Repairing Soundness Properties in Data-Aware Processes

1st Paolo Felli
University of Bologna
Bologna, Italy
paolo.felli@unibo.it

2nd Marco Montali
Free University of Bozen-Bolzano
Bolzano, Italy
montali@inf.unibz.it

3rd Sarah Winkler
Free University of Bozen-Bolzano
Bolzano, Italy
winkler@inf.unibz.it

***Abstract*—Within the growing area of data-aware processes, Data Petri nets (DPNs) with arithmetic data have recently gained popularity thanks to their ability to balance simplicity with expressiveness. DPNs can be automatically mined from event data, but these process discovery techniques typically come without any correctness guarantees. In particular, the generated models may violate the crucial property of *data-aware soundness*. While data-aware soundness can be checked automatically for a large class of models, nothing is known about how to *repair* such processes once a violation is detected. In this paper we are concerned with repairing DPNs so that the refined model satisfies the desired soundness properties. Our approach is based on conservative behavioural changes, which are minimally invasive in the sense that the behaviour of the repaired model coincides with that of the original model except for (prefixes of) traces that caused the violation. We show experimentally that the approach can be used to repair unsound DPNs from the literature.**

***Index Terms*—data-aware processes, process repair, soundness**

I. Introduction

Multi-perspective process models combine control-flow modeling, which specifies the legal execution sequencings of activities within process executions, with additional dimensions. A prominent example are data-aware process models, which explicitly represent how information systems dynamically operate over data through activities; they are of growing relevance for business process management (BPM) applications. Specifically, we consider here Data Petri Nets (DPNs), which are a simple but expressive and flexible data-aware representation [14], [16].

A central characterization of correctness of processes is *soundness* [17]. Intuitively, it requires that (i) all process activities can be executed in some execution; (ii) from every reachable configuration the process can be concluded by reaching a *final* configuration and (iii) final configurations are always reached in a 'clean way', without leaving any thread of the process still hanging. The counterpart of this classical property for data-aware processes is *data-aware soundness* [2], [7]. The combination of data and arithmetics in addition to the control-flow perspective makes checking data-aware soundness highly challenging, and for DPNs it is in fact undecidable [11]. Nevertheless, a number of techniques for checking data-aware soundness [2], [7] were proposed, including extensions of these models to account for activities that perform arithmetic operations on case data [9], [11]. Notably, it was shown to be decidable for relevant classes of DPNs.

However, existing soundness checking techniques do not come with corresponding actionable techniques on how to revise and adapt a DPN that results to be unsound. This is essential in process mining pipelines. In fact, alongside such pipelines, models are continuously transformed and modified through automated techniques and manual interventions. For data-aware processes, unsoundness may arise: *(i)* from a discovery step, as discovery is typically handled with a two-phase approach, where first the control-flow is discovered, and in a second step it is enriched with data [4], [12], [15]; *(ii)* from a manual intervention step, as mastering the interplay between control-flow and data is particularly challenging.

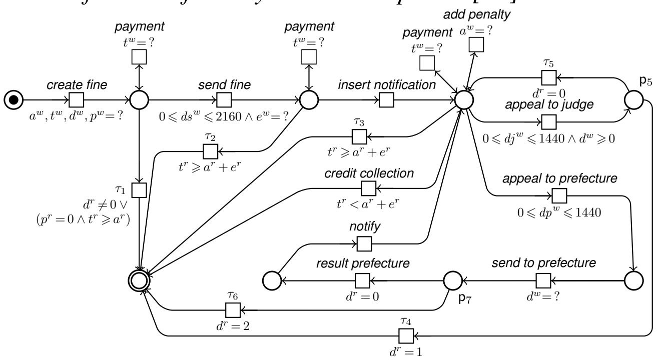
In this paper, we devise a novel approach for *repairing* unsound processes so that they become data-aware sound, by exploiting techniques developed for the verification and soundness analysis of DPNs. This approach can then be used as a component within a process mining pipeline, to ensure that whenever a DPN is discovered or manually altered, the resulting model can be made sound.

Our approach is based on two general principles: (1) *Behavioural conservatism*: We follow the premise that repairs should affect as little as possible the behaviors that are already permitted by the process, since these are assumed to account for executions recorded in the log. (2) *Control flow conservatism*: Considering that discovery techniques guaranteeing control-flow soundness actually exist [3], [15], [16], and recalling the typical two-phase discovery approach mentioned above, we consider input DPNs in which the underlying Petri Net, accounting only for the control-flow dimension, is already sound. Our task is then to enforce data-aware soundness by modifying the transition guards, irrespective of the approach used for their discovery (e.g., decision-aware control-flow discovery techniques or decision-annotated process mining techniques [4]). However, except for the removal of dead transitions, the control flow is not modified.

Due to the undecidability of the soundness checking task, which is required to build repaired DPNs, our procedure is a partial one. It is guaranteed to terminate for a simple class of DPNs, namely when guards are variable-to-variable/constant comparisons. For more general DPNs it need not terminate (see Sec. III), although in experiments this did not occur.

Notwithstanding this limitation, since existing discovery techniques cannot guarantee data-aware soundness in the presence of data and arithmetical conditions, we believe this to be an important contribution in the context of automated process discovery from event logs. We thus regard this work as the first step towards applications in which the automated discovery of data-aware processes is followed by a *repair activity* to enforce desirable properties like soundness. We start by giving a concrete example:

Example I.1. *The following DPN models a management process for road fines by the Italian police [16].*



This model was generated from real-life logs by automatic discovery techniques paired with domain knowledge [16], but without any correctness guarantee. One requirement of data-aware soundness is that a process can reach a final state from any reachable configuration. This property is violated here, as the process can get stuck in p_7 *if e.g.* $d = 1$. *We call such process runs* blocked. *One can ask now how to* repair *the process to make it data-aware sound, while at the same time keeping behavior unchanged as much as possible.*

In principle, two different approaches are conceivable to get rid of blocked runs as in Ex. I.1, without modifying the control flow: transition guards can be made more restrictive, so that the blocked run is no longer possible; we call this repair by *restricting behavior*. Alternatively, guards can be relaxed so that the blocked run can continue; we call this repair by *extending behavior*. This paper presents repair procedures based on both of these paradigms, and show that the imposed modifications are minimal w.r.t. process behavior. We also implemented the procedures, and performed preliminary experiments.

To summarize, our contributions are the following:

1) We propose two repair procedures that either *restrict* or *extend* process behavior to obtain data-aware soundness.
2) The results of these procedures are shown to be data-aware sound, and we prove that they modify the process behavior as little as possible (Props. III.1 and III.5).
3) These procedures are implemented on top of the tool ada. By a conformance checking experiment, we also verify that the repair procedure not only makes the process data-aware sound, but also does not harm conformance with actual behaviors in an event log, in the spirit of [5].

For clarity, we summarize also the limitations of our approach. First, our procedure makes two assumptions on the input DPN: (A1) The Petri net underlying the input DPN must be sound. This is, however, not a limitation since all data-aware DPNs must have a sound underlying Petri net. (A2) The DPN must be in a class where data-aware soundness can be checked, e.g., all guards must be variable-to-constant comparisons. In this particular case our procedure is guaranteed to terminate, but this need not hold in general (see Sec. III).

The remainder of the paper is structured as follows. In Sec. II, we recall preliminaries about DPNs and an approach to check data-aware soundness. In Sec. III, we present two repair procedures, based on either restricting or extending behavior. Their implementation and experiments are described in Sec. IV. In Sec. V, we conclude with a discussion and directions for future work.

II. Background

This section recaps background on DPNs and data-aware dynamic systems as process models, and constraint graphs.

We assume a set of *process variables* V, each of which is associated with a sort from the set $\Sigma = \{\mathtt{int}, \mathtt{rat}\}$ with associated domains $\mathcal{D}(\mathtt{int}) = \mathbb{Z}$ and $\mathcal{D}(\mathtt{rat}) = \mathbb{Q}$. For $\sigma \in \Sigma$, V_σ denotes the subset of variables of type σ. To manipulate variables, we consider expressions c, called *constraints*:

$$c := n{\geqslant}n \mid n{\neq}n \mid n{=}n \mid r{\geqslant}r \mid r{>}r \mid r{\neq}r \mid r{=}r \mid c{\wedge}c \mid c{\vee}c$$
$$n := v_i \mid k \mid n + n \mid -n \qquad\qquad r := v_r \mid q \mid r + r \mid -r$$

where $v_i \in V_{\mathtt{int}}$, $v_r \in V_{\mathtt{rat}}$, $k \in \mathbb{Z}$, and $q \in \mathbb{Q}$. Constraints will be used to capture conditions on the values of variables that are read and written during the execution of process activities. The set of constraints over a set of variables V is denoted $\mathcal{C}(V)$. We will also consider first-order formulas that have constraints as atoms. Since such formulas are over the theory of linear arithmetic, satisfiability is decidable; moreover, linear arithmetic is known to enjoy *quantifier elimination* [1]: if φ is a formula with atoms in $\mathcal{C}(V \cup \{x\})$, there is some φ' with free variables V that is logically equivalent to $\exists x.\varphi$. We denote logical equivalence by $\equiv$, and logical entailment by $\models$.

A. Data Petri nets.

We adopt the standard definition of DPNs [14], [16]. We consider two disjoint, marked copies of the set of process variables V, denoted $V^r = \{v^r \mid v \in V\}$ and $V^w = \{v^w \mid v \in V\}$, called the *read* and *write* variables. They are used to refer to variable values before and after executing a transition, respectively. We write $\overline{V}$ for a vector that contains the variables in V ordered in an arbitrary but fixed way, and $\overline{V}^r$ and $\overline{V}^w$ for vectors ordering V^r and V^w in the same way.

Definition II.1. A *data Petri net* (DPN) is a tuple $\mathcal{N} = \langle P, T, F, \ell, \mathcal{A}, V, \mathit{guard}\rangle$, where *(1)* $\langle P, T, F, \ell\rangle$ is a Petri net with non-empty, disjoint sets of places P and transitions T, a flow relation $F\colon (P \times T) \cup (T \times P) \mapsto \mathbb{N}$ and a labelling function $\ell\colon T \mapsto \mathcal{A}$, where $\mathcal{A}$ is a finite set of activity labels; *(2)* V is a finite set of process variables; and *(3)* $\mathit{guard}\colon T \mapsto \mathcal{C}(V^r \cup V^w)$ is a guard mapping.

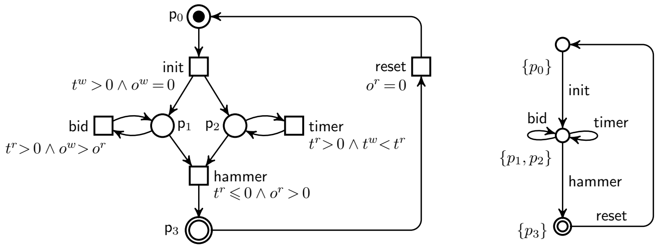


Fig. 1. A DPN (left) and a corresponding DDSA (right).

Example II.2. *The process in Fig. 1 (left) is a DPN modeling a simple auction process with variables $V = \{o, t\}$ of sort* `rat`*, where o holds the last offer issued by a bidder, and t is a timer. Action* init *fixes the timer t to a positive value and the offer o to 0; while the timer did not expire, it can be decreased (action* timer*), or bids can be issued, increasing the current offer (*bid*); the item can be sold if the timer expired and the offer is positive (*hammer*); and* reset *can restart the process if o is 0. The initial and final marking are $\{\mathsf{p}_0\}$ and $\{\mathsf{p}_3\}$, respectively, and we assume o and t have initial value 0.*

Also the process in Ex. I.1 is a DPN. The variables read and written by a transition t are denoted by $read(t) = \{v \mid v^r$ occurs in $guard(t)\}$ and $write(t) = \{v \mid v^w$ occurs in $guard(t)\}$. For instance, for t the activity labelled bid in Fig. 1, $write(t) = \{o\}$ and $read(t) = \{o, t\}$. An assignment with domain V is a *state variable assignment*, to distinguish it from a *transition variable assignment* β, which has as domain the annotated variables $V^r \cup V^w$.

A *state* in a DPN is a pair (M, α) of a marking $M\colon P \mapsto \mathbb{N}$ of the underlying Petri net, together with a state variable assignment α. It thus simultaneously describes the control flow progress and the current values of all variables, as specified by α. E.g., $(\{\mathsf{p}_0\}, \left[\begin{smallmatrix} t=0 \\ o=0 \end{smallmatrix}\right])$ is a state for the DPN of Ex. II.2.

Definition II.3 (Transition firing)**.** Transition $t \in T$ is *enabled* in (M, α) if a transition variable assignment β exists such that:

(i) $\beta(v^r) = \alpha(v)$ for every $v \in read(t)$, i.e., β is as α for read variables;

(ii) $\beta \models guard(t)$, i.e., β satisfies the guard; and

(iii) $M(p) \geqslant F(p, t)$ for every p such that $F(p, t) \geqslant 0$.

An enabled transition may *fire*, producing a new state (M', α'), s.t. $M'(p) = M(p) - F(p, t) + F(t, p)$ for every $p \in P$, and $\alpha'(v) = \beta(v^w)$ for every $v \in write(t)$, and $\alpha'(v) = \alpha(v)$ for every $v \notin write(t)$. Such a pair (t, β) is called (valid) *transition firing*, and we denote its firing by $(M, \alpha) \xrightarrow{(t,\beta)} (M', \alpha')$.

Given a DPN $\mathcal{N}$, we fix one state (M_I, α_I) as *initial*, where M_I is the initial marking of the underlying Petri net and α_I is a state variable assignment specifying the initial values of all variables in V. Similarly, we denote the final marking as M_F, and call *final* any state of the form (M_F, α_F) for some α_F. For instance, the net in Ex. II.2 admits a transition firing $(\{\mathsf{p}_0\}, \left[\begin{smallmatrix} t=0 \\ o=0 \end{smallmatrix}\right]) \xrightarrow{\mathsf{init}} (\{\mathsf{p}_1, \mathsf{p}_2\}, \left[\begin{smallmatrix} t=1 \\ o=0 \end{smallmatrix}\right])$ from its initial state, while $(\{\mathsf{p}_3\}, \left[\begin{smallmatrix} t=0 \\ o=5 \end{smallmatrix}\right])$ is a final state.

A state (M', α') is *reachable* in a DPN if there is a sequence of transition firings $(M_I, \alpha_I) \xrightarrow{(t_1,\beta_1)} \dots \xrightarrow{(t_n,\beta_n)} (M', \alpha')$. Such a sequence is also written $(M_I, \alpha_I) \to^* (M', \alpha')$ and called a *run*. Below, we assume that DPNs are *bounded*, i.e., the number of tokens in reachable markings is upper-bounded, and that there exists at least one run to a final state.

Definition II.4. A DPN is *data-aware sound* iff:

(P1) for all M, α such that $(M_I, \alpha_I) \to^* (M, \alpha)$ there is some α' such that $(M, \alpha) \to^* (M_F, \alpha')$, i.e., any run can be continued to a final state;

(P2) for all M, α such that $(M_I, \alpha_I) \to^* (M, \alpha)$ and $M \geqslant M_F$ it is $M = M_F$, i.e., termination is clean; and

(P3) for all $t \in T$ there is a sequence $(M_I, \alpha_I) \to^* (M, \alpha) \xrightarrow{(t,\beta)} (M', \alpha')$ for some M, M', α, α', and β, i.e., there are no dead transitions.

For instance, as described informally in Ex. I.1, the DPN given there is not data-aware sound. Specifically, it violates (P1) in Def. II.4, since there is a run to a state $(\{\mathsf{p}_7\}, \alpha)$ with $\alpha(d) = 1$. Also the DPN in Fig. 1 is not data-aware sound: it violates (P3) because the reset transition is unreachable as it requires o to be 0, but in marking $\{\mathsf{p}_3\}$ the value of o must be positive. It also violates (P3) because after the following run no further step is possible:

$$(\{\mathsf{p}_0\}, \left[\begin{smallmatrix} t=0 \\ o=0 \end{smallmatrix}\right]) \xrightarrow{\mathsf{init}} (\{\mathsf{p}_1, \mathsf{p}_2\}, \left[\begin{smallmatrix} t=1 \\ o=0 \end{smallmatrix}\right]) \xrightarrow{\mathsf{timer}} (\{\mathsf{p}_1, \mathsf{p}_2\}, \left[\begin{smallmatrix} t=0 \\ o=0 \end{smallmatrix}\right]) \quad (1)$$

B. Data-aware Dynamic Systems with Arithmetic.

To analyze DPNs, we will transform them into the simpler model of data-aware dynamic systems with arithmetic (DDSAs) [7], [13].

Definition II.5. A *DDSA* $\mathcal{B} = \langle B, b_I, \mathcal{A}, Tr, B_F, V, \alpha_I,$ $guard\rangle$ is a labeled transition system where (1) B is a finite set of *control states*, or simply states, with $b_I \in B$ the initial one; (2) $\mathcal{A}$ is a set of *actions*; (3) $Tr \subseteq B \times \mathcal{A} \times B$ is a *transition relation*; (4) $B_F \subseteq B$ are *final states*; (5) V is the set of *process variables*; (6) α_I the *initial variable assignment*; and (7) $guard\colon \mathcal{A} \mapsto \mathcal{C}(V^r \cup V^w)$ specifies *executability constraints* for actions over $V^r \cup V^w$.

Every bounded DPN $\mathcal{N}$ can be unfolded to an equivalent DDSA $\mathcal{B}$ over the same set of data variables, by taking reachable markings of $\mathcal{N}$ as states of $\mathcal{B}$, so that the initial and final markings become initial and final states, respectively. Actions, transition guards, and the initial valuation need not be modified; details can be found in the description of the procedure *DPNtoDDS* [11]. Fig. 1 (right) shows a DDSA which corresponds to the DPN in Fig. 1 (left). The action guards are the same as in the DPN, but omitted for readability.

We assume below that all actions of DDSA transitions are distinct, which can be ensured by adding copies of actions.

Let $\mathcal{B} = \langle B, b_I, \mathcal{A}, Tr, B_F, V, \alpha_I, guard\rangle$ be a given DDSA. If state $b \in B$ admits a transition to b' via action a, i.e., $(b, a, b') \in Tr$, we write $b \xrightarrow{a} b'$. A *state* of $\mathcal{B}$ is a pair (b, α) where $b \in B$ and α is an assignment with domain V. An action a leads from a state (b, α) to a new state (b', α') by updating the assignment α according to the action guard, exactly as in

DPNs: $\mathcal{B}$ *admits a step* from state (b,α) to (b',α') via action a, denoted $(b,\alpha) \xrightarrow{a} (b',\alpha')$, if $b \xrightarrow{a} b'$, $\alpha'(v) = \alpha(v)$ for all $v \in V \setminus \mathit{write}(a)$, and the transition assignment β given by $\beta(v^r) = \alpha(v)$ and $\beta(v^w) = \alpha'(v)$ for all $v \in V$, satisfies the guard of a, that is, $\beta \models \mathit{guard}(a)$.
A *run* ρ of a DDSA $\mathcal{B}$ is a sequence of steps

$$(b_I,\alpha_I) = (b_0,\alpha_0) \xrightarrow{a_1} (b_1,\alpha_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (b_n,\alpha_n)$$

We call the *abstraction* of a run the corresponding transition sequence $b_0 \xrightarrow{a_1} b_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} b_n$. For instance, for the DDSA in Fig. 1 (right), Eq. (1) can also be seen as a run of the DDSA, as each DPN marking is a state of the DDSA.

Data-aware soundness of DPNs can be directly checked on the corresponding DDSAs [11]. To that end, we say that a state (b,α) is a *blocked state* of a DDSA $\mathcal{B}$ if there is a run ρ: $(b_I,\alpha_I) \to^* (b,\alpha)$ but no derivation $(b,\alpha) \to^* (b_f,\alpha')$, for any $b_f \in B_F$ and α'. A run is called blocked if it ends in a blocked state, such as the run in Eq. (1).

Lemma II.6 ([11, Lem. 1]). Given a DPN $\mathcal{N}$ and $\mathcal{B} = \mathit{DPNtoDDS}(\mathcal{N})$ with control states B,

- $\mathcal{N}$ satisfies (P1) iff $\mathcal{B}$ has no blocked states,
- $\mathcal{N}$ satisfies (P2) iff all $b \in B$ that corresponding to a marking M with $M > M_F$ are unreachable in $\mathcal{B}$, and
- $\mathcal{N}$ satisfies (P3) iff for all transitions $t \in T$ of $\mathcal{N}$ there are some $b, b' \in B$ s.t. $(b, \ell(t), b') \in Tr$ is reachable.

As remarked above, the DPN from Fig. 1 violates (P1), and indeed the respective DDSA has a blocked state (see Eq. (1)).

C. Constraint Graphs

Constraint graphs are (in the lucky case, finite) abstractions of the state spaces of DDSAs, they were introduced to check data-aware soundness [11]. We recapitulate the main idea:

Let $\mathcal{B} = \langle B, b_I, \mathcal{A}, Tr, B_F, V, \alpha_I, \mathit{guard}\rangle$ be a given DDSA. The *transition formula* Δ_a of action a is given by $\Delta_a(\overline{V}^r, \overline{V}^w) = \mathit{guard}(a) \wedge \bigwedge_{v \notin \mathit{write}(a)} v^w = v^r$. This formula simply expresses conditions on variables *before and after* executing the action: $\mathit{guard}(a)$ must hold, and the values of all variables that are not written are copied. E.g., for action bid in Fig. 1 (right), we have $\mathit{guard}(\mathsf{bid}) = (t^r > 0) \wedge (o^w > o^r)$, thus $\mathit{write}(\mathsf{bid}) = \{o\}$, so $\Delta_{\mathsf{bid}} = (t^r > 0) \wedge (o^w > o^r) \wedge (t^w = t^r)$.

Every node of a constraint graph contains a first-order formulas φ with free variables V to describe a set of variable assignments. We use the transition formula Δ_a to update φ in order to register the effects of executing a, as follows:

Definition II.7. For a formula φ with free variables V and action a, $\mathit{update}(\varphi, a) = \exists \overline{U}.\varphi[\overline{U}/\overline{V}] \wedge \Delta_a[\overline{U}/\overline{V}^r, \overline{V}/\overline{V}^w]$, where U is a set of variables that has the same cardinality as V and is disjoint from all variables in φ.

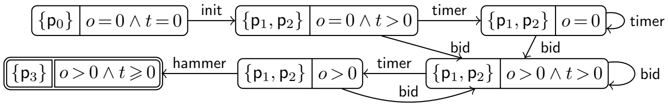
Here, $\varphi[\overline{U}/\overline{V}]$ is the result of replacing variables $\overline{V}$ in φ by $\overline{U}$, and similar for Δ_a. For instance, if $\overline{V} = (o,t)$ we can take the renamed variables $\overline{U} = (o',t')$; hence for $\varphi = (t > 0) \wedge (o = 0)$ we then get $\mathit{update}(\varphi, \mathsf{bid}) = \exists o'\, t'.(t' > 0) \wedge (o' = 0) \wedge (o > o') \wedge (t = t')$, which is equivalent to $(t > 0) \wedge (o > 0)$. In fact, a logically equivalent quantifier-free formula can always be obtained by quantifier elimination.

Given any assignment α, we denote by c_α the formula $\bigwedge_{v \in V} v = \alpha(v)$ which serves to encode α as a formula.

Definition II.8. A *constraint graph* $\mathrm{CG}_{\mathcal{B}}(b_0, \alpha)$ for $\mathcal{B}$, a state $b_0 \in B$, and assignment α is a triple $\langle S, s_0, \gamma\rangle$. The node set S consisting of pairs (b, φ) for $b \in B$ and a formula φ, and the set of edges $\gamma \subseteq S \times \mathcal{A} \times S$ are inductively defined as follows:
(i) $s_0 = (b_0, c_\alpha) \in S$ is the initial node; and
(ii) if $(b, \varphi) \in S$ and $b \xrightarrow{a} b'$ such that $\mathit{update}(\varphi, a)$ is satisfiable, there is some $(b', \varphi') \in S$ with $\varphi' \equiv \mathit{update}(\varphi, a)$, and $(b, \varphi) \xrightarrow{a} (b', \varphi')$ is in γ.

Intuitively, the constraint graph symbolically describes all states reachable in $\mathcal{B}$: every node combines a DDSA state b with a formula φ with free variables V, representing all states (b, α) of the DDSA s.t. α satisfies φ. We denote by $\mathrm{CG}_{\mathcal{B}}$ the constraint graph $\mathrm{CG}_{\mathcal{B}}(b_I, \alpha_I)$ starting at the initial state of $\mathcal{B}$.

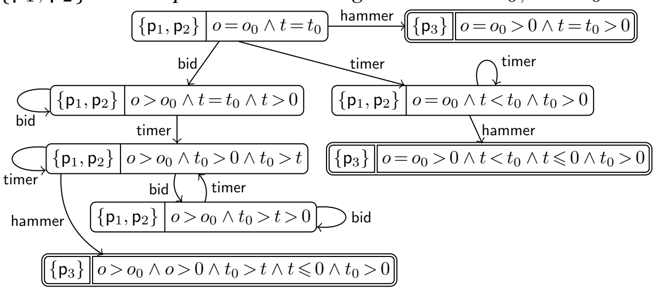
Example II.9. *The* $\mathrm{CG}_{\mathcal{B}}$ *for the DDSA in Ex. II.2 is as follows:*



For instance, the top-left node represents the initial state: the marking is $\{\mathsf{p}_0\}$ *and both variables have value 0. The node* $(\{\mathsf{p}_1, \mathsf{p}_2\}, o = 0)$ *represents the infinite set of states where the marking is* $\{\mathsf{p}_1, \mathsf{p}_2\}$*, o has value 0, and t is unconstrained. Nodes with final control state are drawn with a double border.*

The soundness checking procedure from [11] also needs constraint graphs that start from an arbitrary control state $b \in B$ (rather than the initial control state of the DDSA) and that, instead of assigning variables V their initial values as in the example above, only impose that they have as values some fresh placeholder variables V_0. We denote such a graph by $\mathrm{CG}_{\mathcal{B}}(b)$. Instead of giving the formal definition which is similar to Def. II.8 (cf. [11, Def. 7]), we show an example.

Example II.10. *The following is the constraint graph* $\mathrm{CG}_{\mathcal{B}}(\{\mathsf{p}_1, \mathsf{p}_2\})$ *for the DDSA in Fig. 1 (right), starting at* $\{\mathsf{p}_1, \mathsf{p}_2\}$ *with a placeholder assignment* $o \mapsto o_0, t \mapsto t_0$*:*



Again, every state can be seen as a representation of a set of states, which are, however, parameterised by the initial values o_0 and t_0 of the variables o and t.

The crucial property of a CG that it faithfully and completely represent the state space, as formally stated next by relating paths in the CG to DDSA runs (cf. [11, Lem. 2]). Parts

(1) and (2) basically express the same property, but the former is for the version of CGs that start at the initial state, while the latter is for CGs starting with a parametric assignment.

Lemma II.11. (1) $\mathrm{CG}_{\mathcal{B}}$ has a path $\pi\colon (b_I, C_{\alpha_I}) \xrightarrow{*} (b, \varphi)$ where φ is satisfiable by α, iff $\mathcal{B}$ has a run $(b_I, \alpha_I) \xrightarrow{*} (b, \alpha)$ whose abstraction is $\sigma(\pi)$.
(2) $\mathrm{CG}_{\mathcal{B}}(b)$ has a path $\pi\colon (b, C_{\alpha_0}) \xrightarrow{*} (b', \varphi)$ s.t. φ is satisfiable by α, iff $\mathcal{B}$ has derivation $(b, \alpha_0) \xrightarrow{*} (b', \alpha_n)$ abstracted by $\sigma(\pi)$ with $\alpha_0 = \alpha|_{V_0}$ and $\alpha_n = \alpha|_V$.

Here, for a path π in the CG, $\sigma(\pi)$ is the DDSA transition sequence along this path. The lemma thus states that for every path in the CG ending in a node (b, φ) and every satisfying assignment α of φ, there is a run of the DDSA with the same transition sequence to a state (b, α), and vice versa.

CGs are infinite in general, but it has been shown that for many classes of DDSAs relevant in practice, finite CGs can be computed [11]. These include DDSAs where all constraints are variable-to-variable/constant comparisons over $\mathbb{Q}$ like Ex. II.2, and DDSAs that, intuitively, maintain only a bounded amount of information (this holds e.g. for Ex. I.1).

Finally, we explain how CGs can be used for soundness checking: from $\mathrm{CG}_{\mathcal{B}}(b)$, one can extract a *condition on* V_0 to reach a final state from b, as follows. For Φ the set of formulas occurring in final nodes of $\mathrm{CG}_{\mathcal{B}}(b)$, the formula $\exists V. \bigvee_{\varphi \in \Phi} \varphi$ expresses a condition on V_0 to reach a final state from b. For convenience, we consistently rename V_0 to V in this formula:

$$\mathit{reach\_final}(b) := (\exists V. \bigvee\nolimits_{\varphi \in \Phi} \varphi)[\overline{V}]$$

For instance from the CG in Ex. II.10, we obtain $\mathit{reach\_final}(\{\mathsf{p}_1, \mathsf{p}_2\}) = (t > 0)$. This means that from the marking $\{\mathsf{p}_1, \mathsf{p}_2\}$ a final state can be reached if and only if the current value of t is positive.

Finally, from the proof of [11, Thm. 1], one can directly derive the following procedure for detecting blocked states:

Proposition II.12. *If there exists in* $\mathrm{CG}_{\mathcal{B}}$ *a path* π *to a node* (b, φ) *such that* $\varphi \wedge \neg \mathit{reach\_final}(b)$ *is satisfied by* α*, then the run that is abstracted by* $\sigma(\pi)$ *and ends in* (b, α) *is blocked.*

In such a situation we also call the node (b, φ) *blocked*. Moroever, from Lemma II.6 and by construction (Def. II.8):

Proposition II.13. *Given a DPN* $\mathcal{N}$ *and* $\mathcal{B} = \mathit{DPNtoDDS}(\mathcal{N})$*, if for a transition* $t \in T$ *in* $\mathcal{N}$ *there is no* $(b, \varphi) \xrightarrow{a} (b', \varphi')$ *in* $\mathrm{CG}_{\mathcal{B}}$ *with* $a = \ell(t)$*, then* t *is a dead transition in* $\mathcal{N}$*.*

III. Soundness Repair

We now turn to address the task of repairing data-aware unsoundness of a DPN $\mathcal{N}$, though we will from now on work on its corresponding DDSA $\mathcal{B}$. As motivated in Section I, we assume $\mathcal{N}$ is classically sound, i.e., ignoring data.This implies that $\mathcal{B}$ must satisfy condition (P2) in Lem. II.6, but it can violate conditions (P1) (no blocked states) and (P3) (no dead transitions). As stated in the introduction, our goal is to modify the behavior of the input DPN $\mathcal{N}$ as little as possible.

Algorithm 1 Soundness repair by restriction

```
procedure $\mathit{repairRestrict}(\mathcal{B})$
    $\mathcal{G} \leftarrow \mathrm{CG}_{\mathcal{B}}$
    while $\mathit{hasBlockedState}(\mathcal{B}, \mathcal{G})$ do
        $\mathcal{B} \leftarrow \mathit{restrictBlockedState}(\mathcal{B}, \mathcal{G})$
        $\mathcal{G} \leftarrow \mathrm{CG}_{\mathcal{B}}$
        $\mathcal{B} \leftarrow \mathit{elimDeadTransitions}(\mathcal{B}, \mathcal{G})$
    return $\mathcal{B}$

procedure $\mathit{restrictBlockedState}(\mathcal{B}, \mathcal{G})$
    choose path $\pi$ to $(b, \varphi)$ in $\mathcal{G}$ such that $\varphi \wedge \neg \mathit{reach\_final}(b)$
      is satisfiable
    let $a$ be the last action in $\pi$
    $\psi \leftarrow \mathit{reach\_final}(b)$
    $\mathcal{B}' \leftarrow \mathcal{B}$ with $\mathit{guard}(a) := \mathit{guard}(a) \wedge \mathit{tvars}_a(\psi)$
    return $\mathcal{B}'$
```

To handle dead transitions, several approaches are conceivable: (i) one can remove them, (ii) one can relax their guards to force them to become enabled at least by one run, or (iii) one can modify other guards in the DPN. Although all these options are viable in principle, (ii) has an impact on the relation between these transitions and the data dimension, potentially radically altering the process, whereas we regard (iii) achievable only within a user-guided interactive procedure, as it is not possible to establish a single principled way of altering data conditions that would fit different scenarios. Therefore, the algorithms we illustrate below follow option (i) and eliminate all dead transitions.

To handle a blocked state, we consider two approaches: either the behavior of the process can be restricted so that a blocked run can no longer occur, or the behavior of the process can be extended so that a blocked run is no longer blocked. It depends on the context which modification is more appropriate, and should be decided by a domain expert. Below, we propose procedures for both approaches. A third approach that freely interleaves these two types of repair actions is also imaginable, but not discussed here explicitly.

In both cases, we suppose that the initial state (b_I, α_I) of the DDSA is not blocked, which follows from our assumption that the DDSA has at least one run to a final state.

A. Repair by Restricting Behavior

The procedure *repairRestrict* in Alg. 1 describes our first repair procedure, where the following subroutines are used:

- *elimDeadTransitions* first removes from $\mathcal{B}$ all dead transitions (detected on $\mathrm{CG}_{\mathcal{B}}$ – see Prop. II.13), then prunes all states of $\mathcal{B}$ that becomes unreachable), and
- $\mathit{tvars}_a(\psi)$ replaces a variable v in ψ by v^w if $v \in \mathit{write}(a)$, and v^r otherwise, for all $v \in V$.

The procedure works as follows: it first computes the constraint graph of $\mathcal{B}$, and uses it to check whether $\mathcal{B}$ has a blocked state (using Prop. II.12). If this is the case, *restrictBlockedState* is called to resolve this blocked state. In this subroutine, a node (b, φ) is selected from the CG to witness the blocked state, and we can assume that it is reached on a path with last transition a. The idea is now to "deactivate"

a whenever the run would get blocked in b, or to avoid problematic variable updates in case a writes any variables. In the DDSA returned by *restrictBlockedState*, blocked states with state b are thus resolved. Procedure *repairRestrict* then computes the new CG, removes dead transitions, and repeats these steps until $\mathcal{B}$ has no more blocked states. Note that *restrictBlockedState* is non-deterministic as the choice in line 9 need not be unique.

In general, constraint graph computations need not terminate, though they do for a wide range of DDSAs that model DPNs in practice. The next proposition shows that if constraint graphs can be computed, *repairRestrict* terminates and the resulting DDSA is data-aware sound; and by items (3) and (4) the modifications are as conservative with respect to behaviour as possible, in the sense that only blocked runs are removed.

Proposition III.1. *(1) If constraint graph computations terminate then repairRestrict terminates.*
(2) If $\mathcal{B}' = repairRestrict(\mathcal{B})$ then $\mathcal{B}'$ is data-aware sound.
(3) All runs of $\mathcal{B}'$ are runs of $\mathcal{B}$.
(4) All runs of $\mathcal{B}$ that are not runs of $\mathcal{B}'$ are blocked in $\mathcal{B}$.

Proof. (1) Every call of *restrictBlockedState* modifies the guard of some action a by adding ψ. To see that this change can be applied at most once to every action, suppose towards a contradiction that $guard(a) \models tvars_a(\psi)$, so $guard(a) \models tvars_a(reach\_final(b))$. We have:
$$\varphi \equiv update(\varphi', a) \equiv \exists \overline{U}.\varphi'[\overline{U}/\overline{V}] \wedge \Delta_a[\overline{U}/\overline{V}^r, \overline{V}/\overline{V}^w]$$
for some φ', so by reverting the renaming *tvars* this means that $\varphi \models reach\_final(b)$. This contradicts that $\varphi \wedge \neg reach\_final(b)$ is satisfiable.
(2) The loop is repeated as long as $\mathcal{B}$ has blocked states, and in every loop iteration dead transitions are removed. Thus, when *repairRestrict* returns, $\mathcal{B}$ must be data-aware sound.
(3) Clear, because $\mathcal{B}'$ has more restrictive guards than $\mathcal{B}$.
(4) We show that if $\mathcal{B}_{i+1} = restrictBlockedState(\mathcal{B}_i)$ for some $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$, and ρ is a run of $\mathcal{B}_i$ but not of $\mathcal{B}_{i+1}$, then ρ is blocked in $\mathcal{B}_i$, from which the claim follows by induction. Let the node (b, φ) be the one chosen in line 9, and a the action in line 10. Then ρ must use a, so be of the form
$$\rho\colon (b_0, \alpha_0) \xrightarrow{*} (b_{i-1}, \alpha_{i-1}) \xrightarrow{a_i} (b_i, \alpha_i) \xrightarrow{*} (b_n, \alpha_n)$$
for some $i \geqslant 0$ where $a_i = a$ is the first occurrence of a in ρ such that $\alpha_i \not\models reach\_final(b_i)$; such an occurrence must exist because ρ is not a run of $\mathcal{B}'$. We must have $b_i = b$ since action labels are unique in DDSAs. By Lem. II.11 there is a path π to a node (b, φ') in $\mathrm{CG}_{\mathcal{B}}$ such that $\sigma(\pi)$ is the abstraction of $\rho|_i$ and $\alpha_i \models \varphi'$. Thus $\alpha_i \models \varphi' \wedge \neg reach\_final(b)$, so by Prop. II.12, ρ is blocked. □

We illustrate the procedure on two examples.

Example III.2. *The CG for Ex. II.2 shown in in Ex. II.9 has a node $(\{\mathsf{p}_1, \mathsf{p}_2\}, o = 0)$, but reachability of a final state from $\{\mathsf{p}_1, \mathsf{p}_2\}$ is characterized by $\psi := reach\_final(\{\mathsf{p}_1, \mathsf{p}_2\}) = (t > 0)$. Clearly, $(o = 0) \wedge \neg(t > 0)$ is satisfiable, e.g. by an assignment α that assigns 0 to all variables, cf. the last assignment in run (1). Following Alg. 1, we can add a conjunct $tvars_{\mathsf{timer}}(\psi)$ to $guard(\mathsf{timer})$. We have $tvars_{\mathsf{timer}}(\psi) = (t^w > 0)$, so the new guard of* timer *is $(t^r > 0) \wedge (t^w < t^r) \wedge (t^w > 0)$. Note that at this point run (1) is no longer possible. The DDSA has no more blocked states, and after removing the dead transition* reset*, the DDSA is data-aware sound.*

Algorithm 2 Soundness repair by extension

```
procedure $repairExtend(\mathcal{B})$
    $\mathcal{G} \leftarrow \mathrm{CG}_{\mathcal{B}}$
    while $hasBlockedState(\mathcal{B}, \mathcal{G})$ do
        $\mathcal{B} \leftarrow extendBlockedState(\mathcal{B}, \mathcal{G})$
        $\mathcal{G} \leftarrow \mathrm{CG}_{\mathcal{B}}$
    $\mathcal{B} \leftarrow elimDeadTransitions(\mathcal{B}, \mathcal{G})$
    return $\mathcal{B}$

procedure $extendBlockedState(\mathcal{B}, \mathcal{G})$
    choose a path $\pi$ to some $(b, \varphi)$ in $\mathcal{G}$ such that
      $\varphi \wedge \neg reach\_final(b)$ is satisfiable and there are $a$, $b'$, and
      a non-blocked node $(b', \varphi')$ in $\mathcal{G}$ such that $b \xrightarrow{a} b'$ in $\mathcal{B}$
    $\psi \leftarrow \neg reach\_final(b)$
    $keep_a \leftarrow \bigwedge_{v \in write(a)} v^r = v^w$
    $\mathcal{B}' \leftarrow \mathcal{B}$ with $guard(a) := guard(a) \vee (rvars(\psi) \wedge keep_a)$
    return $\mathcal{B}'$
```

Example III.3. *The CG of the DDSA of Ex. I.1 contains a node (p_7, φ) where $\varphi = (dp = 0) \wedge (d \geqslant 0) \wedge (p \geqslant 0) \wedge (a \geqslant 0) \wedge (t \geqslant 0) \wedge (e \geqslant 0) \wedge (0 \leqslant ds < 2160) \wedge (0 \leqslant dj < 1440)$. However, the reachability of a final state from p_7 is given by $reach\_final(\mathsf{p}_7) = (d = 0) \vee (d = 2)$. Thus $\varphi \wedge \neg reach\_final(\mathsf{p}_7)$ is satisfiable, which reflects the blocked run mentioned in Ex. I.1, with last action a :=* send to prefecture*. According to Alg. 1, this is repaired by adding a conjunct $tvars_a(reach\_final(\mathsf{p}_7))$ to its guard. Since d is written by this transition, we add the write guard $d^w = 0 \vee d^w = 2$. In a second loop iteration of repairRestrict, a similar blocked state with p_5 is detected, for d being different from 0 and 1. We thus extend the guard of* appeal to judge *with conjunct $d^w = 0 \vee d^w = 1$, after which the DDSA is data-aware sound.*

Note that in *repairRestrict*, constraint graphs must be recomputed since the DDSA changes. Even if the constraint graph for the DDSA of the input DPN is finite, it is in general not clear whether this also holds for repaired DDSAs. However, we point out one relevant case where this is guaranteed: *MC-DDSAs* are DDSAs where all guards are boolean combinations of *monotonicity constraints* (MCs), i.e., variable-to-variable and variable-to-constant comparisons, such DDSAs have finite CGs [11]. For instance, Ex. II.2 is a MC-DDSA. For MC-DDSAs, all formulas in CGs are also conjunctions of MCs. Hence, also all guards in repaired DDSAs as returned by *repairRestrict* are MCs, so that the repaired DDSA has again a finite CG. We thus have:

Remark III.4. *repairRestrict terminates on MC-DDSAs.*

B. Repairing Deadlocks by Extending Behavior

We next consider the approach based on extending behavior by executing the procedure *repairExtend* shown in Alg. 2. Here the subroutine $rvars(\psi)$ replaces all variables v in ψ by v^r. Note that this only adds tests on the current values of variables.

The procedure *repairExtend* works similarly to *repairRestrict*, with the difference that once a blocked state corresponding to node (b, φ) in the constraint graph is found, we modify an *outgoing* transition from b by relaxing its guard, namely this transition is allowed for all configurations that were previously blocked, i.e., that satisfy $\neg$*reach_final*(b). We add two technical remarks: In line 9, a state b' with $b \xrightarrow{a} b'$ must exist because the underlying Petri net is sound. The condition that there is a node (b', φ') that is not blocked is needed for termination; such a choice is always possible, intuitively by taking a maximal path to a blocked state. The subformula $keep_a$ defined in line 11 is added to ensure that the written variables in all disjuncts of $guard(a)$ coincide. We next show correctness of *repairExtend*, as well as that the modifications are maximally conservative with respect to behaviour, in the sense that only blocked runs are extended.

Proposition III.5. *(1) If constraint graph computations terminate then repairExtend terminates.*
(2) If $\mathcal{B}'$ = repairExtend($\mathcal{B}$) then $\mathcal{B}'$ is data-aware sound.
(3) All runs of $\mathcal{B}$ are runs of $\mathcal{B}'$.
(4) All runs of $\mathcal{B}'$ that are not runs of $\mathcal{B}$ have as prefix a run that is blocked in $\mathcal{B}$.

Proof. (1) Every call of *extendBlockedState* adds to the guard of an action a a disjunct $rvars($*reach_final*$(b))$ so that a node (b, φ) is no longer blocked. If the DDSA and its constraint graph are changed in later iterations of the loop in *repairExtend*, guards become more relaxed, so that (b, φ) remains unblocked. Thus, by the precondition in line 9, every action is modified at most once.

(2) The loop in *repairExtend* is repeated as long as $\mathcal{B}$ has blocked states and dead transitions are removed in the end, so the returned DDSA must be data-aware sound.

(3) Clear, because $\mathcal{B}'$ has more relaxed guards than $\mathcal{B}$.

(4) We show that if $\mathcal{B}_{i+1}$ = *extendBlockedState*$(\mathcal{B}_i)$ for some $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$ and ρ is a run of $\mathcal{B}_{i+1}$ but not of $\mathcal{B}_i$, then ρ has a prefix ρ' that is blocked in $\mathcal{B}_i$. Then the claim follows by induction. Let (b, φ) be the node chosen in line 9, and a the respective action. Then ρ must be of the form $\rho\colon (b_0, \alpha_0) \xrightarrow{*} (b_{i-1}, \alpha_{i-1}) \xrightarrow{a_i} (b_i, \alpha_i) \xrightarrow{*} (b_n, \alpha_n)$ for some $i \geqslant 0$ where $a_i = a$ is the first occurrence of a in ρ such that $\alpha_i \not\models$ *reach_final*(b_i); such an occurrence must exist because ρ is not a run of $\mathcal{B}$. Moreover, we must have $b_i = b$ since action labels are unique. Given the run above, by Lem. II.11 there is a path π to (b, φ') in $\mathrm{CG}_{\mathcal{B}}$ s.t. $\sigma(\pi)$ abstracts of $\rho|_i$ and $\alpha_i \models \varphi'$. So $\alpha_i \models \varphi' \wedge \neg$*reach_final*$(b)$, hence by Prop. II.12, $\rho|_i$ is blocked. □

As for the previous algorithm, *repairExtend* is not deterministic: different repairs are possible by choosing different nodes and actions a in line 9. We illustrate *repairExtend* on two examples.

Example III.6. *We consider again Ex. II.2, and resolve the blocked state by adding behavior to the DDSA, We have $\neg$reach_final$(\{\mathsf{p}_1, \mathsf{p}_2\}) = (t \leqslant 0)$, so we can e.g. set guard$(\mathsf{hammer}) = (t^r \leqslant 0 \wedge o^r > 0) \vee (t^r \leqslant 0) \equiv (t^r \leqslant 0)$. This resolves all blocked states; again the dead transition* reset *must be removed afterwards.*

Example III.7. *Also Ex. I.1 can be repaired by extension instead of restriction. Given the CG node (p_7, φ) mentioned in Ex. III.3, where reach_final$(\mathsf{p}_7) = (d = 0) \vee (d = 2)$, we can e.g. relax the guard of τ_6 to $d = 2 \vee \neg((d = 0) \vee (d = 2)) \equiv (d = 2) \vee (d \neq 0)$. Subsequently, the guard of τ_5 can be relaxed to $(d = 0) \vee (d \neq 1)$. At this point, the modified process is data-aware sound.*

Since for MC-DDSAs all formulas in constraint graphs are conjunctions of MCs, also all guards in repaired DDSAs as returned by *repairExtend* are MCs, for reasons as mentioned before Rem. III.4, so computed CGs are again finite. Hence,

Remark III.8. *repairExtend terminates on MC-DDSAs.*

IV. Implementation and Experiments

In this section, we describe the implementation of our repair approach, and its evaluation in preliminary experiments.

A. Implementation

We implemented our soundness repair procedure on top of the open-source tool ada[1] (arithmetic DDS analyzer), which has already functionality to check soundness [11]. It is a simple command line tool written in Python, but a web interface is available as well.[2] The tool takes as input a DPN in pnml format, or a DDSA in a simple json format; the latter is described in the tool documentation. Both repair procedures *repairRestrict* and *repairExtend* are implemented as described in Alg. 1 and 2. As output, ada reports dead transitions, dead states, modified guards, and the modified DDSA.

B. Experiments

To assess the practicality of our approach, we performed (a) an experiment where we applied the repair procedures to DPNs from the literature to check feasibility, and (b) an experiment to understand how repair influences conformance with a log.

Process repair experiment. We tested our repair procedure on DPNs from the literature, as well as some synthetic examples. Tab. I reports the results. DPNs (a)–(e) are all those DPNs that were recognized as not data-aware sound in [11]. For both *repairRestrict* and *repairExtend*, we list the time required by the repair procedure in seconds (*time*) and the number of loop iterations (*its*), respectively. Example (d) has zero loop iterations because it has dead transitions but no blocked states. In general, we found that *repairExtend* requires slightly more time than *repairRestrict*. This is likely because the computation time is dominated by the time needed to compute constraint graphs. The complexity of CGs depends on the amount of allowed behavior, so that CGs for DDSAs produced by *repairExtend* tend to be larger.

Further details about the experiments such as the repaired processes, can be found on an accompanying website.[3]

[1] https://gitlab.inf.unibz.it/SarahMaria.Winkler/ada

[2] https://soundness.adatool.dev/

[3] https://soundness.adatool.dev/repair.html

process	*repairRestrict*		*repairExtend*	
	time	*its*	*time*	*its*
(a) road fines normative [16, Fig. 7]	50s	2	71s	2
(b) road fines mined [14, Fig. 12.7]	24s	1	22s	1
(c) dig. whiteboard/transfer [14, Fig. 14.3]	2.1s	1	3s	1
(d) package handling [8, Fig. 5]	6s	0	6s	0
(e) auction [10, Ex. 1.1]	8s	1	20s	1
(f) auction from Ex. II.2	2.5s	1	2.7s	1
(g) livelock example	2.1s	1	2.4s	1

TABLE I
REPAIR EXPERIMENTS

Conformance checking experiment. As already commented, the main use case of our approach is within a process discovery pipeline in which a model is discovered from an event log and afterwards repaired in case it is not data-aware sound. It is therefore desirable that the repaired model still allows as many traces in the event log as possible, i.e., that the conformance score is not affected, in the spirit of [5].

We thus performed a conformance checking experiment with a DPN that is not data-aware sound and where an event log is available: The DPN (a) from Tab. I was mined from an event log of 150370 traces [16], exploiting both fully automatic discovery techniques and expert domain knowledge. To understand whether the repaired model still allows to replay the behavior recorded in the event log, we checked how conformance of the repaired processes compares to that of the original DPN. For the experiment, the data-aware conformance checker cocomot [6] was run on the above-mentioned log with (1) the original DPN, (2) the DPN obtained from *repairRestrict*, as described in Ex. III.3, and (3) the DPN obtained from *repairExtend*, as described in Ex. III.7, and we compared the average cost of an optimal alignment: This was (1) 1.2009 for the original DPN, (2) 1.2009 for the result of *repairRestrict*, and (3) 1.1305 for the result of *repairExtend*. Thus, for the DPN repaired by *repairRestrict*, conformance remains the same (in fact, not a single trace is aligned differently), so the repair did not harm conformance. Since *repairRestrict* restricts behaviour, it is impossible that the average distance decreases, so this is the best possible result that one can hope for. On the other hand, for the DPN repaired by *repairExtend*, conformance effectively improves as the average distance decreases. We see this as an experimental confirmation of Props. III.1 and III.5 that our repair procedure is maximally conservative with respect to behavior.

V. CONCLUSION

In this paper two procedures to repair violations of data-aware soundness in DPNs were presented, which are implemented on top of ada, and evaluated on some examples. We provide proofs, as well as some experimental evidence, that the modifications are minimal as far as behavior is concerned. Our procedure is based on three design decisions that we find reasonable, although they are surely not the only possible ones. 1) The procedure is *automatic* in that no user feedback is required. While this minimizes the requirements on user expertise, it is conceivable that an interactive procedure allowing expert user feedback could lead to better results for some applications. 2) We follow the paradigm of *behavioural conservatism*, and show that the set of accepted process runs is unchanged as much as possible. Other approaches are conceivable, e.g. one could aim at minimizing the number of guard changes or their syntactic distance. 3) Our procedure is *conservative with respect to the control flow*, motivated by the common two-phase discovery approach. In some applications it might be preferable to repair control flow and data in a more integrated way. We leave a procedure that holistically repairs violations on the data and control flow levels for future work. This also connects to the development of integrated techniques for data-aware process discovery, which instead of applying a two-phase approach [12], [15], discover both dimensions at once. We also aim to study how the ideas and techniques presented here can be integrated with frameworks for control-flow process repair based on event logs [5], towards log-driven, data-aware process repair.

REFERENCES

[1] Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, 2^{nd} Ed., vol. 336, pp. 1267–1329. IOS Press (2021)

[2] Batoulis, K., Haarmann, S., Weske, M.: Various notions of soundness for decision-aware business processes. In: Proc. 36th ER. LNCS, vol. 10650, pp. 403–418 (2017)

[3] de Leoni, M., Mannhardt, F.: Decision discovery in business processes. In: Encyclopedia of Big Data Technologies, pp. 1–12. Springer (2018)

[4] De Smedt, J., vanden Broucke, S.K.L.M., Obregon, J., Kim, A., Jung, J., Vanthienen, J.: Decision mining in a broader context: An overview of the current landscape and future directions. In: Business Process Management Workshops. Lecture Notes in Business Informatics Processing, vol. 281, pp. 197–207 (2016)

[5] Fahland, D., van der Aalst, W.M.P.: Model repair - aligning process models to reality. Inf. Syst. **47**, 220–243 (2015)

[6] Felli, P., Gianola, A., Montali, M., Rivkin, A., Winkler, S.: Data-aware conformance checking with SMT. Inf. Syst. **117**, 102230 (2023)

[7] Felli, P., de Leoni, M., Montali, M.: Soundness verification of decision-aware process models with variable-to-variable conditions. In: Proc. 19th ACSD. pp. 82–91. IEEE (2019)

[8] Felli, P., de Leoni, M., Montali, M.: Soundness verification of data-aware process models with variable-to-variable conditions. Fundamenta Informaticae **182**(1), 1–29 (2021)

[9] Felli, P., Montali, M., Winkler, S.: CTL* model checking for data-aware dynamic systems with arithmetic. In: Proc. 11th IJCAR. LNCS, vol. 13385, pp. 36–56 (2022)

[10] Felli, P., Montali, M., Winkler, S.: Linear-time verification of data-aware dynamic systems with arithmetic. In: Proc. 36th AAAI. pp. 5642–5650. AAAI Press (2022)

[11] Felli, P., Montali, M., Winkler, S.: Soundness of data-aware processes with arithmetic conditions. In: Proc. 34th CAiSE. LNCS, vol. 13295, pp. 389–406 (2022)

[12] Leno, V., Dumas, M., Maggi, F.M., La Rosa, M., Polyvyanyy, A.: Automated discovery of declarative process models with correlated data conditions. Inf. Syst. **89**, 101482 (2020)

[13] de Leoni, M., Felli, P., Montali, M.: Strategy synthesis for data-aware dynamic systems with multiple actors. In: Proc. 17th KR. pp. 315–325 (2020)

[14] Mannhardt, F.: Multi-perspective Process Mining. Ph.D. thesis, Technical University of Eindhoven (2018)

[15] Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Decision mining revisited - discovering overlapping rules. In: Proc. 28th CAiSE. LNCS, vol. 9694, pp. 377–392. Springer (2016)

[16] Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. Computing **98**(4), 407–437 (2016)

[17] van der Aalst, W.M.P.: The application of Petri Nets to workflow management. J. Circuits Syst. Comput. **8**(1), 21–66 (1998)