Enhancing workflow-nets with data for trace completion

Riccardo De Masellis[1], Chiara Di Francescomarino[1],
Chiara Ghidini[1], and Sergio Tessaris[2]

[1] FBK-IRST, Italy
{r.demasellis,dfmchiara,ghidini}@fbk.eu
[2] Free University of Bozen-Bolzano, Italy
tessaris@inf.unibz.it

Abstract. The growing adoption of IT-systems for modeling and executing (business) processes or services has thrust the scientific investigation towards techniques and tools which support more complex forms of process analysis. Many of them, such as conformance checking, process alignment, mining and enhancement, rely on *complete* observation of past (tracked and logged) executions. In many real cases, however, the lack of human or IT-support on all the steps of process execution, as well as information hiding and abstraction of model and data, result in incomplete log information of both data and activities. This paper tackles the issue of automatically repairing traces with missing information by notably considering not only activities but also data manipulated by them. Our technique recasts such a problem in a reachability problem and provides an encoding in an action language which allows to virtually use any state-of-the-art planning to return solutions.

1 Introduction

The use of IT systems for supporting business activities has brought to a large diffusion of *process mining* techniques and tools that offer business analysts the possibility to observe the current process execution, identify deviations from the model, perform individual and aggregated analysis on current and past executions.

According to the process mining manifesto, all these techniques and tools can be grouped in three basic types: process discovery, conformance checking and process enhancement (see Figure 1), and require in input an *event log* and, for conformance checking and enhancement, a *(process) model*. A log, usually described in the IEEE standard XES format[3], is a set of execution traces (or cases) each of which is an ordered sequence of events carrying a payload as a set of attribute-value pairs. Process models instead provide a description of the scenario at hand and

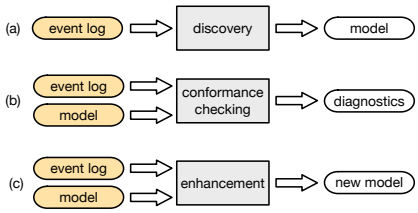

Fig. 1: The three types of process mining.

[3] http://www.xes-standard.org/

can be constructed using one of the available Business Process Modeling Languages, such as BPMN, YAWL and DECLARE.

Event logs are therefore a crucial ingredient to the accomplishment of process mining. Unfortunately, a number of difficulties may hamper the availability of event logs. Among these are partial event logs, where the execution traces may bring only **partial information** in terms of which process activities have been executed and what data or artefacts they produced. Thus repairing incomplete execution traces by reconstructing the missing entries becomes an important task to enable process mining in full, as noted in recent works such as [17, 8]. While these works deserve a praise for having motivated the importance of trace repair and having provided some basic techniques for reconstructing missing entries using the knowledge captured in process models, they all focus on event logs (and process models) of limited expressiveness. In fact, they all provide techniques for the reconstruction of control flows, thus completely ignoring the data flow component. This is a serious limitation, given the growing efforts to extend business process languages with the capability to model complex data objects [5] along with the fact that considering data in the repair task allows, in general, for reducing the number of possible trace completions, as shown in Section 2.2.

In this paper we show how to exploit state-of-the-art planning techniques to deal with the repair of data-aware event logs in the presence of imperative process models. Specifically we will focus on the well established Workflow Nets [20], a particular class of Petri nets that provides the formal foundations of several process models, of the YAWL language and have become one of the standard ways to model and analyze workflows. In particular we provide:

1. a modeling language DAW-net, an extension of the workflow nets with data formalism introduced in [18] so to be able to deal with more expressive data (Section 3);
2. a recast of data aware trace repair as a reachability problem in DAW-net (Section 4);
3. a sound and complete encoding of reachability in DAW-net in a planning problem so to be able to deal with trace repair using planning (Section 5).

The solution of the problem are all and only the repairs of the partial trace compliant with the DAW-net model. The advantage of using automated planning techniques is that we can exploit the underlying logic language to ensure that generated plans conform to the observed traces without resorting to ad hoc algorithms for the specific repair problem. The theoretical investigation presented in this work provides an important step forward towards the exploitation of planning techniques in data-aware processes.

2 Preliminaries

2.1 The Workflow Nets modeling language

Petri Nets (PN) is a modeling language for the description of distributed systems that has widely been applied to business processes. The classical PN is a directed bipartite graph with two node types, called *places* and *transitions*, connected via directed arcs. Connections between nodes of the same type are not allowed.

Definition 1 (Petri Net). *A* Petri Net *is a triple* $\langle P, T, F \rangle$ *where* P *is a set of* places*;* T *is a set of* transitions*;* $F \subseteq (P \times T) \cup (T \times P)$ *is the flow relation describing the arcs between places and transitions (and between transitions and places).*

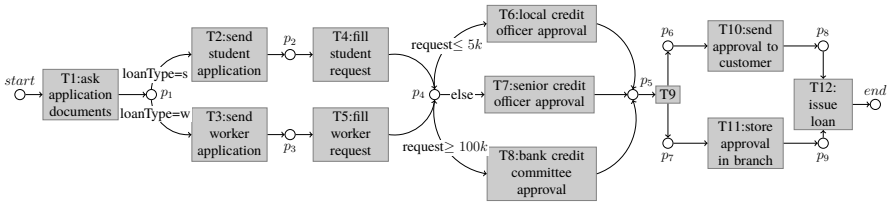

Fig. 2: A process as a Petri Net.

The *preset* of a transition t is the set of its input places: ${}^\bullet t = \{p \in P \mid (p,t) \in F\}$. The *postset* of t is the set of its output places: $t^\bullet = \{p \in P \mid (t,p) \in F\}$. Definitions of pre- and postsets of places are analogous.

Places in a PN may contain a discrete number of tokens. Any distribution of tokens over the places, formally represented by a total mapping $M : P \mapsto \mathbb{N}$, represents a configuration of the net called a *marking*.

Process tasks are modeled in PNs as transitions while arcs and places constraint their ordering. Figure 2 exemplifies how PNs can be used to model parallel and mutually exclusive choices: sequences *T2;T4-T3;T5* (transitions *T6-T7-T8*) are placed on mutually exclusive paths, while transitions *T10* and *T11* are placed on parallel paths. Finally, *T9* prevents connections between nodes of the same type.

The expressivity of PNs exceeds, in the general case, what is needed to model business processes, which typically have a well-defined starting (ending) point. This leads to the following definition of a workflow net (WF-net) [1].

Definition 2 (WF-net). *A PN $\langle P, T, F\rangle$ is a WF-net if it has a single source place $start$, a single sink place end, and every place and every transition is on a path from start to end, i.e., for all $n \in P \cup T$, $(start, n) \in F^*$ and $(n, end) \in F^*$, where F^* is the reflexive transitive closure of F.*

A marking in a WF-net represents the *workflow state* of a single case. The semantics of a PN/WF-net, and in particular the notion of *valid firing*, defines how transitions route tokens through the net so that they correspond to a process execution.

Definition 3 (Valid Firing). *A firing of a transition $t \in T$ from M to M' is* valid*, in symbols $M \xrightarrow{t} M'$, iff*

1. *t is enabled in M, i.e., $\{p \in P \mid M(p) > 0\} \supseteq {}^\bullet t$; and*
2. *the marking M' is such that for every $p \in P$:*

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + 1 & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) & \text{otherwise} \end{cases}$$

Condition 1. states that a transition is enabled if all its input places contain at least one token; 2. states that when t fires it consumes one token from each of its input places and produces one token in each of its output places.

A *case* of a WF-Net is a sequence of valid firings $M_0 \xrightarrow{t_1} M_1, M_1 \xrightarrow{t_2} M_2, \ldots, M_{k-1} \xrightarrow{t_k} M_k$ where M_0 is the marking indicating that there is a single token in $start$.

Definition 4 (k-safeness). *A marking of a PN is k-safe if the number of tokens in all places is at most k. A PN is k-safe if the initial marking is k-safe and the marking of all cases is k-safe.*

From now on we concentrate on 1-safe nets, which generalize the class of *structured workflows* and are the basis for best practices in process modeling [11]. We also use safeness as a synonym of 1-safeness. It is important to notice that our approach can be seamlessly generalized to other classes of PNs, as long as it is guaranteed that they are k-safe. This reflects the fact that the process control-flow is well-defined (see [10]).

Reachability on Petri Nets. The behavior of a PN can be described as a transition system where states are markings and directed edges represent firings. Intuitively, there is an edge from M_i to M_{i+1} labeled by t_i if $M_i \xrightarrow{t} M_{i+1}$ is a valid firing. Given a "goal" marking M_g, the reachability problem amounts to check if there is a path from the initial marking M_0 to M_g. Reachability on PNs (WF-nets) is of enormous importance in process verification as it allows for checking natural behavioral properties, such as satisfiability and soundness in a natural manner [2].

2.2 Trace repair

One of the goals of process mining is to capture the as-is processes as accurately as possible: this is done by examining event logs that can be then exploited to perform the tasks in Figure 1. In many cases, however, event logs are subject to data quality problems, resulting in *incorrect* or *missing* events in the log. In this paper we focus on the latter issue addressing the problem of **repairing execution traces that contain missing entries** (hereafter shortened in trace repair).

The need for trace repair is motivated in depth in [17], where missing entities are described as a frequent cause of low data quality in event logs, especially when the definition of the business processes integrates activities that are not supported by IT systems due either to their nature (e.g. they consist of human interactions) or to the high level of abstraction of the description, detached from the implementation. A further cause of missing events are special activities (such as transition *T9* in Figure 2) that are introduced in the model to guarantee properties concerning e.g., the structure of the workflow or syntactic constraints, but are never executed in practice.

The starting point of trace repair are *execution traces* and the knowledge captured in *process models*. Consider for instance the model in Figure 2 and the (partial) execution trace {T3, T7}. By aligning the trace to the model, techniques such as the ones presented in [17] and [8] are able to exploit the events stored in the trace and the control flow specified in the model to reconstruct two possible repairs:

$$\{T1, T3, T5, T7, T9, T10, T11, T12\}$$
$$\{T1, T3, T5, T7, T9, T11, T10, T12\}$$

Consider now a different scenario in which the partial trace reduces to $\{T7\}$. In this case, by using the control flow in Figure 2 we are not able to reconstruct whether the loan is a student loan or a worker loan. This increases the number of possible repairs and

therefore lowers the usefulness of trace repair. Assume now that the event log conforms to the XES standard and stores some observed data attached to $T7$:

$$\{T7[request = \texttt{60k}, loan = \texttt{50k}]\}$$

If the process model is able to specify how transitions can read and write variables, and furthermore some constraints on how they do it, the scenario changes completely. Indeed, assume that transition *T4* is empowered with the ability to write the variable $request$ with a value smaller or equal than `30k` (the maximum amount of a student loan). Using this fact, and the fact that the request examined by *T7* is greater than `30k`, we can understand that the execution trace has chosen the path of the worker loan. Moreover, if the model specifies that variable *loanType* is written during the execution of *T1*, when the applicant chooses the type of loan she is interested in, we are able to infer that *T1* sets variable *loanType* to `w`. This example, besides illustrating the idea of trace repair, also motivates why data are important to accomplish this task, and therefore why extending repair techniques beyond the mere control flow is a significant contribution to address data quality problems in event logs.

2.3 The planning language $\mathcal{K}$

The main elements of action languages are *fluents* and *actions*. The former represent the state of the system which may change by means of actions. Causation statements describe the possible evolution of the states, and preconditions associated to actions describe which action can be executed according to the current state. A planning problem in $\mathcal{K}$ [9] is specified using a Datalog-like language where fluents and actions are represented by literals (not necessarily ground). The specification includes the list of fluents, actions, initial state and goal conditions; also a set of statements specifies the dynamics of the planning domain using causation rules and executability conditions. The semantics of $\mathcal{K}$ borrows heavily from Answer Set Programming (ASP). In fact, the system enables the reasoning with partial knowledge and provides both weak and strong negation.

A *causation rule* is a statement of the form

caused f **if** b_1,..., b_k, **not** b_{k+1}, ..., **not** b_ℓ
after a_1,..., a_m, **not** a_{m+1}, ..., **not** a_n.

The rule states that f is true in the new state reached by executing (simultaneously) some actions, provided that $a_1, \ldots, a_m$ are known to hold while $a_{m+1}, \ldots, a_n$ are not known to hold in the previous state (some of the a_j might be actions executed on it), and $b_1, \ldots, b_k$ are known to hold while $b_{k+1}, \ldots, b_\ell$ are not known to hold in the new state. Rules without the **after** part are called *static*.

An *executability condition* is a statement of the form

executable a **if** b_1,..., b_k, **not** b_{k+1}, ..., **not** b_ℓ.

Informally, such a condition says that the action a is eligible for execution in a state, if $b_1, \ldots, b_k$ are known to hold while $b_{k+1}, \ldots, b_\ell$ are not known to hold in that state.

Terms in both kind of statements could include variables (starting with capital letter) and the statements must be safe in the usual Datalog meaning w.r.t. the first fluent or action of the statements.

A *planning domain PD* is a pair $\langle D, R \rangle$ where D is a finite set of action and fluent declarations, and R is a finite set of rules, initial state constraints, and executability conditions.

The semantics of the language is provided in terms of a transition system where the states are ASP models (sets of atoms) and actions transform the state according to the rules. A state transition is a tuple $t = \langle s, A, s' \rangle$ where s, s' are states and A is a set of action instances. The transition is said to be legal if the actions are executable in the first state and both states are the minimal ones that satisfy all causation rules. Semantics of plans including default negation is defined by means of a Gelfond-Lifschitz type reduction to a positive planning domain. A sequence of state transitions $\langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{n-1}, A_n, s_n \rangle$, $n \geq 0$, is a trajectory for PD, if s_0 is a legal initial state of PD and all $\langle s_{i-1}, A_i, s_i \rangle$, are legal state transitions of PD.

A *planning problem* is a pair composed of a planning domain PD and a ground goal $g_1, \ldots, g_m$, **not** g_{m+1}, ..., **not** g_n that has to be satisfied at the end of the execution.

3 Framework

In this section we suitably extend WF-nets to represent data and their evolution as transitions are performed. In order for such an extension to be meaningful, i.e., allowing reasoning on data, it has to provide: (i) a model for representing data; (ii) a way to make decisions on actual data values; and (iii) a mechanism to express modifications to data. We provide (i)–(iii) by enhancing WF-nets with the following elements:

- a set of variables taking values from possibly different domains (provides(i));
- queries on such variables used as transitions preconditions (provides(ii));
- variables updates and deletion in the specification of net transitions (provides(iii)).

Our framework follows the approach of state-of-the-art WF-nets with data [18, 12], from which it borrows the above concepts, extending them by allowing reasoning on actual data values as better explained in Section 6.

We make use of the WF-net in Figure 2 extended with data as a running example.

3.1 Data Model

As our focus is on trace repair, we follow the data model of the IEEE XES standard for describing logs, which represents data as a set of variables. Variables take values from specific sets on which a partial order can be defined. As customary, we distinguish between the data model, namely the intensional level, from a specific instance of data, i.e., the extensional level.

Definition 5 (Data model). *A* data model *is a tuple* $\mathcal{D} = (\mathcal{V}, \Delta, \mathsf{dm}, \mathsf{ord})$ *where:*

- $\mathcal{V}$ *is a possibly infinite set of variables;*
- $\Delta = \{\Delta_1, \Delta_2, \ldots\}$ *is a possibly infinite set of domains (not necessarily disjoint);*
- $\mathsf{dm} : \mathcal{V} \to \Delta$ *is a total and surjective function which associates to each variable* v *its domain* Δ_i*;*
- ord *is a partial function that, given a domain* Δ_i*, if* $\mathsf{ord}(\Delta_i)$ *is defined, then it returns a* partial order *(reflexive, antisymmetric and transitive)* $\leq_{\Delta_i} \subseteq \Delta_i \times \Delta_i$.

A data model for the loan example is $\mathcal{V} = \{loanType, request, loan\}$, $\mathsf{dm}(loanType) = \{\mathtt{w}, \mathtt{s}\}$, $\mathsf{dm}(request) = \mathbb{N}$, $\mathsf{dm}(loan) = \mathbb{N}$, with $\mathsf{dm}(loan)$ and $\mathsf{dm}(request)$ being totally ordered by the natural ordering $\leq$ in $\mathbb{N}$.

An actual instance of a data model is a partial function associating values to variables.

Definition 6 (Assignment). *Let $\mathcal{D} = \langle \mathcal{V}, \Delta, \mathsf{dm}, \mathsf{ord} \rangle$ be a data model. An* assignment *for variables in $\mathcal{V}$ is a* partial *function $\eta : \mathcal{V} \to \bigcup_i \Delta_i$ such that for each $v \in \mathcal{V}$, if $\eta(v)$ is defined, i.e., $v \in img(\eta)$ where img is the image of η, then we have $\eta(v) \in \mathsf{dm}(v)$.*

We now define our boolean query language, which notably allows for equality and comparison. As will become clearer in Section 3.2, queries are used as *guards*, i.e., preconditions for the execution of transitions.

Definition 7 (Query language - syntax). *Given a data model, the language $\mathcal{L}(\mathcal{D})$ is the set of formulas Φ inductively defined according to the following grammar:*

$$\Phi \quad := \quad true \mid \mathsf{def}(v) \mid t_1 = t_2 \mid t_1 \leq t_2 \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_2$$

where $v \in \mathcal{V}$ and $t_1, t_2 \in \mathcal{V} \cup \bigcup_i \Delta_i$.

Examples of queries of the loan scenarios are $request \leq \mathtt{5k}$ or $loanType = \mathtt{w}$. Given a formula Φ and an assignment η, we write $\Phi[\eta]$ for the formula Φ where each occurrence of variable $v \in img(\eta)$ is replaced by $\eta(v)$.

Definition 8 (Query language - semantics). *Given a data model $\mathcal{D}$, an assignment η and a query $\Phi \in \mathcal{L}(\mathcal{D})$ we say that $\mathcal{D}, \eta$ satisfies Φ, written $D, \eta \models \Phi$ inductively on the structure of Φ as follows:*

- $\mathcal{D}, \eta \models true$;
- $\mathcal{D}, \eta \models \mathsf{def}(v)$ *iff* $v \in img(\eta)$;
- $\mathcal{D}, \eta \models t_1 = t_2$ *iff* $t_1[\eta], t_2[\eta] \notin \mathcal{V}$ *and* $t_1[\eta] \equiv t_2[\eta]$;
- $\mathcal{D}, \eta \models t_1 \leq t_2$ *iff* $t_1[\eta], t_2[\eta] \in \Delta_i$ *for some* i *and* $\mathsf{ord}(\Delta_i)$ *is defined and* $t_1[\eta] \leq_{\Delta_i} t_2[\eta]$;
- $\mathcal{D}, \eta \models \neg\Phi$ *iff it is not the case that* $\mathcal{D}, \eta \models \Phi$;
- $\mathcal{D}, \eta \models \Phi_1 \wedge \Phi_2$ *iff* $\mathcal{D}, \eta \models \Phi_1$ *and* $\mathcal{D}, \eta \models \Phi_2$.

Intuitively, def can be used to check if a variable has an associated value or not (recall that assignment η is a partial function); equality has the intended meaning and $t_1 \leq t_2$ evaluates to true iff t_1 and t_2 are values belonging to the same domain Δ_i, such a domain is ordered by a partial order $\leq_{\Delta_i}$ and t_1 is actually less or equal than t_2 according to $\leq_{\Delta_i}$.

3.2 Data-aware net

We now combine the data model with a WF-net and formally define how transitions are guarded by queries and how they update/delete data. The result is a Data-AWare net (DAW-net) that incorporates aspects (i)–(iii) described at the beginning of Section 3.

Definition 9 (DAW-net). *A* DAW-net *is a tuple $\langle \mathcal{D}, \mathcal{N}, \mathsf{wr}, \mathsf{gd} \rangle$ where:*

- $\mathcal{N} = \langle P, T, F \rangle$ *is a WF-net;*
- $\mathcal{D} = \langle \mathcal{V}, \Delta, \mathsf{dm}, \mathsf{ord} \rangle$ *is a data model;*
- $\mathsf{wr} : T \mapsto (\mathcal{V}' \mapsto 2^{\mathsf{dm}(\mathcal{V})})$*, where* $\mathcal{V}' \subseteq \mathcal{V}$*,* $\mathsf{dm}(\mathcal{V}) = \bigcup_{v \in V} \mathsf{dm}(v)$ *and* $\mathsf{wr}(t)(v) \subseteq \mathsf{dm}(v)$ *for each* $v \in \mathcal{V}'$*, is a function that associates each transition to a (*partial*) function mapping variables to a finite subset of their domain.*
- $\mathsf{gd} : T \mapsto \mathcal{L}(\mathcal{D})$ *is a function that associates a guard to each transition.*

Function gd associates a guard, namely a query, to each transition. The intuitive semantics is that a transition t can fire if its guard $\mathsf{gd}(t)$ evaluates to true (given the current assignment of values to data). Examples are $\mathsf{gd}(T6) = request \leq \texttt{5k}$ and $\mathsf{gd}(T8) = \neg(request \leq \texttt{99999})$. Function wr is instead used to express how a transition t modifies data: after the firing of t, each variable $v \in \mathcal{V}'$ can take any value among a specific finite subset of $\mathsf{dm}(v)$. We have three different cases:

- $\emptyset \subset \mathsf{wr}(t)(v) \subseteq \mathsf{dm}(v)$: t nondeterministically assigns a value from $\mathsf{wr}(t)(v)$ to v;
- $\mathsf{wr}(t)(v) = \emptyset$: t deletes the value of v (hence making v undefined);
- $v \notin dom(\mathsf{wr}(t))$: value of v is not modified by t.

Notice that by allowing $\mathsf{wr}(t)(v) \subseteq \mathsf{dm}(v)$ in the first bullet above we enable the specification of restrictions for specific tasks. E.g., $\mathsf{wr}(T4) : \{request\} \mapsto \{\texttt{0} \ldots \texttt{30k}\}$ says that $T4$ writes the $request$ variable and intuitively that students can request a maximum loan of $\texttt{30k}$, while $\mathsf{wr}(T5) : \{request\} \mapsto \{\texttt{0} \ldots \texttt{500k}\}$ says that workers can request up to $\texttt{500k}$.

The intuitive semantics of gd and wr is formalized next. We start from the definition of DAW-net state, which includes both the state of the WF-net, namely its marking, and the state of data, namely the assignment. We then extend the notions of state transition and valid firing.

Definition 10 (DAW-net state). *A* state *of a DAW-net* $\langle \mathcal{D}, \mathcal{N}, \mathsf{wr}, \mathsf{gd} \rangle$ *is a pair* (M, η) *where* M *is a marking for* $\langle P, T, F \rangle$ *and* η *is an assignment for* $\mathcal{D}$*.*

Definition 11 (DAW-net Valid Firing). *Given a DAW-net* $\langle \mathcal{D}, \mathcal{N}, \mathsf{wr}, \mathsf{gd} \rangle$*, a firing of a transition* $t \in T$ *is a* valid firing *from* (M, η) *to* (M', η')*, written as* $(M, \eta) \xrightarrow{t} (M', \eta')$*, iff conditions 1. and 2. of Def. 3 holds for* M *and* M'*, i.e., it is a WF-Net valid firing, and*

1. $\mathcal{D}, \eta \models \mathsf{gd}(t)$*,*
2. *assignment* η' *is such that, if* $\textsc{wr} = \{v \mid \mathsf{wr}(t)(v) \neq \emptyset\}$*,* $\textsc{del} = \{v \mid \mathsf{wr}(t)(v) = \emptyset\}$*:*
   - *its domain* $dom(\eta') = dom(\eta) \cup \textsc{wr} \setminus \textsc{del}$*;*
   - *for each* $v \in dom(\eta')$*:*

$$\eta'(v) = \begin{cases} d \in \mathsf{wr}(t)(v) & \text{if } v \in \textsc{wr} \\ \eta(v) & \text{otherwise.} \end{cases}$$

Condition 1. and 2. extend the notion of valid firing of WF-nets imposing additional pre- and postconditions on data, i.e., preconditions on η and postconditions on η'. Specifically, 1. says that for a transition t to be fired its guard $\mathsf{gd}(t)$ must be satisfied by the current assignment η. Condition 2. constrains the new state of data: the domain of η' is defined as the union of the domain of η with variables that are written (WR), minus the

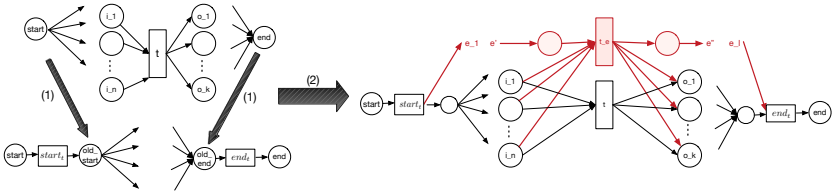

Fig. 3: Outline of the trace "injection"

set of variables that must be deleted (DEL). Variables in $dom(\eta')$ can indeed be grouped in three sets depending on the effects of t: (i) OLD $= dom(\eta) \setminus$ WR: variables whose value is unchanged after t; (ii) NEW $=$ WR $\setminus dom(\eta)$: variables that were undefined but have a value after t; and (iii) OVERWR $=$ WR $\cap\, dom(\eta)$: variables that did have a value and are updated with a new one after t. The final part of condition 2. says that each variable in NEW $\cup$ OVERWR takes a value in $\mathsf{wr}(t)(v)$, while variables in OLD maintain the old value $\eta(v)$.

A *case* of a DAW-net is defined as a case of a WF-net, with the only difference that the assignment η_0 of the initial state (M_0, η_0) is empty, i.e., $dom(\eta_0) = \emptyset$.

4 Trace repair as reachability

In this section we provide the intuition behind our technique for solving the trace repair problem via reachability. Full details and proofs are contained in [7].

A *trace* is a sequence of observed *events*, each with a payload including the transition it refers to and its effects on the data, i.e., the variables updated by its execution. Intuitively, a DAW-net case is *compliant* w.r.t. a trace if it contains all the occurrences of the transitions observed in the trace (with the corresponding variable updates) in the right order.

As a first step, we assume without loss of generality that DAW-net models start with a special transition $start_t$ and terminate with a special transition end_t. Every process can be reduced to such a structure as informally illustrated in the left hand side of Figure 3 by arrows labeled with (1). Note that this change would not modify the behavior of the net: any sequence of firing valid for the original net can be extended by the firing of the additional transitions and vice versa.

Next, we illustrate the main idea behind our approach by means of the right hand side of Figure 3: we consider the observed events as transitions (in red) and we suitably "inject" them in the original DAW-net. By doing so, we obtain a new model where, intuitively, tokens are forced to activate the red transitions of DAW-net, when events are observed in the trace. When, instead, there is no red counterpart, i.e., there is missing information in the trace, the tokens move in the black part of the model. The objective is then to perform reachability for the final marking (i.e., to have one token in the end place and all other places empty) over such a new model in order to obtain all and only the possible repairs for the partial trace.

More precisely, for each event e with a payload including transition t and some effect on variables we introduce a new transition t_e in the model such that:

- t_e is placed in parallel with the original transition t;
- t_e includes an additional input place connected to the preceding event and an additional output place which connects it to the next event;
- $\mathsf{gd}(t_e) = \mathsf{gd}(t)$ and
- $\mathsf{wr}(t_e)$ specifies exactly the variables and the corresponding values updated by the event, i.e. if the event set the value of v to d, then $\mathsf{wr}(t_e)(v) = \{d\}$; if the event deletes the variable v, then $\mathsf{wr}(t_e)(v) = \emptyset$.

Given a trace τ and a DAW-net W, it is easy to see that the resulting *trace workflow* (indicated as W^τ) is a strict extension of W (only new nodes are introduced) and, since all newly introduced nodes are in a path connecting the start and sink places, it is a DAW-net, whenever the original one is a DAW-net net.

We now prove the soundness and completeness of the approach by showing that: (1) all cases of W^τ are compliant with τ; (2) each case of W^τ is also a case of W and (3) if there is a case of W compliant with τ, then that is also a case for W^τ.

Property (1) is ensured by construction. For (2) and (3) we need to relate cases from W^τ to the original DAW-net W. We indeed introduce a projection function $\mathit{\Pi}_\tau$ that maps elements from cases of the enriched DAW-net to cases of elements from the original DAW-net. Essentially, $\mathit{\Pi}_\tau$ maps newly introduced transitions t_e to the corresponding transitions in event e, i.e., t, and also projects away the new places in the markings. Given that the structure of W^τ is essentially the same as that of W with additional copies of transitions that are already in W, it is not surprising that any case for W^τ can be replayed on W by mapping the new transitions t_e into the original ones t, as shown by the following:

Lemma 1. *If C is a case of W^τ then $\mathit{\Pi}_\tau(C)$ is a case of W.*

This lemma proves that whenever we find a case on W^τ, then it is an example of a case on W that is compliant with τ, i.e., (2). However, to reduce the original problem to reachability on DAW-net, we need to prove that *all* the W cases compliant with τ can be replayed on W^τ, that is, (3). In order to do that, we can build a case for W^τ starting from the compliant case for W, by substituting the occurrences of firings corresponding to events in τ with the newly introduced transitions. The above results pave the way to the following:

Theorem 1. *Let W be a DAW-net and $\tau = (e_1, \ldots, e_n)$ a trace; then W^τ characterises all and only the cases of W compatible with τ. That is*
$\Rightarrow$ *if C is a case of W^τ containing t_{e_n} then $\mathit{\Pi}_\tau(C)$ is compatible with τ; and*
$\Leftarrow$ *if C is a case of W compatible with τ, then there is a case C' of W^τ s.t. $\mathit{\Pi}_\tau(C') = C$.*

Theorem 1 provides the main result of this section and is the basis for the reduction of the trace repair for W and τ to the reachability problem for W^τ. In fact, by enumerating all the cases of W^τ reaching the final marking (i.e. a token in end) we can provide all possible repairs for the partial observed trace. Moreover, the transformation generating W^τ is preserving the safeness properties of the original workflow:

Lemma 2. *Let W be a DAW-net and τ a trace of W. If W is k-safe then W^τ is k-safe as well.*

This is essential to guarantee the decidability of the reasoning techniques described in the next section.

5 Reachability as a planning problem

In this section we exploit the similarity between workflows and planning domains in order to describe the evolution of a DAW-net by means of a planning language. Once the original workflow behaviour has been encoded into an equivalent planning domain, we can use the automatic derivation of plans with specific properties to solve the reachability problem. In our approach we introduce a new action for each transition (to ease the description we will use the same names) and represent the status of the workflow – marking and variable assignments – by means of fluents. Although their representation as dynamic rules is conceptually similar we will separate the description of the encoding by considering first the behavioural part (the WF-net) and then the encoding of data (variable assignments and guards).

5.1 Encoding DAW-net behaviour

Since we focus on 1-safe WF-nets the representation of markings is simplified by the fact that each place can either contain 1 token or no tokens at all. This information can be represented introducing a propositional fluent for each place, true iff the corresponding place holds a token. Let us consider $\langle P, T, F\rangle$ the *safe WF-net* component of a DAW-net system. The declaration part of the planning domain will include:

- a fluent declaration p for each place $p \in P$;
- an action declaration t for each task $t \in T$.

Since each transition can be fired[4] only if each input place contains a token, then the corresponding action can be executed when place fluents are true: for each task $t \in T$, given $\{i_1^t, \ldots, i_n^t\} = {}^\bullet t$, we include the executability condition:

executable t **if** $i_1^t, \ldots, i_n^t$.

As valid firings are sequential, namely only one transition can be fired at each step, we disable concurrency in the planning domain introducing the following rule for each pair of tasks $t_1, t_2 \in T$[5]

caused false **after** t_1, t_2.

Transitions transfer tokens from input to output places. Thus the corresponding actions must clear the input places and set the output places to true. This is enforced by including

caused $-i_1^t$ **after** t. ... **caused** $-i_n^t$ **after** t.

caused o_1^t **after** t. ... **caused** o_k^t **after** t.

for each task $t \in T$ and $\{i_1^t, \ldots, i_n^t\} = {}^\bullet t \setminus t^\bullet$, $\{o_1^t, \ldots, o_k^t\} = t^\bullet$. Finally, place fluents should be inertial since they preserve their value unless modified by an action. This is enforced by adding for each $p \in P$

caused p **if not** $-p$ **after** p.

[4] Guards will be introduced in the next section.

[5] For efficiency reasons we can relax this constraint by disabling concurrency only for transitions sharing places or updating the same variables. This would provide shorter plans.

Planning problem. Besides the domain described above, a planning problem includes an initial state, and a goal. In the initial state the only place with a token is the source:

initially: $start$.

The formulation of the goal depends on the actual instance of the reachability problem we need to solve. The goal corresponding to the state in which the only place with a token is end is written as:

goal: end, **not** p_1, ..., **not** p_k?

where $\{p_1, \ldots, p_k\} = P \setminus \{end\}$.

5.2 Encoding data

For each variable $v \in \mathcal{V}$ we introduce a fluent unary predicate var_v holding the value of that variable. Clearly, var_v predicates must be functional and have no positive instantiation for undefined variables.

We also introduce auxiliary fluents to facilitate the writing of the rules. Fluent def_v indicates whether the v variable is *not* undefined – it is used both in tests and to enforce models where the variable is assigned/unassigned. The fluent chng_v is used to inhibit inertia for the variable v when its value is updated because of the execution of an action.

DAW-net includes the specification of the set of values that each transition can write on a variable. This information is static, therefore it is included in the background knowledge by means of a set of unary predicates $\mathsf{dom}_{v,t}$ as a set of facts:

$\mathsf{dom}_{v,t}(\mathsf{e})$.

for each $v \in \mathcal{V}$, $t \in T$, and $e \in \mathsf{wr}(t)(v)$.

Constraints on variables. For each variable $v \in \mathcal{V}$:

- we impose functionality

  caused false **if** $\mathsf{var}_v(\mathsf{X})$, $\mathsf{var}_v(\mathsf{Y})$, X != Y.
- we force its value to propagate to the next state unless it is modified by an action (chng_v)

  caused $\mathsf{var}_v(\mathsf{X})$ **if not** $-\mathsf{var}_v(\mathsf{X})$, **not** chng_v **after** $\mathsf{var}_v(\mathsf{X})$.
- the defined fluent is the projection of the argument

  caused def_v **if** $\mathsf{var}_v(\mathsf{X})$.

Variable updates. The value of a variable is updated by means of causation rules that depend on the transition t that operates on the variable, and depends on the value of $\mathsf{wr}(t)$. For each v in the domain of $\mathsf{wr}(t)$:

- $\mathsf{wr}(t)(v) = \emptyset$: delete (undefine) a variable v

  caused false **if** def_v **after** t.

  caused chng_v **after** t.
- $\mathsf{wr}(t)(v) \subseteq \mathsf{dm}(v)$: set v with a value nondeterministically chosen among a set of elements from its domain

  caused $\mathsf{var}_v(\mathsf{V})$ **if** $\mathsf{dom}_{v,t}(\mathsf{V})$, **not** $-\mathsf{var}_v(\mathsf{V})$ **after** t.

  caused $-\mathsf{var}_v(\mathsf{V})$ **if** $\mathsf{dom}_{v,t}(\mathsf{V})$, **not** $\mathsf{var}_v(\mathsf{V})$ **after** t.

  caused false **if not** def_v **after** t.

  caused chng_v **after** t.

If $\mathsf{wr}(t)(v)$ contains a single element d, then the assignment is deterministic and the first three rules above can be substituted with[6]

caused $\mathsf{var}_v(\mathsf{d})$ **after** t.

Guards. To each subformula φ of transition guards is associated a fluent grd_φ that is true when the corresponding formula is satisfied. To simplify the notation, for any transition t, we will use grd_t to indicate the fluent $\mathsf{grd}_{\mathsf{gd}(t)}$. Executability of transitions is conditioned to the satisfiability of their guards; instead of modifying the executability rule including the grd_t among the preconditions, we use a constraint rule preventing executions of the action whenever its guard is not satisfied:

caused false **after** t, **not** grd_t.

Translation of atoms (ξ) is defined in terms of var_v predicates. For instance $\xi(v = w)$ corresponds to $\mathsf{var}_v(\mathsf{V})$, $\mathsf{var}_w(\mathsf{W})$, V == W. That is $\xi(v, T) = \mathsf{var}_t(\mathsf{T})$ for $t \in \mathcal{V}$, and $\xi(d, T) = \mathsf{var}_t\mathsf{T} == d$ for $d \in \bigcup_i \Delta_i$. For each subformula φ of transition guards a static rule is included to "define" the fluent grd_φ:

$$
\begin{aligned}
true &: \textbf{caused } \mathsf{grd}_\varphi \textbf{ if true} .\\
\mathsf{def}(v) &: \textbf{caused } \mathsf{grd}_\varphi \textbf{ if } \mathsf{def}_v .\\
t_1 = t_2 &: \textbf{caused } \mathsf{grd}_\varphi \textbf{ if } \xi(t_1,\mathsf{T1}),\ \xi(t_2,\mathsf{T2}),\ \mathsf{T1} == \mathsf{T2} .\\
t_1 \leq t_2 &: \textbf{caused } \mathsf{grd}_\varphi \textbf{ if } \xi(t_1,\mathsf{T1}),\ \xi(t_2,\mathsf{T2}),\ \mathsf{ord}(\mathsf{T1},\mathsf{T2}) .\\
\neg\varphi_1 &: \textbf{caused } \mathsf{grd}_\varphi \textbf{ if not } \mathsf{grd}_{\varphi_1} .\\
\varphi_1 \wedge \ldots \wedge \varphi_n &: \textbf{caused } \mathsf{grd}_\varphi \textbf{ if } \mathsf{grd}_{\varphi_1}, \ldots, \mathsf{grd}_{\varphi_n} .
\end{aligned}
$$

5.3 Correctness and completeness

We provide a sketch of the correctness and completeness of the encoding. Proofs can be found in [7].

Planning states include all the information to reconstruct the original DAW-net states. In fact, we can define a function $\Phi(\cdot)$ mapping consistent planning states into DAW-net states as following: $\Phi(s) = (M, \eta)$ with

$$
\forall p \in P,\ M(p) = \begin{cases} 1 \text{ if } p \in s \\ 0 \text{ otherwise} \end{cases} \qquad \eta = \{(v, d) \mid \mathrm{var}_v(d) \in s\}
$$

$\Phi(s)$ is well defined because s it cannot be the case that $\{\mathrm{var}_v(d), \mathrm{var}_v(d')\} \subseteq s$ with $d \neq d'$, otherwise the static rule

caused false **if** $\mathsf{var}_v(\mathsf{X})$, $\mathsf{var}_v(\mathsf{Y})$, X != Y.

would not be satisfied. Moreover, 1-safeness implies that we can restrict to markings with range in $\{0, 1\}$. By looking at the static rules we can observe that those defining the predicates def_v and grd_t are stratified. Therefore their truth assignment depends only on the extension of $\mathrm{var}_v(\cdot)$ predicates. This implies that grd_t fluents are satisfied iff the variables assignment satisfies the corresponding guard $\mathsf{gd}(t)$. Based on these observations, the correctness of the encoding is relatively straightforward since we need

[6] The deterministic version is a specific case of the non-deterministic ones and equivalent in the case that there is a single $\mathrm{dom}_{v,t}(d)$ fact.

to show that a legal transition in the planning domain can be mapped to a valid firing. This is proved by inspecting the dynamic rules.

Lemma 3 (Correctness). *Let W be a DAW-net and $\Omega(W)$ the corresponding planning problem. If $\langle s, \{t\}, s' \rangle$ is a legal transition in $\Omega(W)$, then $\Phi(s) \xrightarrow{t} \Phi(s')$ is a valid firing of W.*

The proof of completeness is more complex because – given a valid firing – we need to build a new planning state and show that it is minimal w.r.t. the transition. Since the starting state s of $\langle s, \{t\}, s' \rangle$ does not require minimality we just need to show its existence, while s' must be carefully defined on the basis of the rules in the planning domain.

Lemma 4 (Completeness). *Let W be a DAW-net, $\Omega(W)$ the corresponding planning problem and $(M, \eta) \xrightarrow{t} (M', \eta')$ be a valid firing of W. Then for each consistent state s s.t. $\Phi(s) = M$ there is a consistent state s' s.t. $\Phi(s') = M'$ and $\langle s, \{t\}, s' \rangle$ is a legal transition in $\Omega(W)$.*

Lemmata 3 and 4 provide the basis for the inductive proof of the following theorem:

Theorem 2. *Let W be a* safe WF-net *and $\Omega(PN)$ the corresponding planning problem. Let (M_0, η_0) be the initial state of W – i.e. with a single token in the source and no assignments – and s_0 the planning state satisfying the initial condition.*

($\Rightarrow$) *For* any *case in W*

$$\zeta : (M_0, \eta_0) \xrightarrow{t_1} (M_1, \eta_1) \dots (M_{n-1}, \eta_{n-1}) \xrightarrow{t_n} (M_n, \eta_n)$$

there is a trajectory in $\Omega(W)$

$$\eta : \langle s_0, \{t_1\}, s_1 \rangle, \dots, \langle s_{n-1}, \{t_n\}, s_n \rangle$$

such that $(M_i, \eta_i) = \Phi(s_i)$ for each $i \in \{0 \dots n\}$ and viceversa.

($\Leftarrow$) *For each trajectory*

$$\eta : \langle s_0, \{t_1\}, s_1 \rangle, \dots, \langle s_{n-1}, \{t_n\}, s_n \rangle$$

in $\Omega(W)$, the following sequence of firings is a case of W

$$\zeta : \Phi(s_0) \xrightarrow{t_1} \Phi(s_1) \dots \Phi(s_{n-1}) \xrightarrow{t_n} \Phi(s_n).$$

Theorem 2 above enables the exploitation of planning techniques to solve the reachability problem in DAW-net. Indeed, to verify whether the final marking is reachable it is sufficient to encode it as a condition for the final state and verify the existence of a trajectory terminating in a state where the condition is satisfied. Decidability of the planning problem is guaranteed by the fact that domains are effectively finite, as in Definition 9 the wr functions range over a finite subset of the domain and by the fact that the planner takes as input the maximum length of the plan to be returned (note that this allows for dealing with loops).

6 Related Work and Conclusions

The key role of data in the context of business processes has been recently recognized. A number of variants of PNs have been enriched so as to make tokens able to carry data and transitions aware of the data, as in the case of Workflow nets enriched with data [18, 12], the model adopted by the business process community. In [18] Workflow Net transitions are enriched with information about data (e.g., a variable $request$) and about how it is used by the activity (for reading or writing purposes). Nevertheless, these nets do not consider data values (e.g., in the example of Section 2.2 we would not be aware of the values of the variable $request$ that *T4* is enabled to write). They only allow for the identification of whether the value of the data element is `defined` or `undefined`, thus limiting the reasoning capabilities that can be provided on top of them. For instance, in the example of Section 2.2, we would not be able to discriminate between the worker and the student loan for the trace in (2.2), as we would only be aware that $request$ is `defined` after *T4*.

The problem of incomplete traces has been investigated in a number of works of trace alignment in the field of process mining, where it still represents one of the challenges. Several works have addressed the problem of aligning event logs and procedural models, without [3] and with [13, 12] data. All these works, however, explore the search space of possible moves in order to find the best one aligning the log and the model. Differently from them, in this work (i) we assume that the model is correct and we focus on the repair of incomplete execution traces; (ii) we want to exploit state-of-the-art planning techniques to reason on control and data flow rather than solving an optimisation problem.

We can overall divide the approaches facing the problem of reconstructing flows of model activities given a partial set of information in two groups: quantitative and qualitative. The former rely on the availability of a probabilistic model of execution and knowledge. For example, in [17], the authors exploit stochastic PNs and Bayesian Networks to recover missing information (activities and their durations). The latter stand on the idea of describing "possible outcomes" regardless of likelihood; hence, knowledge about the world will consist of equally likely "alternative worlds" given the available observations in time, as in this work. For example, in [4] the same issue of reconstructing missing information has been tackled by reformulating it in terms of a Satisfiability(SAT) problem rather than as a planning problem.

Planning techniques have already been used in the context of business processes, e.g., for verifying process constraints [16] or for the construction and adaptation of autonomous process models [19, 15]. In [6] automated planning techniques have been applied for aligning execution traces and declarative models. As in this work, in [8], planning techniques have been used for addressing the problem of incomplete execution traces with respect to procedural models. However, differently from the two approaches above, this work uses for the first time planning techniques to target the problem of completing incomplete execution traces with respect to a procedural model that also takes into account data and the value they can assume.

Despite this work mainly focuses on the problem of trace completion, the proposed automated planning approach can easily exploit reachability for model satisfiability and trace compliance and furthermore can be easily extended also for aligning data-

aware procedural models and execution traces. Moreover, the presented encoding in the planning language $\mathcal{K}$, can be directly adapted to other action languages with an expressiveness comparable to $\mathcal{C}$ [14]. In the future, we would like to explore these extensions and implement the proposed approach and its variants in a prototype.

References

1. van der Aalst, W.: The application of petri nets to workflow management. J. of Circuits, Sys. and Comp. 08, 21–66 (Feb 1998)
2. van der Aalst, W.M.P.: Verification of workflow nets. In: Proc. of ICATPN. pp. 407–426 (1997)
3. Adriansyah, A., van Dongen, B.F., van der Aalst, W.: Conformance checking using cost-based fitness analysis. In: Proc. of EDOC. pp. 55–64 (2011)
4. Bertoli, P., Di Francescomarino, C., Dragoni, M., Ghidini, C.: Reasoning-based techniques for dealing with incomplete business process execution traces. In: AI*IA, LNCS, vol. 8249, pp. 469–480. Springer (2013)
5. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data-aware process analysis: A database theory perspective. pp. 1–12 (2013)
6. De Giacomo, G., Maggi, F.M., Marrella, A., Sardiña, S.: Computing trace alignment against declarative process models through planning. In: ICAPS. pp. 367–375 (2016)
7. De Masellis, R., Di Francescomarino, C., Ghidini, C., Tessaris, S.: Enhancing workflow-nets with data for trace completion (2017), `https://arxiv.org/abs/1706.00356`
8. Di Francescomarino, C., Ghidini, C., Tessaris, S., Sandoval, I.V.: Completing workflow traces using action languages. In: CAiSE. LNCS, vol. 9097, pp. 314–330. Springer (2015)
9. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, II: The DLVK system. Art. Intell. 144(1–2), 157–211 (2003)
10. van Hee, K., Sidorova, N., Voorhoeve, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: ICATPN. No. 2679 in Lecture Notes in Computer Science, Springer (2003)
11. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.J.: On structured workflow modelling. In: Seminal Contributions to Information Systems Engineering (2013)
12. de Leoni, M., van der Aalst, W.: Data-aware Process Mining: Discovering Decisions in Processes Using Alignments. In: Proc of ACM SAC. pp. 1454–1461 (2013)
13. de Leoni, M., van der Aalst, W., van Dongen, B.F.: Data- and resource-aware conformance checking of business processes. In: BIS, LNBIP, vol. 117, pp. 48–59 (2012)
14. Lifschitz, V.: Action languages, answer sets and planning. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 357–373. Springer (1999)
15. Marrella, A., Russo, A., Mecella, M.: Planlets: Automatically recovering dynamic processes in YAWL. In: OTM Conferences (1). pp. 268–286 (2012)
16. Regis, G., Ricci, N., Aguirre, N., Maibaum, T.S.E.: Specifying and verifying declarative fluent temporal logic properties of workflows. In: Proc. of SBMF. pp. 147–162 (2012)
17. Rogge-Solti, A., Ronny, S., van der Aalst, W., Weske, M.: Improving documentation by repairing event logs. In: The Practice of Enterprise Modeling, LNBIP, vol. 165, pp. 129–144. Springer (2013)
18. Sidorova, N., Stahl, C., Trčka, N.: Soundness verification for conceptual workflow nets with data. Inf. Sys. 36(7), 1026–1043 (Nov 2011)
19. da Silva, C.E., de Lemos, R.: A framework for automatic generation of processes for self-adaptive software systems. Informatica (Slov.) 35(1), 3–13 (2011)
20. van der Aalst, W., Hee, K.v., Hofstede, A.t., Sidorova, N., Verbeek, H., Voorhoeve, M., Wynn, M.: Soundness of workflow nets. Formal Aspects of Comp. 23(3), 333–363 (2010)