph $CG^{lazy}_{\mathcal{N}}$ for the DPN $\mathcal{N}$ as in Figure 2, which is partially depicted in Figure 4. The initial node is $(\{i\}, \{reqd = 0, ok = \bot, granted = 0\})$. As depicted, runs exist which reach nodes that have no outgoing transitions and thus cannot be extended to reach a node with final marking. For instance, consider the run in which the transition make proposal writes a value for $granted$ that is smaller or equal to 10k while ok is true: in this case, place p_6 can never be marked. Analogously, when ok is assigned value false, then the current run cannot be extended so that place p_7 is marked, and thus an output marking cannot be reached. Therefore, one should be able to conclude that $\mathcal{N}$ is not data-aware sound. In the next section we formalise this intuition. Note how the firing of transition refuse proposal induces a constraint set in which $granted \leq reqd$ is replaced by $granted < reqd$ in the resulting node (as the result of adding the guard of the transition, namely $granted \neq reqd$, to the current constraint set).

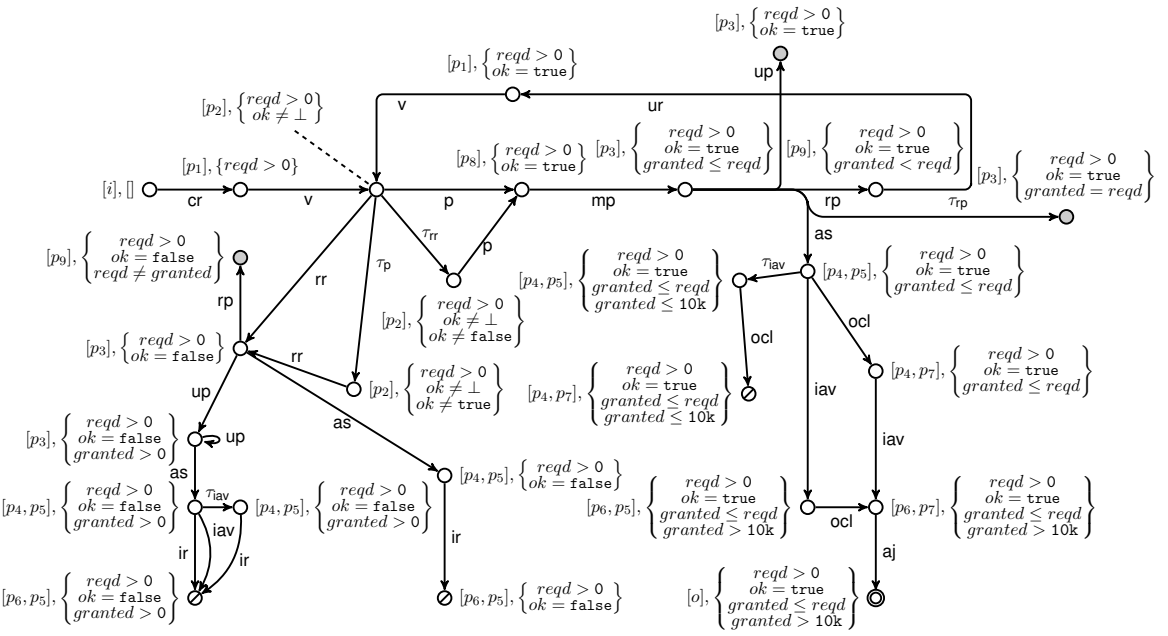

Figure 4. A fragment of the constraint graph $CG^{lazy}_{\mathcal{N}}$ for the DPN $\mathcal{N}$ in Figure 2, built by following Algorithm 2 (grey nodes are not expanded further for lack of space). Arcs are labelled with the initials of the transition names of the DPN, and constraints that assign variables to 0 or $\perp$ (which is the case for all variables in the initial node) are not shown. Note that only nodes with consistent constraint sets are generated.

4.1. Checking data-aware soundness on the lazy constraint graph

We now prove that $CG^{lazy}_{\mathcal{N}}$ suitably abstracts all possible runs of the given DPN $\mathcal{N}$, which allows us to check the data-aware soundness of $\mathcal{N}$ on $CG^{lazy}_{\mathcal{N}}$. First, for comparing the runs of $CG^{lazy}_{\mathcal{N}}$ to those of $\mathcal{N}$, it is convenient to resort to the notion of reachability graph of a DPN.

Definition 4.4. Given a DPN $\mathcal{N}$, the reachability graph of $\mathcal{N}$ is a graph $\langle S, E\rangle$ where:

- $S = Reach_{\mathcal{N}}$ is the set of reachable states of $\mathcal{N}$, each of the form (M, α); and
- $E \subseteq S \times T \times S$ is the set of arcs s.t. there exists an arc $s \xrightarrow{t,\beta} s'$ in $RG_{\mathcal{N}}$ iff $s \xrightarrow{t,\beta} s'$ for $\mathcal{N}$.

We extend to reachability graphs the notions of trace and run, as well as that of reachability. Hence we say that a state (M, α) is reachable in $RG_{\mathcal{N}}$ iff $(M, \alpha) \in S$.

We now define a simulation notion between $CG^{lazy}_{\mathcal{N}}$ and $RG_{\mathcal{N}}$, capturing a formal relationship between their runs. First, considering a special empty symbol ϵ for transitions, we define the observation function $O : T \cup \tau_T \to T \cup \{\epsilon\}$ so that $O(t) \doteq t$ if $t \in T$ and $O(t) \doteq \epsilon$ otherwise (i.e., $t \in \tau_T$). Given a trace $\sigma = t^1 \cdots t^n$ of a constraint graph as above, the corresponding *observed trace* is the sequence $O(\sigma) \doteq O(t^1) \cdots O(t^n)$. E.g., for $\sigma = t \cdot \tau_{t'} \cdot t''$ ($\cdot$ is used to denote concatenation) we have $O(\sigma) = t \cdot t''$. A trace σ is said to be *one-step* iff $O(\sigma)$ is not equal to ϵ and it has length one: it is composed by an arbitrary number of silent transitions and exactly one transition in T.

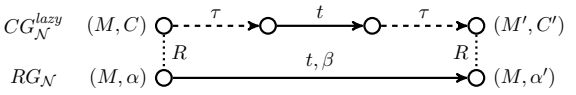

Figure 5. Intuitive depiction of the definition of the O-simulation relation R.

Definition 4.5. Given a reachability graph $RG_{\mathcal{N}} = \langle S, E\rangle$ of a DPN $\mathcal{N}$ and its constraint graph $CG^{lazy}_{\mathcal{N}} = \langle N, n_0, A\rangle$, we say that a relation $R \subseteq N \times S$ is a O-simulation relation of $RG_{\mathcal{N}}$ by $CG^{lazy}_{\mathcal{N}}$ iff $(n,(M,\alpha)) \in R$ implies that for any $(M,\alpha) \xrightarrow{t,\beta} (M',\alpha')$ there exists a one-step trace σ with $O(\sigma) = t$ and $n \xrightarrow{\sigma} n'$, so that $(n',(M',\alpha')) \in R$.

A node of $CG^{lazy}_{\mathcal{N}}$ O-simulates a state in $RG_{\mathcal{N}}$ iff there exists a O-simulation relation R of $RG_{\mathcal{N}}$ by $CG^{lazy}_{\mathcal{N}}$ such that these are included in the relation, and we say that $CG^{lazy}_{\mathcal{N}}$ O-simulates $RG_{\mathcal{N}}$ if n_0 O-simulates (M_I,α_I). An intuitive depiction of the definition above is given in Figure 5.

Lemma 4.6. $CG^{lazy}_{\mathcal{N}}$ O-simulates $RG_{\mathcal{N}}$.

Proof. ([21]) We show this by induction. First, note that the marking reached by executing a sequence of observable transitions is the same, in both the reachability graph (hence the DPN) and the constraint graph (irrespective of the unobservable transitions added). This is because the variable assignment β^i considered at each step plays no role in computing the next marking of the DPN. Hence in what follows we use the same symbol M for the new marking of both $RG_{\mathcal{N}}$ and $CG^{lazy}_{\mathcal{N}}$. To prove the claim, we first need to make sure that the same set of transitions $t \in T$ are enabled in both (M_I,α_I) and (M_I,C_0), i.e., the initial state of $RG_{\mathcal{N}}$ and initial node of $CG^{lazy}_{\mathcal{N}}$. This is true by construction because $C_0 \doteq \bigcup_{v\in V}\{(v = \alpha_I(v))\}$: by defining $\mathit{enabled}((M,\alpha)) \doteq \{t \mid (M,\alpha) \xrightarrow{t,\beta} (M',\alpha') \text{ for some } M',\alpha',\beta\}$, and $\mathit{enabled}((M)) \doteq \{t \mid \exists M'.\ M[t\rangle M'\}$, we thus have $\mathit{enabled}((M_I,\alpha_I)) = \mathit{enabled}((M_I,C_0))$. Then, if $(M_I,\alpha_I) \xrightarrow{t,\beta} (M,\alpha)$ there must exist a trace σ with $O(\sigma) = t$ such that $(M_I,C_0) \xrightarrow{\sigma} (M,C)$ is in $CG^{lazy}_{\mathcal{N}}$, and so that (M,C) O-simulates (M,α). In particular, this is true for $\sigma = t \cdot \tau_E$ with $E = \{t' \in T \mid t' \neq t \wedge t' \in \mathit{enabled}(M) \wedge t' \notin \mathit{enabled}((M,\alpha)) \wedge \mathit{write}(t') = \emptyset\}$. The idea is that $\sigma = t \cdot \tau_E$ corresponds to the execution of t in the constraint graph, followed by an unobservable transition for each transition $t' \in \mathit{enabled}(M)$ which cannot be fired from the new state (M,α) of the reachability graph due to the SV assignment α. Such σ makes sure that the new generated constraint set C correctly encodes the initial set C_0 updated with the negated guard of each transition that cannot be fired next in $RG_{\mathcal{N}}$. Notice that, by construction, a run with trace σ always exists in $CG^{lazy}_{\mathcal{N}}$: unobservable transitions τ_t are always added for any t with $\mathit{write}(t) = \emptyset$ as long as the resulting constraint set is satisfiable (cf. Definition 4.2). Since $O(\sigma) = t$, it remains to show that (M,C) O-simulates (M,α). This can be done by repeating in the inductive step the same reasoning as above, since the marking is the same (hence the set of transitions in $\mathit{enabled}(M)$) and it also holds that $\mathit{enabled}((M,\alpha)) \subseteq \mathit{enabled}((M,C))$. □

This result implies that the lazy constraint graph can step-wise "mimic" any possible run of the reachability graph (hence of DPN). This, however, does not imply that *any* property true in $RG_{\mathcal{N}}$ is

also true in $CG^{lazy}_{\mathcal{N}}$, or vice-versa, as we will comment further in Section 5.

We now adapt the definition of data-aware soundness to constraint graphs, as our objective is to assess these properties on these objects. The three properties $P1$-$P3$ as in Definition 3.6 become as follows. By denoting as $Reach(CG^{lazy}_{\mathcal{N}})$ the set of reachable nodes of $CG^{lazy}_{\mathcal{N}}$, namely the set $\{(M, C) \mid (M_I, C_0) \xrightarrow{*} (M, C)\}$ we then have:

P1. $\forall (M, C) \in Reach(CG^{lazy}_{\mathcal{N}}).\ \exists C'.(M, C) \xrightarrow{*} (M_F, C')$;
P2. $\forall (M, C) \in Reach(CG^{lazy}_{\mathcal{N}}).\ M \geq M_F \Rightarrow (M = M_F)$;
P3. $\forall t \in T.\ \exists M, M', C, C'.\ (M, C) \in Reach(CG^{lazy}_{\mathcal{N}})$ and $(M, C) \xrightarrow{t} (M', C')$.

A constraint graph $CG^{lazy}_{\mathcal{N}}$ of a DPN $\mathcal{N}$ is said to be data-aware sound iff these properties are true. With this definition at hand, we can now state our main result: intuitively, that the lazy constraint graph faithfully abstracts the original DPN *with respect to* data-aware soundness.

Theorem 4.7. $RG_{\mathcal{N}}$ is data-aware sound iff $CG^{lazy}_{\mathcal{N}}$ is data-aware sound.

Proof. ([21]) Consider property $P1$, which requires the a state (M_F, α) on $RG_{\mathcal{N}}$ (resp., node (M_F, C) on $CG^{lazy}_{\mathcal{N}}$) is always reachable from any other state (resp., node).

($\Rightarrow$) This direction follows by O-simulation. Assume to fix a state (M, α) reached by executing a trace σ', and for which the property must hold if $RG_{\mathcal{N}}$ is data-aware sound: there exists a trace σ such that $(M, \alpha) \xrightarrow{\sigma} (M_F, \alpha')$ for some α'. Then at least one node (M, C) exists that is reached in $RG_{\mathcal{N}}$ through a run with trace σ', for which (M, C) O-simulates (M, α) – in the base case, these are the initial state and node, respectively. Then by Lemma 4.6 there must also exist a run $(M, C) \xrightarrow{\sigma''} (M_F, C')$, for some C', with $\sigma'' = O(\sigma)$ and which only traverses nodes that (stepwise, by considering one-step traces) O-simulate those in σ. If instead P1 is not true for $RG_{\mathcal{N}}$ but holds for $CG^{lazy}_{\mathcal{N}}$ then either $RG_{\mathcal{N}}$ has additional runs which do not correspond to runs of $CG^{lazy}_{\mathcal{N}}$ (and which cannot be extended to reach a final node) or that there exists in $CG^{lazy}_{\mathcal{N}}$ at least one run, with trace σ, such that $O(\sigma)$ is not a trace of $RG_{\mathcal{N}}$. The first case is not possible due to the lemma, whereas the latter is excluded by construction (Def. 4.2) since each possible $t \in T$ is considered when building $CG^{lazy}_{\mathcal{N}}$

($\Leftarrow$) First, consider a run $s \xrightarrow{\sigma'} s'$ in $CG^{lazy}_{\mathcal{N}}$ where σ' is not required to be a one-step trace. Observe that if one such run exists which is cyclic, then also an acyclic one ending in s' must exist, as cycles do not affect reachability of nodes (a run is cyclic iff the same node appears twice). Therefore in what follows we are allowed to restrict to acyclic runs of $CG^{lazy}_{\mathcal{N}}$. To satisfy the requirement $P1$ it is indeed enough to find an acyclic run in $CG^{lazy}_{\mathcal{N}}$ which reaches a node with final marking that also includes a given node s, for every reachable s. Hence we show that by construction every acyclic run ρ on $CG^{lazy}_{\mathcal{N}}$ with trace σ has a corresponding run ρ' in $RG_{\mathcal{N}}$ with trace σ', and in particular one such that $\sigma' = O(\sigma)$, so that for every $\rho \cdot \rho''$ there exists $\rho' \cdot \rho'''$ in $RG_{\mathcal{N}}$ with the same observed trace. For the base case we have (M_I, C_0) and (M_I, α_I), with $C_0 \doteq \bigcup_{v \in V} \{v = \alpha_I(v)\}$, and (M_I, α_I) and (M_I, C_0) enabled the same transitions. Then, consider $(M, C) \xrightarrow{\sigma} (M', C')$ to be one-step, with (M, C) O-simulating (M, α). A transition $(M, \alpha) \xrightarrow{t,\beta} (M', \alpha')$ with $O(\sigma) = t$ so that $(M'C')$ O-simulates (M', α') must exist by construction, as a transition exists in $CG^{lazy}_{\mathcal{N}}$ iff it is enabled in the current marking in $\mathcal{N}$, $guard(t)$ is satisfied in C, $M[t\rangle M'$ and C' is consistent. Since C' is consistent,

Algorithm 2: Data-aware soundness-checking procedure

Input: A DPN $\mathcal{N} = (P, T, F, V, dom, \alpha_I, guard)$, with initial marking M_I and final marking M_F.

1 $C_0 \leftarrow \bigcup_{v \in V} \{v = \alpha_I(v)\}$, $s_0 \leftarrow \langle M_I, C_0 \rangle$, $S \leftarrow \{s_0\}$, $A \leftarrow \emptyset$, $L \leftarrow \{s_0\}$
2 **while** $L \neq \emptyset$ **do**
3 $(M, C) \leftarrow \texttt{pick}(L)$
4 $L \leftarrow L \setminus \{(M, C)\}$
5 **foreach** $t \in T$ *s.t.* $M \xrightarrow{t} M'$ **do**
6 $C' \leftarrow C \oplus guard(t)$
7 $C'' \leftarrow C$
8 **if** $write(t) = \emptyset$ **then** $C'' \leftarrow C'' \oplus \neg guard(t)$
9 **if** $\texttt{satisfiable}(C')$ **then**
10 **if** $\exists (\bar{M}, C') \in S$ s.t. $M' > \bar{M} \wedge (\bar{M}, C') \xrightarrow{*} (M', C')$
11 **then return** `false` //$\mathcal{N}$ is unbounded (see Proposition 4.8)
12 $N \leftarrow N \cup \{(M', C')\}$
13 $A \leftarrow A \cup \{\langle (M, C), t, (M', C') \rangle\}$
14 $L \leftarrow L \cup \{(M', C')\}$
15 **if** $\texttt{satisfiable}(C'') \wedge C \neq C''$ **then**
16 $N \leftarrow N \cup \{(M, C'')\}$
17 $A \leftarrow A \cup \{\langle (M, C), \tau_t, (M, C'') \rangle\}$
18 $L \leftarrow L \cup \{(M, C'')\}$
19 **return** $\texttt{analyzeConstraintGraph}(\langle N, n_0, A \rangle, M_F)$

a solution α' exists. If the property is not true for $CG^{lazy}_{\mathcal{N}}$, then there exists a run ρ reaching a node (M, C) from where a node with final marking cannot be reached, i.e., there is no run ρ' from (M, C) reaching such nodes. By construction, if σ is the trace of the run $\rho \cdot \rho'$, then $O(\sigma)$ is not a trace of $\mathcal{N}$.

For $P2$ and $P3$ we follow the same reasoning as for $P1$: they all require the existence of runs from states in $RG_{\mathcal{N}}$ (and nodes in $CG^{lazy}_{\mathcal{N}}$), that are reached through corresponding traces σ' and σ, respectively, so that $\sigma' = O(\sigma)$. □

4.2. A computational procedure for checking data-aware soundness of DPNs

Definition 4.2 specifies all the conditions that are required in order to build the transitions between the nodes of a constraint graph, starting from the initial node. As such, it can be used to devise a *procedure* for checking data aware soundness, which is shown in Algorithm 2. Given a DPN $\mathcal{N}$, the procedure computes the complete constraint graph $CG^{lazy}_{\mathcal{N}}$ of $\mathcal{N}$ when this is bounded, then invokes an explicit soundness checking procedure on it. This procedure takes in input $CG^{lazy}_{\mathcal{N}}$ as well as the final marking of $\mathcal{N}$. If instead, during the computation, the DPN is determined to be unbounded (see the condition at line 11) then a negative result is returned, as this implies that $\mathcal{N}$ is in fact unbounded and thus cannot be data-aware sound. First, we prove that the condition checked by the procedure in fact detects when the given DPN $\mathcal{N}$ is unbounded, and that this implies that $\mathcal{N}$ is not data-aware sound; then we comment the steps in the procedure.

Proposition 4.8. Given a constraint graph $CG^{lazy}_{\mathcal{N}}$ for a DPN $\mathcal{N}$, if there exist two nodes (M, C) and (M', C) in the constraint graph so that $M > M'$ and $(M', C) \xrightarrow{*} (M, C)$ then $\mathcal{N}$ is unbounded. If $\mathcal{N}$

is unbounded then it is also not data-aware sound.

Proof. Assume (M, C) and (M', C) exist in $CG^{lazy}_{\mathcal{N}}$ so that $M > M'$ and $(M', C) \xrightarrow{\sigma} (M, C)$. We show that no bound exists on the number of tokens in the markings M of reachable nodes of $CG^{lazy}_{\mathcal{N}}$. Indeed, it follows that a node (M'', C) exists in $CG^{lazy}_{\mathcal{N}}$ with $M'' > M$ and $(M, C) \xrightarrow{\sigma} (M'', C)$, or otherwise this would contradict that $(M', C) \xrightarrow{\sigma} (M, C)$. This implies that the state-space of $CG^{lazy}_{\mathcal{N}}$ is infinite. Since the number of possible constraints is finite, by Prop. 4.1 we conclude that the state-space of $CG^{lazy}_{\mathcal{N}}$ can be unbounded only if $\mathcal{N}$ is unbounded. Assume now $\mathcal{N}$ is unbounded but data-aware sound, and let $M = M' \uplus \overline{M}$.[2]. If $\mathcal{N}$ is data-aware sound, $(M', \alpha) \xrightarrow{*} (M_F, \alpha')$ (Property P1 of Def. 3.6). However, this also means that $(M' \uplus \overline{M}, \alpha) \xrightarrow{*} (M_F \uplus \overline{M}, \alpha')$, which violates Property P2 of Def. 3.6. Therefore $\mathcal{N}$ is not data-aware sound. □

We can now comment the body of the procedure. The algorithm uses a set L to hold the nodes of the constraint graph being built that still need to be expanded. As new nodes of the form (M, C) are generated, these are added to L and are removed only when all transitions t such that $M[t\rangle M'$ is in $\mathcal{N}$, for some M', have been considered in the for-each loop at line 5. N and A are, respectively, the nodes and the arcs of the constraint graph returned by the algorithm. An arc labelled with t is built for every $t \in T$ enabled in the current marking of the DPN $\mathcal{N}$, and at the beginning of the foreach cycle at line 5 the constraint set C' is obtained by adding to C the guard of t (and saturating). Then, consistently with the definition, if the guard of t is a test on the current value of variables, then C'' is computed by adding to C the negated guard, to explicitly represent those SV assignments in which the guard is not satisfied. However, if the net is found to be unbounded then `false` is returned (line 11). If this is not the case, an edge is created for the silent transition τ_t. The node reached by this edge is such that the marking is not updated (as no transition in the DPN was fired), whereas the new constraint set is the C'' computed as above. The algorithms then continues considering a new transition.

The following theorem states the soundness of the approach.

Theorem 4.9. Algorithm 2 terminates and is correct, that is, it returns `true` iff $\mathcal{N}$ is data-aware sound.

The algorithm always terminates: as already observed, when the net is bounded then $CG^{lazy}_{\mathcal{N}}$ is finite-state; if instead the net is unbounded then this will be eventually detected by the algorithm, which will then terminate and simply return `false`.

Regarding the complexity of the procedure, the state-space of $CG^{lazy}_{\mathcal{N}}$ is bounded by $|\mathcal{M}| \times 2^{|\mathcal{C}_{\mathcal{N}}|}$ in the bounded case, otherwise the procedure is guaranteed to terminate (with a negative answer) with the same worst-case complexity of boundedness-checking for standard Petri nets (which is EXSPACE-complete) [32]. The second component of the state space, i.e. the number of possible constraint sets, is at most exponential in the number of variables, operators in the domains and constants in $Const_{\mathcal{N}}$. Verifying properties P1-P3 as in Def. 3.6 can be done by executing a search procedures on this finite-state graph: from each node a final node must be reachable $(P1)$ and a transition firing for each $t \in T$ must be reachable from the initial node $(P3)$. $P2$ requires checking that there exists no node with $M > M_F$.

[2] We use $\uplus$ to indicate the multiset union, e.g. $[a, b] \uplus [b, c] = [a, b, b, c]$

As a consequence, if and only if the procedure `analyzeConstraintGraph` returns `true` for a given DPN $\mathcal{N}$ then we conclude that $\mathcal{N}$ is data-aware sound. We here do not present an algorithm for such procedure, as it only needs to apply the definition of data-aware soundness on $CG^{lazy}_{\mathcal{N}}$, which is finite-state. This can be implemented by exploiting verification or search techniques. Thanks to the results above, we have provided a practical and implementable procedure for checking data-aware soundness of DPNs.

5. Beyond data-aware soundness

In the previous section we have shown that a DPN $\mathcal{N}$ is data-aware sound if and only if the lazy constraint graph $CG^{lazy}_{\mathcal{N}}$ is so. We have shown that this result holds even though, as implied by Lemma 4.6, these structures are not *O-bisimilar* in general. Nonetheless, the notion of data-aware soundness does not depend on the specific *branching structure* of $\mathcal{N}$ and thus of $RG_{\mathcal{N}}$, and therefore inspecting $CG^{lazy}_{\mathcal{N}}$ is sufficient to verify this property. However, this clearly implies that any property which depends on the specific branching structure of a DPN (even when disregarding the specific values assigned to variable) cannot be verified on $CG^{lazy}_{\mathcal{N}}$. As anticipated, in this section we illustrate a technique that is based on a variant of constraint graph of a given DPN $\mathcal{N}$, assumed to be bounded. We call this variant of constraint graph *eager*, which we show to be a faithful abstraction of the given DPN $\mathcal{N}$. As expected, this implies that the size of $CG^{eager}_{\mathcal{N}}$ is in general larger than the size of $CG^{lazy}_{\mathcal{N}}$, although the new construction does not require the distinction between observable and unobservable transitions, which was central for $CG^{lazy}_{\mathcal{N}}$.

We consider at least two practical scenarios justifying the need for an abstraction that preserve the branching structure of DPNs. First, consider the case in which a DPN is determined not to be data-aware sound by the procedure `analyzeConstraintGraph`, as described in Section 4. As a simple negative verdict may not be satisfactory, can we check whether the process can be restricted in such a way that data-aware soundness can be achieved? As a further example, even when a DPN is data-aware sound, can we analyze its branching structure so as to determine whether the soundness property relies on the collaboration of all actors involved in the enactment of the process, or only part of them?

To answer these questions, in this section we consider the property of *controllable* data-aware soundness and provide a technique for verifying it on a given DPN. We need to impose two conditions:

- $\mathcal{N}$ is bounded. As suggested above, boundedness may be known as a result of the (negative) verdict of the soundness-checking approach in Section 4 (specifically, thanks to the boundedness test in Algorithm 2). If instead it is not known whether $\mathcal{N}$ is bounded, then we simply observe that the same boundedness test as in Algorithm 2 can be performed when explicitly constructing $CG^{eager}_{\mathcal{N}}$, although an analogous construction algorithm is not given here. In fact, Prop. 4.8 still holds when $CG^{lazy}_{\mathcal{N}}$ is replaced by $CG^{eager}_{\mathcal{N}}$, and this can be directly proved in the same way.
- The domains of variables are restricted to $\mathcal{D}_{\mathbb{R}}$ and $\mathcal{D}_{bool}$. Allowing other finite or *dense* domains (as well as $\mathcal{D}_{strings}$) is possible with no modification, but we keep the formalization simple.

5.1. Controllable data-aware soundness of bounded DPNs

We first give an intuition. Consider a process, modeled by a DPN, that describes the interaction of multiple actors. The bank process in Figure 2 is an example of such processes, as it describes the interaction of (at least) the customer and a bank clerk. In such scenarios, each of the actors (customer and clerk) has some control on the execution of the activities represented by DPN transitions. For instance, the transition credit request is clearly executed by the customer, whereas transitions such as deny credit or make proposal are executed by clerk, which has decisional authority on which transition is executed in each case (and if make proposal is chosen, the clerk will also decide the amount).

We thus can assign transitions to each actor, and say that each transition is *controlled* by exactly one actor. A *process execution strategy* for an actor AC is thus a policy that tells the actor which are the possible transitions to fire at each step, possibly specifying the values to write for variables (if these are under the control of the agent). If then we adopt the *perspective* of one such actors AC in a DPN $\mathcal{N}$, so that all the transitions that are not controlled by AC are assumed to be controlled by the *environment* (an abstract antagonist controlling everything else), we might be interested in checking whether the actor AC can make its choices, namely adopt a process execution strategy, so that the resulting sub-process, obtained by removing any branch of $RG_{\mathcal{N}}$ that is not allowed by the process execution strategy, is data-aware sound as in Definition 3.6. As suggested at the beginning of this section, a DPN may not be data-aware sound but an actor might still be able enforce such condition. If this is not the case, the same verification task can be carried out with respect to data-aware *weak* soundness (defined below Def. 3.6), i.e., allowing dead transitions.

In what follows, we formalize: (i) which are the sub-processes (henceforth called *refinements*) of $RG_{\mathcal{N}}$ that AC can enforce; (ii) how to link such refinements to process execution strategies for AC; (iii) say that the DPN $\mathcal{N}$ can be made data-aware (weak) sound by the actor AC when there exists a process execution strategy for AC that induces a refinement of $RG_{\mathcal{N}}$ that is data-aware (weak) sound. Note that we are only interested here in the *existence* of one such process execution strategy: the problem of automatically *synthesizing* process execution strategy is left as future work.

Formally, consider a subset $T_{\text{AC}} \subseteq T$ of transitions and a subset $V_{\text{AC}} \subseteq V$ of variables of a DPN $\mathcal{N}$ that are *under the control of* AC. Intuitively, only the actor AC can decide to execute transitions in T_{AC} and similarly it is the responsibility of actor AC to decide the new value of variables in V_{AC} whenever these are written (by transitions in T_{AC} as well as by other transitions controlled by the environment). To avoid conflicts, without loss of generality we impose one restriction on $\mathcal{N}$: all transitions enabled from any marking M (i.e. the set $\{t \mid \exists M'.\ M[t\rangle M'\}$) are controlled either by our actor AC or by the environment. We refer to the former case by saying that AC controls M, and similarly we say that AC controls a state $s = (M, \alpha)$ of the reachability graph $RG_{\mathcal{N}}$ iff AC controls M.

Definition 5.1. Given a DPN $\mathcal{N}$, the actor AC can enforce a set of transition firings X from a state s of $RG_{\mathcal{N}}$ iff one of the following conditions are satisfied:

- if AC controls s then for all $(t, \beta) \in X$ either $write(t) = \emptyset$ or $write(t) \subseteq V_{\text{AC}}$ or for every β' we have that $s \xrightarrow{t,\beta'} s''$ implies $(t, \beta') \in X$;
- if AC does not control s then for all $(t, \beta) \in X$ either:
  - $write(t) \neq \emptyset$ and $write(t) \subseteq V_{\text{AC}}$, or
  - for every β', $s \xrightarrow{t,\beta'} s''$ implies $(t, \beta') \in X$;

and there is no t' enabled in s (s.t. $s \xrightarrow{t',-} s'$) for which there is no β' so that $(t', \beta') \in X$.

Intuitively, this definition captures the conditions under which an actor can act in s so that, no matter how the environment behaves, the selected transition firing will be in X. Since transition firings are deterministic (i.e., the resulting state is uniquely determined), this implies that AC can force the next state of $RG_{\mathcal{N}}$ to be in a certain subset. In the first case of the definition above, when AC controls s, it is sufficient to check that the environment cannot force the next state s' to be outside X by choosing some value of written variable. An analogous condition captures the other case.

Example 5.2. Consider the DPN in Figure 2 and the state $(\{p_5, p_4\}, \alpha)$ where we have $\alpha(ok) =$ `true`, $\alpha(granted) =$ `8k` and $\alpha(reqd) =$ `9k`. If AC controls $\{p_5, p_4\}$, hence open credit loan, inform acceptance VIP and inform rejection, then AC can enforce each of the possible specific transition firings that is legal, since no variable is written (by the environment). If instead AC does not control the marking above, then it cannot enforce specific transition firings, since it will be the environment to decide the next marking. As a further example, consider $(\{p_1\}, \alpha)$ with $\alpha(ok) = \bot$, $\alpha(granted) =$ `0` and $\alpha(reqd) =$ `9k`. Even if AC controls verify this does not imply that AC can enforce a specific transition firing (e.g. verify with $\{ok \rightarrow$ `true`$\}$): in order to do so, it would also need to control variable ok.

We denote the set of all sets of transition firings that can be enforced by AC from a state s of $RG_{\mathcal{N}}$ as $Ctrl^s_{\text{AC}}$. Note that it is often possible that the set of transition firings that can be enforced by AC from a state s is not unique (i.e., $|Ctrl^s_{\text{AC}}| > 1$) but it is closed under union.

We now turn to define the executions of $\mathcal{N}$ that an actor can enforce. We model this as a process execution strategy for AC: a partial function $strat$ which, given the sequence of states $s_0, \ldots, s_n$ of $RG_{\mathcal{N}}$ traversed so far by the current run, either returns a set of legal transition firings that can be enforced by AC or it is undefined (if s_n has no possible legal successors). If we define $Ctrl^S_{\text{AC}}$ as $\{Ctrl^s_{\text{AC}} \mid s \in S\}$ then $strat : S^+ \mapsto Ctrl^S_{\text{AC}}$ is so that $strat(s_0, \ldots, s_n) \in Ctrl^{s_n}_{\text{AC}}$. In words, $strat$ always returns *a set* of possible transition firings that actor AC can control from the last state s_n in the given sequence of traversed states.

We denote by $RG^{strat}_{\mathcal{N}}$ the (refined) reachability graph obtained by executing $strat$ on $RG_{\mathcal{N}}$, that is, so that a run $s_0 \xrightarrow{t_1,\beta_1} \cdots \xrightarrow{t_n,\beta_n}$ of $RG_{\mathcal{N}}$ is in $RG^{strat}_{\mathcal{N}}$ iff $(t_{i+1}, \beta_{i+1}) \in strat(s_0, \ldots, s_i)$ for $i \in [0, n-1]$. Since multiple functions $strat$ may exist then multiple refinements are possible, but we do not need to actually compute them. They are needed to define our verification task:

Definition 5.3. Given a DPN $\mathcal{N}$ and sets $V_{\text{AC}} \subseteq V$ and $T_{\text{AC}} \subseteq T$ of variables and transitions (respectively) controlled by an actor AC, we say that AC *can guarantee data-aware (weak) soundness for* $\mathcal{N}$ iff there exists a process execution strategy $strat$ for AC so that $RG^{strat}_{\mathcal{N}}$ is data-aware (weak) sound.

Example 5.4. In our example in Figure 2 we might be interested in checking whether the actor AC representing the clerk can guarantee data-aware soundness, assuming $T_{\text{AC}} = \{$verify, reject request, prepare, make proposal$\}$ and $V_{\text{AC}} = \{granted\}$. It is easy to see that this actor cannot guarantee data-aware soundness nor data-aware weak soundness: since the actor does not control variable ok, when this variable is set to `false` then the marking $\{p_5, p_4\}$ is a deadlock. If we add ok to V_{AC} then

the previous deadlock can be avoided by always writing $ok = \texttt{true}$ with transition verify, but another deadlock is encountered: when $ok = \texttt{true}$ and $reqd$ is set to a value smaller or equal to 10k, all states with marking $\{p_5, p_4\}$ have no legal transition firings, and this marking cannot be avoided. If we add update proposal to T_{AC} then the resulting refinement of $RG_{\mathcal{N}}$ is data-aware weak sound: in addition to the above it is enough, from states with $ok = \texttt{true}$ and $reqd$ set to a value smaller or equal to 10k, to fire transition modify proposal to raise the proposed loan amount to a value higher than 10k. Still, this refinement is not data-aware sound: it will never fire some transitions, for instance reject request.

As already commented informally, the property above cannot be checked on the lazy constraint graph $CG^{lazy}_{\mathcal{N}}$ of a DPN $\mathcal{N}$, because it clearly depends on the specific branching structure of the reachability graph $RG_{\mathcal{N}}$ induced by $\mathcal{N}$ (although modulo the values of variables).

5.2. Eager constraint graph

Intuitively, an eager constraint graph $CG^{eager}_{\mathcal{N}}$ for a given DPN $\mathcal{N}$ differs from the lazy constraint graph $CG^{lazy}_{\mathcal{N}}$ of $\mathcal{N}$ in that it does not attempt to parsimoniously explore the state space, but explicitly encodes the possible successor states that are reachable by firing each of the enabled transitions. The construction is analogous, but instead of resorting on unobservable transitions, we finitely yet explicitly account for all possibilities when a transition is fired that updates a variable. As in the previous section, when $\mathcal{N}$ is bounded then this structure is finite-state. First we define a finite set of constraint sets that account for all possible ways in which a variable may be updated.

Recall that we denote by $\mathit{Const}_{\mathcal{N}}$ the set of constants that appear anywhere in $\mathcal{N}$, i.e., $\mathit{Const}_{\mathcal{N}} = \{k \mid t \in T \wedge (v \odot k) = \mathit{guard}(t)\} \cup \{k \mid v \in V \wedge \alpha_I(v) = k\}$. For instance, for the DPN in Figure 2 we have $\mathit{Const}_{\mathcal{N}} = \{0, \texttt{10k}\}$. Based on this, we define the set of intervals partitioning the domain of variables that are assigned domain $\mathcal{D}_{\mathbb{R}}$. Let $\mathit{Const}^{+}_{\mathcal{N}} \doteq \mathit{Const}_{\mathcal{N}} \cup \{+\infty, -\infty\}$.

Definition 5.5. The representative intervals for $v \in V$ is the set $Intervals^{v}_{\mathcal{N}}$ of constraint sets encoding all possible relationship between v and the constants in $\mathit{Const}_{\mathcal{N}}$. It is defined as follows:

- If v has domain $\mathcal{D}_{\mathbb{R}}$, it is the set containing $\{(v = k)\}$ for each $k \in \mathit{Const}_{\mathcal{N}}$, and $\{(v > k_1), (v < k_2)\}$ for any two distinct successive $k_1, k_2 \in \mathit{Const}^{+}_{\mathcal{N}}$;
- If v has domain $\mathcal{D}_{\mathit{bool}}$, it is trivially defined as $\{\{(v = \texttt{true})\}, \{(v = \texttt{false})\}\}$.

$Intervals^{v}_{\mathcal{N}}$ is unique for each $v \in V$, although one could optimize by analyzing the structure of $\mathcal{N}$ and avoid considering constraints that are not relevant. For the DPN in Figure 2, for both $v = reqd$ and $v = granted$, we have $Intervals^{v}_{\mathcal{N}} = \{\{(v < \texttt{0})\}, \{(v = \texttt{0})\}, \{(v > \texttt{0}, v < \texttt{10k})\}, \{(v = \texttt{10k})\}, \{(v > \texttt{10k})\}\}$. For ok, we have $Intervals^{ok}_{\mathcal{N}} = \{\{(ok = \texttt{true})\}, \{(ok = \texttt{false})\}\}$.

The sets defined above allow to finitely guess the possible (relevant intervals of) values assigned to written variables when a transition of the DPN is fired. However, such guesses are not yet sufficient for evaluating successive guards that compare variables between them, i.e. of the form $(v^r_1 \odot v^r_2)$. For this reason, we also consider the following notion.

Definition 5.6. The set of two-variable constraint sets of a $\mathcal{N}$ is the set $C^2_{\mathcal{N}}$ of all satisfiable constraint sets C in which, for each transition t for which $\mathit{guard}(t)$ is of the form $(v^r_1 \odot v^r_2)$, then either $(v^r_1 \odot v^r_2)$ or $\neg(v^r_1 \odot v^r_2)$ is in C.

For instance, for the DPN in Figure 2 we have $C^2_{\mathcal{N}}$ composed of only two sets, namely $\{(granted^r = reqd^r)\}$ and $\{(granted^r \neq reqd^r)\}$, because the only guard of the form $(v^r_1 \odot v^r_2)$ is the guard of transition refuse proposal.

Finally we can give the definition of eager constraint graph, as a structure that finitely abstracts all possible runs of $RG_{\mathcal{N}}$. As we are going to show, this structure also preserves the branching structure of the the DPN (modulo variable assignment). Given $\mathcal{N} = (P, T, F, V, dom, \alpha_I, guard)$:

Definition 5.7. The eager constraint graph $CG^{eager}_{\mathcal{N}}$ of $\mathcal{N}$ is a tuple $\langle N, n_0, A\rangle$ where:

- $N \subseteq \mathcal{M} \times 2^{\mathcal{C}_{\mathcal{N}}}$ is a set of nodes;
- $n_0 = (M_I, C_0)$ is the initial node, with $C_0 = \bigcup_{v \in V} \{v = \alpha_I(v)\}$;
- $A \subseteq N \times T \times N$ is the set of arcs, defined with S by mutual induction as follows. A transition $((M, C), t, (M', C'))$ is in A iff $M[t\rangle M'$ and one of the following cases applies:
  - (i) $write(t) = \{v\}$;
    (ii) $C'' \in Intervals^v_{\mathcal{N}}$;
    (iii) $C''' \in C^2_{\mathcal{N}}$;
    (iv) $C' = (C \oplus guard(t)) \cup C'' \cup C'''$ is satisfiable.
  - (i) $write(t) = \emptyset$
    (ii) $C' = C \oplus guard(t)$ is satisfiable;

Intuitively, in the case of a transition t with $write(t) \neq \emptyset$, an arc exists in $CG^{eager}_{\mathcal{N}}$ for each possible way in which a representative interval and a two-variable constraint set can be selected (guessed), so that C' is satisfiable. If $write(t) = \emptyset$ it is enough to check that $C \oplus guard(t)$ is satisfiable.

Example 5.8. A fragment of $CG^{eager}_{\mathcal{N}}$ for $\mathcal{N}$ as in Figure 2 is shown in Figure 6, exemplifying how representative intervals and two-variable constraint sets are employed to account for all possible (and satisfiable) evolutions of the original DPN. The guard of transition credit request is $(reqd^w \neq \bot)$, hence one possible successor node is added for each possible representative interval for variable $reqd$. Additionally, one possible two-variable constraint set is also guessed for each of these nodes: in this example, since $granted = \texttt{0}$, there exists only one constraint set among $\{(granted = reqd)\}$ and $\{(granted \neq reqd)\}$ that can be added to a successor node so that the resulting set is satisfiable. Since representative intervals and two variable constraints precisely encode the conditions on which the evolution of a DPN may branch in following steps, by guessing all possible combinations we can make sure to account for all possible runs. For transition verify, two possible successor nodes exist for each of the nodes added in the previous step: one for $ok = \texttt{true}$ and one for $ok = \texttt{false}$.

As for $CG^{lazy}_{\mathcal{N}}$, if $\mathcal{N}$ is bounded then also $CG^{eager}_{\mathcal{N}}$ is finite-state. Note how this implies that we can devise a construction algorithm, similarly to the case of a lazy-constraint graph, which checks on-the-fly (i.e., during the construction itself) whether $\mathcal{N}$ is bounded. A run of $CG^{eager}_{\mathcal{N}}$ is a sequence $\rho = (M_I, C_0) \xrightarrow{t_1} \cdots \xrightarrow{t_n} (M_n, C_n)$ and we say that AC controls $n = (M, C)$ iff AC controls M.

5.3. Verifying controllable data-aware soundness for bounded DPNs

We now present our algorithm for checking whether a given actor AC can guarantee the data-aware (weak) soundness of a bounded DPN (given T_{AC} and V_{AC}). The algorithm is executed directly on

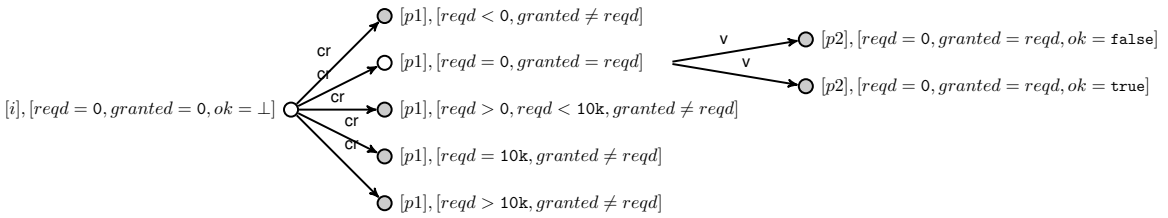

Figure 6. A fragment of $CG^{eager}_{\mathcal{N}}$ for $\mathcal{N}$ as in Figure 2, restricted to the first two transitions credit request and verify. As before, grey nodes are not expanded further for lack of space.

$CG^{eager}_{\mathcal{N}}$, computed following the definition in the previous section. It is based on a backward reachability procedure: starting from the nodes N' of the form (M_F, C) containing the final marking, we proceed backwards by computing the *controllable pre-image* of N', namely the set of states from where AC can enforce a transition firing that will lead to one in N'. By repeating such procedure until the initial node (M_I, C_0) is reached, or until the set of currently explored nodes does not change, we can assess whether the property is true. We thus formalize on $CG^{eager}_{\mathcal{N}}$ a definition analogous to Definition 5.1 on $RG_{\mathcal{N}}$, as follows.

Definition 5.9. Given a DPN $\mathcal{N}$ and sets T_{AC} and V_{AC} for a given actor AC, and considering the resulting eager constraint graph $CG^{eager}_{\mathcal{N}} = \langle N, n_0, A\rangle$, we say that AC can *enforce a set of nodes* $N' \subseteq N$ from the current node n iff N' is a set computed as follows:

- if AC controls n then for each $n' \in N'$ we have that $n \xrightarrow{t} n'$ and either:
  - $write(t) = \emptyset$ or $write(t) \subseteq V_{\text{AC}}$, or
  - all n'' so that $n \xrightarrow{t} n''$ are in N';
- if AC does not control n then for each $n' \in N'$ we have that $n \xrightarrow{t} n'$ and either:
  - $write(t) \neq \emptyset$ and $write(t) \subseteq V_{\text{AC}}$, or
  - all n'' so that $n \xrightarrow{t} n''$ are in N';

  there is no t' so that $n' \notin N'$ for some n' with $n \xrightarrow{t'} n'$.

We denote the sets of all possible sets N' of nodes as above as $Ctrl^n_{\text{AC}}$: these are the sets of nodes that can be enforced by the actor AC from a node n. As it was the case for $RG_{\mathcal{N}}$, if the actor AC controls all enabled transitions and all the variables that these write, then AC can handpick a specific node to be the successor (namely $Ctrl^n_{\text{AC}}$ also contains all the sets $\{n'\}$ so that $n \xrightarrow{t} n'$ for some t).

Given a set $N' \subseteq N$ of arbitrary nodes of $CG^{eager}_{\mathcal{N}}$, the controllable pre-image $Pre^{\text{AC}}(N')$ of the set N' for the actor AC is the set computed as follows:

$$Pre^{\text{AC}}(N') \doteq \{n \in N \mid \exists N'' \in Ctrl^n_{\text{AC}}.\ N'' \subseteq N'\}.$$

In other words, $Pre^{\text{AC}}(N')$ is the set of all nodes such that, no matter how the environment chooses transitions and/or variable assignments (if allowed to do so, i.e., if AC does not control the current marking or the written variable is not in V_{AC}), the next node can be guaranteed to be in N'.

Based on this simple building block, we compute the set $Y \subseteq N$ of nodes of $CG^{eager}_{\mathcal{N}}$ from where a node with final marking can always be reached (no matter how the environment behaves) by

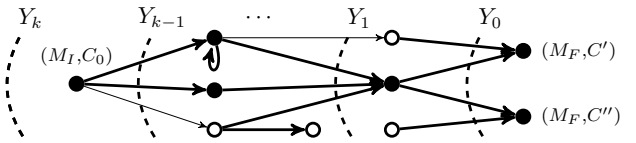

Figure 7. Intuitive depiction of the procedure. For simplicity, for each n $Ctrl^n_{\text{AC}}$ contains only one set, marked with bold outgoing arcs. Black nodes are in Y_R.

repeatedly applying the Pre^{AC} operator until a fixpoint is reached (see Fig. 7). Formally:

$$Y_0 \doteq \{(M, C) \in N \mid M = M_F\}; \qquad Y_{i+1} \doteq Y_i \cup Pre^{\text{AC}}(Y_i); \qquad Y \doteq \bigcup_{i \leq |N|} Y_i.$$

namely, Y_0 is the set of nodes with a final marking; then Y_{i+1} for $i \geq 0$ is computed from Y_i by applying the Pre^{AC} operator. After a bounded number of iterations, a fixpoint is necessarily reached.

As a consequence, from the states in Y the actor AC can force the process to eventually reach a node of the form (M_F, C), i.e., with final marking. The following result is thus a direct consequence of the computation above: if the initial node of $CG^{eager}_{\mathcal{N}}$ is in Y it means that the actor AC can, step-by-step, enforce successor nodes so that eventually a node with final marking is reached. As we are going to prove, since there exists a *bisimulation* property between $RG_{\mathcal{N}}$ and $CG^{eager}_{\mathcal{N}}$ these graphs have the same branching structure (modulo the identity of variable values) and therefore a process execution strategy $strat$ exists for $RG_{\mathcal{N}}$ so that $RG^{strat}_{\mathcal{N}}$ is data-aware sound iff nodes with final marking are reachable in $CG^{eager}_{\mathcal{N}}$ by only traversing nodes in Y.

Theorem 5.10. Given a bounded DPN $\mathcal{N}$ and sets T_{AC} and V_{AC} for a given actor AC, then AC can guarantee $\mathcal{N}$ to be data-aware weak sound iff the initial node (M_I, C_0) of $CG^{eager}_{\mathcal{N}}$ is in Y.

For (full) data-aware soundness, which also requires that no dead transition exists (see Def. 3.6), we also check that each transition $t \in T$ labels at least one arc between nodes in Y that are reachable in $CG^{eager}_{\mathcal{N}}$ from the initial node, denoted Y_R. More precisely, nodes in Y_R are those reachable by only traversing nodes in Y through arcs $n \xrightarrow{t} n'$ so that $n \in Y_{i+1}$ and $n' \in Y_i$.

The result then follows by construction. The algorithm above (computing Y) always terminates: the number of possible iterations are bounded by $|N|$, which is finite. In fact, the same considerations on the complexity as the previous section apply to this procedure as well.

Proof. (Sketch.) First, we need to prove that a (simple modification of the classical notion of) bisimulation exists between $CG^{eager}_{\mathcal{N}}$ and $RG_{\mathcal{N}}$. The definition is as follows. Given a reachability graph $RG_{\mathcal{N}} = \langle S, s_0, E\rangle$ of a DPN $\mathcal{N}$ and its constraint graph $CG^{eager}_{\mathcal{N}} = \langle N, n_0, A\rangle$, we say that a relation $R \subseteq N \times S$:

- is a abstraction relation of $RG_{\mathcal{N}}$ by $CG^{eager}_{\mathcal{N}}$ if $\langle (M, C), (M', \alpha)\rangle \in R$ implies that
  1. $M = M'$ and α is a solution of C (the first requirement trivially holds for DPNs);
  2. for any arc $(M', \alpha) \xrightarrow{t,\beta} (M'', \alpha')$ in $RG_{\mathcal{N}}$ there is in $CG^{eager}_{\mathcal{N}}$ an arc $(M, C) \xrightarrow{t} (M''', C')$;
  3. $\langle (M''', C'), (M'', \alpha')\rangle \in R$.
- is a grounding relation of $CG^{eager}_{\mathcal{N}}$ by $RG_{\mathcal{N}}$ if $\langle (M, C), (M', \alpha)\rangle \in R$ implies that

1. $M = M'$ and α is a solution of C;
2. for any arc $(M, C) \xrightarrow{t} (M', C')$ in $CG^{eager}_{\mathcal{N}}$ there exists $(M', \alpha) \xrightarrow{t,\beta} (M''', \alpha')$ in $RG_{\mathcal{N}}$;
3. $((M', C'), (M''', \alpha')) \in R$.

Proving that a relation R that is both a grounding and abstraction relation between $CG^{eager}_{\mathcal{N}}$ and $RG_{\mathcal{N}}$ can be done by induction and it relies on the definitions of these structures, which both encode the possible transitions of the DPN $\mathcal{N}$. By inspecting their definition, it is evident that there is indeed a direct correspondence between the arcs of $CG^{eager}_{\mathcal{N}}$ and $RG_{\mathcal{N}}$: a transition $(M, C) \xrightarrow{t} (M', C')$ in $CG^{eager}_{\mathcal{N}}$ implies that both C and C' are satisfiable, hence a SV assignment α' and variable assignment β exist so that $(M, \alpha) \xrightarrow{t,\beta} (M', \alpha')$ is in $RG_{\mathcal{N}}$, so that α' is a solution of C' (the fact that α is a solution of C is guaranteed by induction, since it is trivially true in the initial state and node).

With this result at hand, it is sufficient to note that if a run exists in $CG^{eager}_{\mathcal{N}}$ that only traverses nodes in Y (and in particular through arcs $n \xrightarrow{t} n'$ so that $n \in Y_{i+1}$ and $n' \in Y_i$), then a corresponding run exists in $RG_{\mathcal{N}}$ so that a relation R as above holds step-by-step. This in turn implies that if (M_I, C_0) is in Y and hence a run ρ exists from (M_I, C_0) to a node (M_F, C) irrespective of the choices made by the environment, we can find corresponding runs ρ' from (M_I, α_I) in $RG_{\mathcal{N}}$ so that, at each step, a SV assignment α that is a solution of the constraint sets C traversed by ρ exists, otherwise ρ would not be a legal run of $CG^{eager}_{\mathcal{N}}$. It then remains to note that a process execution strategy for $RG_{\mathcal{N}}$ can be simply obtained by returning transition firings (t, β) that are consistent with the constraint sets along runs ρ', since there is a direct correspondence between Definition 5.1 and Definition 5.9. Nonetheless, automatically extracting the process execution strategy requires further computation and it is not necessary for our result, so it is left as future work. □

6. Conclusions and Future work

In this paper we extended the work in [20, 21] to study the notion of soundness of DPNs to a general class of DPNs. We have shown the decidability of the problem of assessing the data-aware soundness for this class of nets by reducing it to the analysis of a finite-state abstraction of the possible executions of the original DPN. Our result extends to any constraint language that generates only boundedly many constraints over a fixed set of variables and constants and has decidable satisfiability. Further, we extended known results to a more involved verification task: that of verifying whether an actor can guarantee a process to be data-aware sound, if this can be represented as a bounded DPN. We gave technical insights on why the known approaches for data-aware soundness are not sufficient for addressing such task, and then provided a sound and complete technique for its solution, based on a novel definition of constraint graph. The same technique is therefore applicable to any further property which only relies on the branching structure (modulo variable values) of $RG_{\mathcal{N}}$. This includes the study of automated approaches for computing process repairs and process variations that guarantee given properties, from soundness to temporal specifications.

In future work, we plan to implement the approach described in this paper by combining standard state-space construction methods with constraint programming techniques, needed to perform the satisfiability checks required by the abstraction. We also plan to implement *synthesis* algorithms that allow not only to verify controllable data-aware soundness but also compute process execution strategies.

Acknowledgements

This work was supported in part by the Unibz projects DACoMan and VERBA.

References

[1] Reichert M. Process and Data: Two Sides of the Same Coin? In: Proc. of OTM, volume 7565 of *LNCS*. Springer, 2012 pp. 2–19.

[2] Calvanese D, De Giacomo G, Montali M. Foundations of Data Aware Process Analysis: A Database Theory Perspective. In: Proc. of PODS. ACM, 2013 .

[3] Decision Model and Notation (DMN) V1.1, 2016. URL `http://www.omg.org/spec/DMN/1.1/`.

[4] Calvanese D, Dumas M, Laurson Ü, Maggi FM, Montali M, Teinemaa I. Semantics and Analysis of DMN Decision Tables. In: Proc. of BPM, volume 9850 of *LNCS*. Springer, 2016 pp. 217–233.

[5] Batoulis K, Weske M. Soundness of Decision-Aware Business Processes. In: Proc. of BPM Forum. Springer, 2017 pp. 106–124.

[6] de Leoni M, van der Aalst WMP. Data-aware Process Mining: Discovering Decisions in Processes Using Alignments. In: SAC 2013. ACM. ISBN 978-1-4503-1656-9, 2013 pp. 1454–1461.

[7] de Leoni M, Mannhardt F. Decision Discovery in Business Processes. In: Sakr S, Zomaya A (eds.), Encyclopedia of Big Data Technologies. Springer International Publishing. ISBN 978-3-319-63962-8, 2018 .

[8] Mannhardt F. Multi-perspective process mining. Ph.d. thesis, Eindhoven University of Technology, 2018. https://research.tue.nl/en/publications/multi-perspective-process-mining.

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

[10] Montali M, Calvanese D. Soundness of Data-Aware, Case-Centric Processes. *Int. Journal on Software Tools for Technology Transfer*, 2016.

[11] Batoulis K, Haarmann S, Weske M. Various Notions of Soundness for Decision-Aware Business Processes. In: Proc. of ER, volume 10650 of *LNCS*. Springer, 2017 pp. 403–418.

[12] Rosa-Velardo F, de Frutos-Escrig D. Decidability and complexity of Petri nets with unordered data. *Theor. Comput. Sci.*, 2011. **412**(34):4439–4451.

[13] Lasota S. Decidability Border for Petri Nets with Data: WQO Dichotomy Conjecture. In: Proc. of PN, volume 9698 of *LNCS*. Springer, 2016 pp. 20–36.

[14] Triebel M, Sürmeli J. Homogeneous Equations of Algebraic Petri Nets. In: Proc. of CONCUR, LNCS. Springer, 2016 pp. 1–14.

[15] Fahland D. Describing Behavior of Processes with Many-to-Many Interactions. In: Proc. of PN. Springer, 2019 pp. 3–24.

[16] Montali M, Rivkin A. DB-Nets: On the Marriage of Colored Petri Nets and Relational Databases. *T. Petri Nets and Other Models of Concurrency*, 2017. **12**:91–118.

[17] Polyvyanyy A, van der Werf JMEM, Overbeek S, Brouwers R. Information Systems Modeling: Language, Verification, and Tool Support. In: Proc. of CAiSE, volume 11483 of *LNCS*. Springer, 2019 pp. 194–212.

[18] Ghilardi S, Gianola A, Montali M, Rivkin A. Petri Nets with Parameterised Data - Modelling and Verification. In: Proc. of BPM, volume 12168 of *LNCS*. Springer, 2020 pp. 55–74.

[19] de Leoni M, Munoz-Gama J, Carmona J, van der Aalst WMP. Decomposing Alignment-Based Conformance Checking of Data-Aware Process Models, pp. 3–20. Springer, 2014.

[20] de Leoni M, Felli P, Montali M. A Holistic Approach for Soundness Verification of Decision-Aware Process Models. In: Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings. 2018 pp. 219–235.

[21] Felli P, de Leoni M, Montali M. Soundness Verification of Decision-Aware Process Models with Variable-to-Variable Conditions. In: 2019 19th International Conference on Application of Concurrency to System Design (ACSD). 2019 pp. 82–91.

[22] Clarke EM, Grumberg O, Long DE. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 1994. **16**(5):1512–1542.

[23] Morimoto S. A Survey of Formal Verification for Business Process Modeling. In: Proceedings of the 8th International Conference on Computational Science, Part II, ICCS '08. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-69386-4, 2008 pp. 514–522.

[24] Sadiq S, Orlowska M, Sadiq W, Foulger C. Data Flow and Validation in Workflow Modelling. In: Proceedings of the 15th Australasian Database Conference - Volume 27, Proc. of ADC '04. Australian Computer Society, Inc., Darlinghurst, Australia, 2004 pp. 207–214.

[25] Corradini F, Fornari F, Polini A, Re B, Tiezzi F, Vandin A. BProVe: A Formal Verification Framework for Business Process Models. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017. IEEE Press, 2017 pp. 217–228.

[26] Corradini F, Fornari F, Polini A, Re B, Tiezzi F. A formal approach to modeling and verification of business process collaborations. *Science of Computer Programming*, 2018. **166**:35 – 70.

[27] Sidorova N, Stahl C, Trčka N. Soundness Verification for Conceptual Workflow Nets with Data: Early Detection of Errors with the Most Precision Possible. *Information Systems*, 2011. **36**(7):1026–1043.

[28] Knuplesch D, Ly LT, Rinderle-Ma S, Pfeifer H, Dadam P. On Enabling Data-Aware Compliance Checking of Business Process Models. In: Proc. of ER, volume 6412 of *LNCS*. Springer, 2010 pp. 332–346.

[29] Cassandras CG, Lafortune S. Introduction to Discrete Event Systems. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387333320.

[30] Giua A, Seatzu C. Petri nets for the control of discrete event systems. *Software & Systems Modeling*, 2015. **14**(2):693–701. doi:10.1007/s10270-014-0425-1.

[31] Iordache M, Antsaklis P. Supervisory control of concurrent systems: A petri net structural approach, pp. 1–278. Springer, 2006.

[32] Esparza J. Decidability and Complexity of Petri Net Problems - An Introduction. In: Reisig W, Rozenberg G (eds.), Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1996 pp. 374–428.