Repair of Unsound Data-Aware Process Models

Matteo Zavatteri, Davide Bresolin, and Massimiliano de Leoni

Department of Mathematics, University of Padova, Padova, Italy
{matteo.zavatteri, davide.bresolin, massimiliano.deleoni}@unipd.it

Abstract. Process-aware Information Systems support the enactment of business processes, and rely on a model that prescribes which executions are allowed. As a result, the model needs to be sound for the process to be carried out. Traditionally, soundness has been defined and studied by only focusing on the control-flow. Some works proposed techniques to repair the process model to ensure soundness, ignoring data and decision perspectives. This paper puts forward a technique to repair the data perspective of process models, keeping intact the control flow structure. Processes are modeled by acyclic Data Petri Nets. Our approach repairs the Constraint Graph, a finite symbolic abstraction of the infinite state-space of the underlying Data Petri Net. The changes in the Constraint Graph are then projected back onto the Data Petri Net.

Keywords: Data Petri Net · soundness · business process · model repair

1 Introduction

Process-aware Information Systems (PAISs) are instances of a type of system that supports the execution of processes within an organization. The main advance of PAISs is that they guarantee the compliance of process executions with respect to a process model that is provided as input.

PAISs can only function properly if the input process model is sound. Otherwise, some process executions carried on through a PAIS might remain blocked in a deadlock, or might never be completed. Consider, e.g., a financial institute that grants loans to customers: if the PAIS is configured via an unsound model, some loan applications might remain within the organization without ever being accepted or rejected. This is clearly not desired, because it affects the customer satisfaction and/or hampers the reputation of the organization.

Checking soundness of process models has attracted a lot of attention in the past [1–6]. Recently, Felli et al. [7] studied verification of soundness of DPNs with guards that include comparison of variables, and introduced the constraint graph, on which we also rely for model repair, but did not consider the repair of unsound models. Later on, the framework was extended to support full arithmetic constraints [8], at the price of loosing decidability of the soundness verification problem, as the underlying constraint graph is not guaranteed to be finite anymore.

Conversely, few research works have focused on automatically repairing process models to ensure soundness, and they do not consider the data perspective,

namely process variables and the related guards [9, 10]. Indeed, processes manipulate data when being executed, and this data restricts the behavior that the process allows. Repairing the process model while ignoring data may cause the model to remain unsound, as activities and data are intertwined.

This work is the first attempt to repair the data perspective of data-aware process models, keeping intact the control flow structure. Process models are here represented in the form of Data Petri Nets (DPNs). A general algorithm is proposed that can repair acyclic DPNs and that keeps intact the place/transition structure of the network, and that tries to minimize the number of guards that ought to be changed. The underlying assumption is that unsoundness is caused by the data perspective: if the unsoudness were due to the control-flow (i.e. the structure of the Petri-net underneath), this should be fixed beforehand, using techniques such as those described in [11]. We support guards in form of difference constraints, namely $x - y \bowtie k$ where x and y are real variables, k is a constant, and $\bowtie$ can be either $<$ or $\leq$. Unary guards, e.g. $x < k$, can also be supported. We prove that the algorithm always terminates, returning a sound DPN.

Organization. Section 2 introduces the notions of *(i)* systems of difference constraints and their consistency (i.e., satisfiability), and *(ii)* DPNs and soundness. Section 3 reports on our algorithm for repairing DPNs and proves termination, while Section 4 draws conclusions and discusses future work.

2 Data Petri Nets: Syntax, Semantics and Verification

This section summarizes the main concepts and formalization of the syntax and semantics of Data Petri nets, which are adapted from Felli et al. [7] to support difference constraints.

Data Petri nets are Petri Nets that are complemented by a set V of variables, whose values are updated through the transition firings. Guards are also associated to transitions, and provide further constraints to have transitions enabled.

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 $V^r := \{v^r \mid v \in V\}$ and $V^w := \{v^w \mid v \in V\}$. Intuitively, since an activity of the process may require to read and/or update the value of variables, we use v^r and v^w to respectively denote the variable v before and after the transition is executed. For this reason, we also refer to them as read and written variables, respectively. We omit the superscripts r and w to refer to a variable that can be either read or written.

A *difference constraint* over two real variables x, y has the form $y - x \bowtie k$, where $\bowtie$ is a comparison operator that can be either $<$ or $\leq$ and $k \in \mathbb{R} \cup \{+\infty\}$. Unary constraints of the form $x \bowtie k$ can be encoded into difference constraints $x - Z \bowtie k$ where Z is a fresh real variable intended to be always set to zero. An equality constraint $y - x = k$ can be encoded into two difference constraints $y - x \leq k$ and $x - y \leq -k$. This also holds for constraints $x = k$ (once rewritten as $x - Z = k$). A system of difference constraints is a set of difference constraints. A

system of difference constraint is *consistent* if there exists an assignment of real values to the variables that satisfies all constraints. A consistent system of difference constraints admits a unique *canonical representation* that can be computed via a generalization of Floyd-Warshall algorithm using *difference-bound matrices* to represent the canonical form [12], with time and space complexity $\Theta(n^3)$ where n is the number of variables. We can now formalize DPNs as follows.

Definition 1 (Data Petri Nets). *Let V be a set of variables. Let $\mathcal{C}_V$ be the universe of difference constraints over $V^r \cup V^w \cup \{Z\}$. A Data Petri Net (DPN) $\mathcal{N} = (P, T, F, V, \alpha_I, guard)$ is a Petri net (P, T, F) with additional components describing the additional perspectives of the process model:*

- *V is a finite set of real process variables*
- *$\alpha_I : V \cup \{Z\} \mapsto \mathbb{R}$ is a function defining the initial assignment with $\alpha_I(Z) = 0$*
- *$guard : T \to \mathcal{C}_V$ is a function returning the guard of a transition.*

Conjunctions and disjunctions are not allowed in the guard for simplicity. However, disjunctive guards can be mimicked by having multiple transitions from and to the same places, whereas read-only conjunctive guards can be modeled as "non-interruptible" sequences, as discussed in [7].

Given $t \in T$, as a shorthand we write $read(t) := \{v \in V \mid v \in read(guard(t))\}$, 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.

Consider the DPN in Figure 1a. From the initial marking $M_I = \{p_1\}$, transition t_1 updates the value of `a` to a value greater than `5`. Then, t_2 or t_3 may be executable depending on the current value of `a` being greater or smaller than `10`. Similarly, t_4 can be executed only if the initial value of `b` is smaller than the current value of `a`. The only possible sequence of transitions that reaches the final marking is t_1, t_2, t_4, as $\alpha_I(\texttt{b}) = \texttt{10}$. A simplistic analysis that disregards the possible 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 [13].

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 is the marking of the Petri net, that is, a multiset of places from P, and α is an assignment of the variables in V. 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 $write(t)$. A DPN $\mathcal{N}$ evolves from state (M, α) to state (M', α') through t if:

- t is enabled and the new marking is M' (denoted $M[t\rangle M'$) according to the Petri net semantics;

- for each $v \in V$, if $v \notin \mathit{write}(t)$ then the value of v is unchanged: $\alpha'(v) = \alpha(v)$;
- the guard is satisfied when we assign values to read variables according to α and to written variables according to α'.

We denote a legal transition firing by writing $(M, \alpha) \xrightarrow{t} (M', \alpha')$. We also extend this definition to sequences of legal transition firings (runs), and we write $(M, \alpha) \xrightarrow{*} (M', \alpha')$ if there exists a sequence of legal transition firings $(M, \alpha) \xrightarrow{t^1} \ldots \xrightarrow{t^n} (M', \alpha')$. For instance, referring to the simple DPN $\mathcal{N}$ in Figure 1a, a possible legal transition firing from the initial state is $(\{p_1\}, \{\alpha_I(\mathtt{a}) = \mathtt{0}, \alpha_I(\mathtt{b}) = \mathtt{10})\}) \xrightarrow{t_1} (\{p_2\}, \{\alpha(\mathtt{a}) = \mathtt{7}, \alpha(\mathtt{b}) = \mathtt{10})\})$.

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 $k \in \mathbb{N}$ 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).

Data-aware soundness. We recall here the lifting of the standard notion of soundness [13] to the the data-aware setting of DPNs, as illustrated in [7]. 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 the 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 2 (Data-aware soundness [7]). *A DPN with initial marking M_I and final marking M_F is data-aware sound iff all the following properties hold.*

P1. *For every reachable state (M, α), $\exists \alpha_F.\ (M, \alpha) \xrightarrow{*} (M_F, \alpha_F)$*
P2. *For every reachable state (M, α), $M \geq M_F \Rightarrow (M = M_F)$*
P3. *For every transition $t \in T$, there exist two reachable states (M_1, α_1) and (M_2, α_2) such that $(M_1, \alpha_1) \xrightarrow{t} (M_2, \alpha_2)$.*

The first condition imposes that it is *always* possible to reach the final marking by suitably choosing a continuation of the current run (i.e., legal transition firings). The second condition captures that the final marking 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 to enable it through the execution of the process. For instance, $P1$ is false for the DPN in Figure 1a: when transition t_1 assigns a value not greater than `10` to `a` there exists no run from there which marks M_F.

DPN Soundness Verification via Constraint Graph. This paper bases the soundness verification and the repair algorithm on the structure of the *constraint graph*, introduced by Felli et al. [7]. A constraint graph is a finite symbolic abstraction of the (possibly infinite) traces of a DPN, that allows to verify soundness of acyclic DPNs and to identify the changes needed to repair the DPN, if

Algorithm 1: Procedure for computing $C \oplus c$.

if $c = y^r - x^r \bowtie k$ **then** ▷ c is a read-only constraint
  $C' := C \cup \{y - x \bowtie k\}$
  return `CanonicalForm`(C')
else ▷ c writes some variable
  if $write(c) = \{x\}$ **then**
    $C' := C' \cup \{y - x^w \bowtie k\}$
  else if $write(c) = \{y\}$ **then**
    $C' := C' \cup \{y^w - x \bowtie k\}$
  else ▷ $write(c) = \{x, y\}$
    $C' := C' \cup \{y^w - x^w \bowtie k\}$
  $C' :=$ `CanonicalForm`(C')
  $C' := C' \setminus \{x' - y' \bowtie' k' \mid x' \in write(c) \text{ or } y' \in write(c)\}$
  Rename all occurrences of x^w to x and all occurrences of y^w to y in C'
  return C' ▷ C' is already in canonical form

found unsound. A constraint graph is characterized by a state-transition structure where each node is associated with a marking and an abstraction of the data, given as a canonical representation of a system of difference constraints.

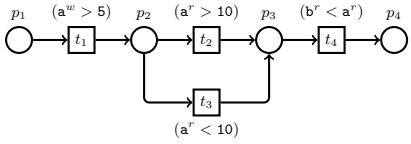
Given a set of difference constraints C, and a constraint c, we now define the procedure of computing the new constraint set C' resulting from the addition of a constraint c to C so that C' is uniquely determined, denoted $C' = C \oplus c$. This is shown in Algorithm 1, where we maintain the same notation as before, so that x, y, z can be either constants or read variables in V^r. It requires a `CanonicalForm` procedure that, given a set C' of difference constraints as input, returns the minimal constraint network derived from C' using the generalized Floyd-Warshall algorithm described above. When given an unsatisfiable constraint set, we assume `CanonicalForm` to return a null value, so that it can be used also to signal inconsistency of a set of difference constraints.

Definition 3 (Constraint Graph of a DPN). *Let $\mathcal{N} = (P, T, F, V, \alpha_I, guard)$ be a DPN, $\mathcal{M}$ be the set of markings of $\mathcal{N}$, and M_I the initial marking.da The constraint graph $CG_{\mathcal{N}}$ of $\mathcal{N}$ is a tuple $\langle N, n_0, A \rangle$ consisting of:*

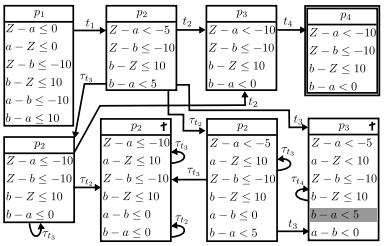
- *$N \subseteq \mathcal{M} \times \mathcal{C}^*$ is the set of* nodes *of the graph.*
- *$n_0 = (M_I, C_0)$ is the initial node where C_0 is the canonical form of the system of difference constraints $\bigcup_{v \in V} \{v = \alpha_I(v)\}$;*
- *$A \subset N \times (T \cup \tau_T) \times N$ is the set of arcs such that:*
  - *a transition $((M, C), t, (M', C'))$, where $t \in T$, is in A iff $M[t\rangle M'$ and $C' = C \oplus guard(t)$ is consistent;*
  - *a transition $((M, C), \tau_t, (M, C''))$, where $\tau_t \in \tau_T$, is in A iff $write(t) = \emptyset$, $\exists M'$ s.t. $M[t\rangle M'$, and $C'' = C \oplus \neg guard(t)$ is consistent.*

3 The Repair of Data Petri Nets

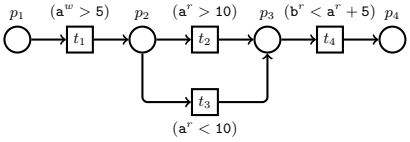
In this section we describe the repair algorithm. As stressed in the introduction, we assume that the underlying Petri Net (i.e., without the data dimension) of



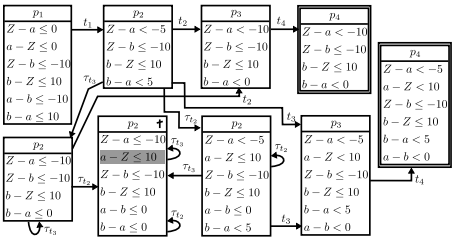
(a) Example of DPN $\mathcal{N}$ [7]. $M_I = \{p_1\}$, $M_F = \{p_4\}$, $\alpha_I(\mathtt{a}) = \mathtt{0}$, $\alpha_I(\mathtt{b}) = \mathtt{10}$.



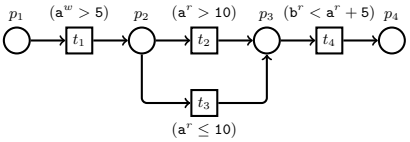
(b) $CG_{\mathcal{N}}$ showing unsoundness of $\mathcal{N}$ (dead nodes).



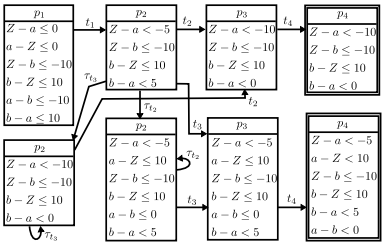
(c) Modification 1: $\mathcal{N}'$ is $\mathcal{N}$ with the guard of t_4 modified as $\mathtt{b}^r < \mathtt{a}^r + \mathtt{5}$.



(d) $CG_{\mathcal{N}'}$ showing unsoundness of $\mathcal{N}'$ (dead nodes).



(e) Modification 2: $\mathcal{N}''$ is $\mathcal{N}'$ with the guard of t_3 modified as $\mathtt{a}^r \leq \mathtt{10}$.



(f) $CG_{\mathcal{N}''}$ showing soundness of $\mathcal{N}''$.

Fig. 1: Repair algorithm using `ForwardRepair` only.

the DPN to repair is sound and acyclic, and we focus on the repair of the data perspective only. Informally, given an acyclic DPN $\mathcal{N}$, we want to build a sound DPN $\mathcal{N}'$ by changing only the constants and comparison operators in the guards. Moreover, we look for a repair that limits the number of changes.

Definition 4. *Let $\mathcal{N} = (P, T, F, V, \alpha_I, guard)$. A repair of $\mathcal{N}$ is a DPN $\mathcal{N}' = (P', T', F', V', \alpha'_I, guard')$ meeting the following three conditions:*

1. $(P, T, F, V, \alpha_I) = (P', T', F', V', \alpha'_I)$;
2. *for each transition $t \in T$ with $guard(t) = y - x \bowtie k$, $guard'(t) = y - x \bowtie' k'$;*
3. *$\mathcal{N}'$ is data-aware sound.*

The cost of the repair $\mathcal{N}'$ is the number of guards in $\mathcal{N}'$ that differ from $\mathcal{N}$.

Algorithm 2: An algorithm to repair a DPN.

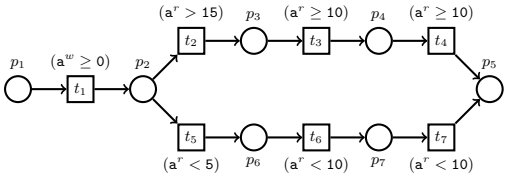
```
1  DPNRepair (𝒩)                                   ▷ Assume that 𝒩 has global visibility
2      Let 𝒬 be a empty priority queue.            ▷ global variable
3      Push the input DPN 𝒩 in 𝒬 with priority 0.
4      Let 𝒩′ be an empty DPN.                     ▷ the one to return eventually
5      while true do
6          Extract from 𝒬 a DPN with minimum priority p and save it in 𝒩′
7          Let CG_𝒩′ be the constraint graph of 𝒩′
8          if CG_𝒩′ is data-aware sound then break   ▷ we are done, exit the while loop
9          FixDead(𝒩′, CG_𝒩′)
10         FixMissing(𝒩′, CG_𝒩′)
11     return 𝒩′                                   ▷ the repaired DPN (last extracted from 𝒬)
12 UpdateQ (𝒩′)                                    ▷ used in FixDead, FixMissing
13     if 𝒩′ has not been visited yet then
14         Let p be the number of transitions of 𝒩′ with a different guard in 𝒩.
15         Push 𝒩′ in 𝒬 with priority p.
16 FixDead (𝒩′, CG_𝒩′)
17     foreach dead node (M, C) in CG_𝒩′ do
18         Let FW := {t ∈ T | M[t⟩M′ for some marking M′}.
19         foreach t ∈ FW do ForwardRepair(𝒩′, t, C)
20         Let BW be the set of non-silent transitions in all paths (M_0, C_0) ⇝ (M, C).
21         foreach t ∈ BW do BackwardRepair(𝒩′, t, C)
22 ForwardRepair (𝒩′, t, C)                        ▷ "replace with the same constraint of C"
23     Let 𝒩′′ := (P, T, F, V, α_I, guard′′) be a copy of 𝒩′.
24     Let y − x ⋈ k be the guard of t.
25     Let y − x ⋈′ k′ be the corresponding constraint in C.
26     guard′′(t) := y − x ⋈′ k′
27     UpdateQ(𝒩′′)
28 BackwardRepair (𝒩′, t, C)                       ▷ "replace with the opposite constraint of C"
29     Let 𝒩′′ := (P, T, F, V, α_I, guard′) be a copy of 𝒩′.
30     Let y − x ⋈ k be the guard of t.
31     if x − y ⋈′ k′ in C is such that k′ ≠ ∞ then
32         if ⋈′ is ≤ then guard′′(t) := y − x < −k′
33         else guard′′(t) := y − x ≤ −k′
34         UpdateQ(𝒩′′)
35 FixMissing (𝒩′, CG_𝒩′)
36     Let Missing be the set of missing transitions in CG_𝒩′.
37     foreach t ∈ Missing do
38         Let Nodes := {(M, C) | ∃M′.M[t⟩M′}
39         foreach (M, C) ∈ Nodes do
40             ForwardRepair(𝒩′, t, C)
41             Let BW be the set of non-silent transitions in all paths (M_0, C_0) ⇝ (M, C).
42             foreach t ∈ BW do BackForwardRepair(𝒩′, t)
43 BackForwardRepair (𝒩′, t)                        ▷ "make the guard true"
44     Let 𝒩′′ := (P, T, F, V, α_I, guard′) be a copy of 𝒩′.
45     Let y − x ⋈ k be the guard of t.
46     guard′′(t) := y − x ≤ ∞
47     UpdateQ(𝒩′′)
```

We show the repair algorithm for acyclic DPNs in Algorithm 2. The main procedure is `DPNRepair` which implements a Breadth First Search in a Dijkstra fashion to explore the space of possible Data Petri Nets. The function uses a priority queue to store the partial solutions that are being explored. At each

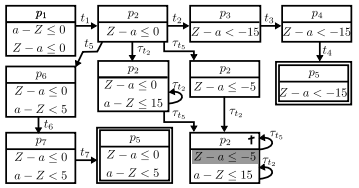
iteration the algorithm extracts a DPN with minimum priority, builds the corresponding constraint graph, and, if such a graph is unsound, it calls `FixDead` and `FixMissing`. `FixDead` processes dead nodes, that is, nodes that violate condition P1 of the definition of data-aware soundness (Definition 2). For each dead node, it identifies a set of transitions to operate on. For each transition, it computes a new guard from the systems of difference constraints belonging to the dead node. After that, it pushes in the queue the resulting new DPN. `FixMissing` processes transitions that do not appear in the constraint graph, namely those that violate condition P3 of Definition 2. For each transition, it identifies the set of nodes in the constraint graph from which the transition might fire, and another set of transitions to operate on. For each transition in this set, it computes a new guard and pushes the resulting DPN in the queue.

Dead nodes can be treated in two possible ways. Either we enable a transition that is currently prevented to fire from that node, or we prevent the execution to reach that node. The first case is managed by `ForwardRepair`, whereas the second one by `BackwardRepair`. Consider Figure 1. The initial DPN (Figure 1a) is data-aware unsound. Indeed, its constraint graph (Figure 1b) reveals the presence of two dead nodes (identified by a cross). Consider the rightmost dead node. That node has marking $M = \{p_3\}$ and thus the only transition that might fire from the node is t_4. However, t_4 is not enabled because the set of constraints in the node augmented with the guard of t_4 is inconsistent. The current guard of t_4 is $\mathtt{b}^r < \mathtt{a}^r$, that in difference constraint form, regardless of read and write operations, is $\mathtt{b} - \mathtt{a} < 0$. The same constraint in the set of constraints C of the dead node is $\mathtt{b} - \mathtt{a} < 5$. This says that if the guard of t_4 was $\mathtt{b}^r < \mathtt{a}^r + 5$, then t_4 would be able to fire. Therefore, such constraint becomes the new guard of t_4. The resulting DPN $\mathcal{N}'$ (Figure 1d) is computed by $\mathtt{ForwardRepair}(\mathcal{N}, t_4, C)$ by applying this modification and it is pushed in the queue with priority 1, since the number of guards differing from the original DPN is currently one. The algorithm proceeds with a breadth-first search by exploring the other modifications given by the subsequent calls to `ForwardRepair` and `BackwardRepair` and pushing other candidate DPNs in the queue. When $\mathcal{N}'$ is extracted from the queue, the algorithm builds its constraint graph (Figure 1d) discovering another dead node. By proceeding similarly, we can push a new DPN $\mathcal{N}''$ (Figure 1e) in the queue, obtained by modifying the current guard of t_3 which is $\mathtt{a}^r < 10$ (i.e., $\mathtt{a} - Z < 10$) to $\mathtt{a}^r \leq 10$ (i.e., $\mathtt{a} - Z \leq 10$). When $\mathcal{N}''$ is extracted from the queue the algorithm verifies that it is data-aware sound and returns $\mathcal{N}''$ as the repaired DPN. This example shows a possible path to a solution that uses only `ForwardRepair`.

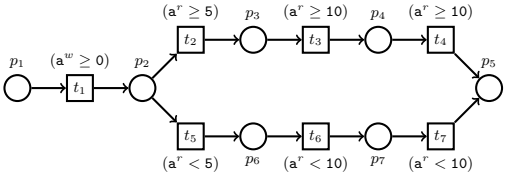
Yet, there are cases in which solutions with smaller costs can be found if we also use `BackwardRepair`. For instance, consider $\mathcal{N}$ in Figure 2a. In the constraint graph of $\mathcal{N}$ (Figure 2b) there is a dead node. Consider the application of $\mathtt{ForwardRepair}(\mathcal{N}, t_2, C)$, where C is the set of difference constraints of the dead node. The result is $\mathcal{N}'$, shown in Figure 2c, still contains a dead node. One possible path to continue is to apply `ForwardRepair` to t_3 and then to t_4 (since changing t_3 generates a dead node from which t_4 cannot be executed) to



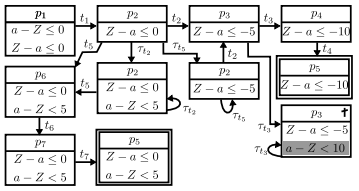
(a) Example of DPN $\mathcal{N}$. $M_I = \{p_1\}$, $M_F = \{p_5\}$, and $\alpha_I(\mathtt{a}) = 0$.



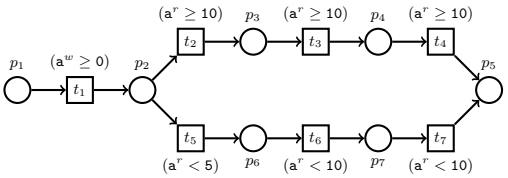
(b) $CG_{\mathcal{N}}$ showing unsoundness of $\mathcal{N}$ (dead node).



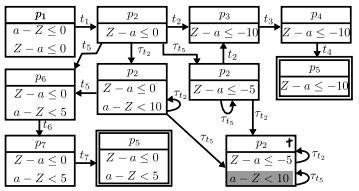
(c) Modification 1: $\mathcal{N}'$ is $\mathcal{N}$ with the guard of t_2 modified as $Z - a \leq -5$.



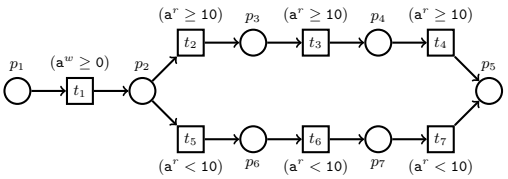
(d) $CG_{\mathcal{N}'}$ showing unsoundness of $\mathcal{N}'$ (dead nodes).



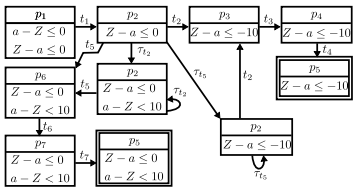
(e) Modification 2: $\mathcal{N}''$ is $\mathcal{N}'$ with the guard of t_2 modified as $Z - a \leq -10$.



(f) $CG_{\mathcal{N}''}$ showing unsoundness of $\mathcal{N}''$ (dead nodes).



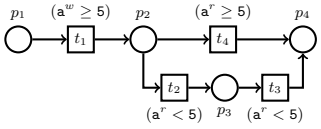
(g) Modification 3: $\mathcal{N}'''$ is $\mathcal{N}''$ with the guard of t_5 modified as $a - Z < 10$.



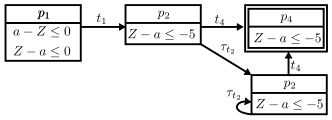
(h) $CG_{\mathcal{N}'''}$ showing soundness of $\mathcal{N}'''$.

Fig. 2: Repair algorithm using `ForwardRepair` and `BackwardRepair` only.

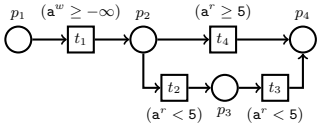
obtain a solution of cost 3. However, a solution with smaller cost can be found by following a different path, that proceeds by applying `BackwardRepair` to t_2, since t_2 is a transition belonging to a path that can reach the dead node from the initial one. The current guard of t_2 is $\mathtt{a}^r \geq 5$ (i.e., $Z - \mathtt{a} \leq -5$). To prevent the execution to reach the dead node we can restrict the guard of t_2 so that the constraint system of the dead node becomes inconsistent with this new guard. To do so, we need to modify the guard of t_2 to $\mathtt{a}^r \geq 10$ (i.e., $Z - \mathtt{a} \leq -10$). This modification generates a negative weight cycle (i.e., a certificate of inconsistency)



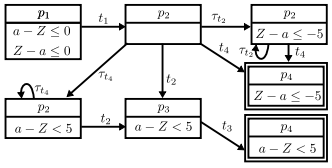
(a) Example of DPN $\mathcal{N}$. $M_I = \{p_1\}$, $M_F = \{p_3\}$, and $\alpha_I(\mathtt{a}) = 0$.



(b) $CG_{\mathcal{N}}$ showing unsoundness of $\mathcal{N}$ (missing transitions).



(c) Modification 1: $\mathcal{N}'$ is $\mathcal{N}$ with the guard of t_1 modified as $\mathtt{a}^w \geq -\infty$.



(d) $CG_{\mathcal{N}'}$ showing soundness of $\mathcal{N}'$.

Fig. 3: One-shot repair using `BackForwardRepair` only.

with the constraint $\mathtt{a} - Z < 10$ contained in the system of constraints of the dead node. Hence, that node will no longer exist in the constraint graph. Figure 2e shows the DPN $\mathcal{N}''$ obtained by this last modification, whose constraint graph (Figure 2f) still contains a dead node. If we apply `ForwardRepair` to t_5 we obtain the repaired DPN $\mathcal{N}'''$ in Figure 2g with a total cost of 2. Thus, there are cases in which, by operating several times on the same transition, we can obtain repairs with smaller costs.

A constraint graph can also be unsound because of missing transitions. This situation does not necessarily imply the existence of dead nodes and it is therefore handled by the function `FixMissing`. Missing transitions can be treated in two possible ways. Either we enable the missing transitions to fire from the nodes where the marking allows them but the data does not, or we remove constraints by operating on transitions along the paths from the initial node to the node under analysis. The former case is still handled by `ForwardRepair`, whereas the latter is handled by `BackForwardRepair` that mixes ideas from `ForwardRepair` and `BackwardRepair`. Once again, we proceed by discussing a concrete example. Consider the DPN $\mathcal{N}$ in Figure 3a and its constraint graph in Figure 3b. There are no dead nodes in the constraint graph. However, t_2 (and thus t_3 occurring after t_2) are missing in the constraint graph. A solution of cost 1 can be found by applying `BackForwardRepair` to t_1 since t_1 is in the path that goes from the initial node to a node with the marking $M = \{p_2\}$ from which t_2 can fire in the underlying "dataless" Petri Net. `BackForwardRepair` replaces the guard of t_1 with $\mathtt{a}^w \geq -\infty$ (i.e., $Z - \mathtt{a}^w \leq \infty$) by obtaining the data-aware sound $\mathcal{N}'$ (Figure 3c and Figure 3d). Despite `BackForwardRepair` basically sets a guard to "true", we recall that the same guard might be later be processed by `BackwardRepair` in order to be restricted adequately.

Theorem 1. *Let $\mathcal{N}$ be an acyclic DPN where the underlying dataless Petri net is sound. Algorithm 2 terminates on $\mathcal{N}$ by returning a data-aware sound DPN.*

Proof. Let $\mathcal{N}$ be an acyclic DPN where the underlying Petri net is sound. First of all, notice that, by neglecting self-loop silent transitions, the constraint graph of an acyclic DPN is a DAG. Since the set of transitions T is finite and the underlying DPN is bounded, the branching factor of each node in the constraint graph is bounded by $2 \cdot |T|$ (i.e., all transitions t plus the corresponding silent transitions τ_t), and the longest path from the initial node to a final one cannot exceed $|T|$. As a result, there are at most $(2 \cdot |T|)^{|T|}$ nodes in the constraint graph. Among the possible sequences of modifications that Algorithm 2 can follow there always exists one that uses `ForwardRepair` only. Such a sequence can always be explored since (i) all constraint graphs built along the way are finite, (ii) the possible modifications applied to a constraint graph are finite, and (iii) such modifications are explored following a BFS strategy. Consider therefore the sequence that calls only `ForwardRepair`. Every time a new DPN is generated the guard of a transition t is replaced with some constraint in the constraint system of some node: either a dead node if `ForwardRepair` is called inside `FixDead`, or a node from which a missing transition can be fired if `ForwardRepair` is called inside `FixMissing`. In the former case (fixing a dead node), some paths in the constraint graph starting from the initial node and ending with the silent transition τ_t are removed. Also, such paths can never be introduced again by subsequent modifications: if t is processed again, the current guard $y - x \bowtie k$ is replaced by a weaker guard $y - x \bowtie' k'$. In the latter case (fixing a missing transition), some paths in the constraint graph are extended with the transition t. By the same monotonicity argument on subsequent modifications of t, such extended paths can never be removed in subsequent applications of `ForwardRepair`. Since the number of paths in a constraint graph is finite, the sequence of `ForwardRepair` reaches a sound DPN in a finite number of steps. □

4 Conclusions

This paper focuses on repairing data-aware process models to ensure soundness. We use DPNs as modelling formalism, and employ difference constraints over real variables as transition guards. We defined a general algorithm that can repair acyclic DPNs, keeps intact the place/transition structure of the network, and tries to minimize the number of guards that ought to be changed. The algorithm exploits the full power of difference constraint to build a repaired network with as few changes as possible. We rely on the canonical form of systems of difference constraints to compute the modifications on the guards of transitions. We proved that the algorithm terminates, returning a repaired DPN.

As future work, we plan to implement the algorithm and experimentally evaluate the efficiency with models of increasing complexity. We aim to investigate the optimality of the algorithm, and to extend the algorithm to support repair of cyclic DPNs. Finally, we are currently assuming that every guard change is equivalent to repair the model: in reality, process modellers and analysts may favor certain changes over others. This requires to define a cost framework where certain guard changes come at lower costs, thus being preferable.

Acknowledgements

This work was supported by the Project funded under the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.5 - Call for tender No. 3277 of 30 dicembre 2021 of Italian Ministry of University and Research funded by the European Union – NextGenerationEU; Project code: ECS00000043, Concession Decree No. 1058 of June 23, 2016, CUP C43C22000340006, Project title "iNEST: Interconnected Nord-Est Innovation Ecosystem". This work was also supported by INdAM, GNCS 2023, Project "Analisi simbolica e numerica di sistemi ciberfisici".

References

1. Morimoto, S.: A survey of formal verification for business process modeling. In: ICCS 2008, Springer-Verlag (2008) 514–522
2. 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 **166** (2018) 35 – 70
3. 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 **36**(7) (2011) 1026–1043
4. Calvanese, D., Dumas, M., Laurson, Ü., Maggi, F.M., Montali, M., Teinemaa, I.: Semantics and analysis of DMN decision tables, Springer (2016) 217–233
5. Batoulis, K., Weske, M.: Soundness of decision-aware business processes. In: Proc. of BPM Forum, Springer (2017) 106–124
6. Batoulis, K., Haarmann, S., Weske, M.: Various notions of soundness for decision-aware business processes. In: Conceptual Modeling, Springer (2017) 403–418
7. Felli, P., de Leoni, M., Montali, M.: Soundness verification of data-aware process models with variable-to-variable conditions. Fund. Inform. **182**(1) (2021) 1–29
8. Felli, P., Montali, M., Winkler, S.: Soundness of data-aware processes with arithmetic conditions. In Franch, X., Poels, G., Gailly, F., Snoeck, M., eds.: Advanced Information Systems Engineering, Springer (2022) 389–406
9. Gambini, M., La Rosa, M., Migliorini, S., Ter Hofstede, A.H.M.: Automated error correction of business process models. In: BPM 2011, Springer (2011) 148–165
10. Armas Cervantes, A., van Beest, N.R.T.P., La Rosa, M., Dumas, M., García-Bañuelos, L.: Interactive and incremental business process model repair. In: On the Move to Meaningful Internet Systems. OTM 2017, Springer (2017) 53–74
11. Gambini, M., La Rosa, M., Migliorini, S., Ter Hofstede, A.H.M.: Automated error correction of business process models. In Rinderle-Ma, S., Toumani, F., Wolf, K., eds.: Business Process Management, Springer (2011) 148–165
12. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: CAV 1989. Volume 407 of LNCS., Springer (1989) 197–212
13. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers **8**(1) (1998) 21–66