Cleaning Structured Event Logs: A Graph Repair Approach

Jianmin Wang†, Shaoxu Song†, Xuemin Lin‡, Xiaochen Zhu†, Jian Pei‡

†Key Laboratory for Information System Security, MoE; TNList; School of Software, Tsinghua University, Beijing, China
{jinwang, sxsong}@tsinghua.edu.cn zhu-xcl@mails.tsinghua.edu.cn
‡University of New South Wales, Sydney, Australia lxue@cse.unsw.edu.au
†Simon Fraser University, Burnaby, BC, Canada jpei@cs.sfu.ca

Abstract—Event data are often dirty owing to various recording conventions or simply system errors. These errors may cause many serious damages to real applications, such as inaccurate provenance answers, poor profiling results, or concealing interesting patterns from event data. Cleaning dirty event data is strongly demanded. While existing event data cleaning techniques view event logs as sequences, structural information do exist among events. We argue that such structural information enhances not only the accuracy of repairing inconsistent events but also the computation efficiency. It is notable that both the structure and the names (labeling) of events could be inconsistent. In real applications, while unsound structure is not repaired automatically (which needs manual effort from business actors to handle the structure error), it is highly desirable to repair the inconsistent event names introduced by recording mistakes. In this paper, we propose a graph repair approach for 1) detecting unsound event names introduced by recording mistakes, and 2) repairing inconsistent event name.

I. INTRODUCTION

Event data, logging the execution of business processes or workflows, are often varying in precision, duration and relevance [24]. In particular, execution of a business process may be distributed in multiple companies or divisions, with various event recording conventions or even erroneous executions. The corresponding event data scattered over a heterogeneous environment involve inconsistencies and errors [26]. According to our survey on a real dataset (in Section V-A), about 82% execution traces of processes are dirty.

The dirty event data lead to wild data provenance answers [28], mislead the aggregation profiling in process data warehousing [9], or obstruct finding interesting process patterns [17]. Indeed, the event data quality is essential in process mining, and known as the first challenge in the Process Mining Manifesto by the IEEE Task Force on Process Mining [30].

Existing approaches [11], [32] on cleaning event data treat event logs as unstructured sequences. It is worth noting that structural information do exist among events. A very common example is the task passing relationships, e.g., the manager assigns the work to another staff for succeeding operations (see details below). We argue that such structural information are not only essential to obtaining more precise event repairs but also useful in improving the computation efficiency.

Example 1. We illustrate a real example of part design process in a major bus manufacturer†. Figure 1(a) illustrates 6 steps (events $t_1$ to $t_6$) of accomplishing a part design, a.k.a., an execution trace. Each event includes a Name of being processed task, the Operator executing the task, and the Successors of the follow-up activities being assigned to. The links of Successor and Operator between events indicate the structural information. For example, the Successor of $t_2$ assigns the follow-up tasks to J.Zhe and O. Chu (corresponding to Operators in $t_3$ and $t_4$, respectively). It indicates the links from $t_2$ to $t_3$ and $t_4$ as illustrated in the graph of Figure 1(a).

The execution of events should follow some process specifications, as shown in Figure 1(c). Following the standard of PNML2, we represent specifications by the notation of Petri Net [29], which is a bipartite directed graph of circles (places) and rectangles (transitions). Each transition denotes an event type, e.g., the first event type submit in Figure 1(c). Arcs with dependent relationships between transitions and places represent the control flow. In particular, flows attached to transitions have AND semantics, such as AND-split after transition design. It indicates that both flows after

†It involves about 70 process specifications for product lifecycle management, which are manipulated by more than 100 outsourcing companies.

‡Petri Net Markup Language, an XML-based syntax for interchanging workflows. See an introduction in the reference site http://www.pnml.org/
design should be executed simultaneously. On the other hand, transition evaluate involving AND-join can be enabled when both the preceding flows are complete. Moreover, places specify XOR semantics, e.g., only one of the flows after place a (XOR-split) will be executed, i.e., either design a new part or revise an existing one. Consequently, the XOR-join, e.g., in place s, indicates the end of XOR choices, that is, the execution will proceed when one of the flows before place s is processed. Likewise, either insulation proof or electrician proof can appear in an execution trace after design.

It is notable that the events in this part design process are executed by (more than 10) distinct departments and outsourcing companies. Due to various event recording conventions, a simple proof event name is reported in t3. It is not clear whether it denotes insulation proof, electrician proof or proof check. Such ambiguity leads to violations in conformance checking between execution and specification.

To resolve the inconsistencies, existing repairing techniques [11] may repair the sequence of events t1, ..., t6 to either σ1 (submit, design, insulation proof, check inventory, evaluate, archive) σ2 (submit, design, electrician proof, check inventory, evaluate, archive) for designing a new part, or σ3 (submit, revise, proof check, merge, re-evaluate, archive) for revising an existing part. Referring to the structural information, i.e., t2 evoking two parallel events t3 and t4 (by J. Zhe & O. Chu, respectively), the later one σ3 is an invalid repair where no parallel tasks exist. Instead, the parallel insulation/electrician proof and check inventory after design in σ1/σ2 match exactly the structure. With the structural information, we are able to identify the more likely repair σ1/σ2 and discard the irrational σ3, which cannot be distinguished by existing [11] with sequential information only. (σ3 and σ2 can further be distinguished via the cost model in Section II-D.)

To capture structural information and conformance to specification, we also use Petri Net to represent execution traces, called Causal Net. Figure 1(b) shows the net of the execution trace in Figure 1(a). It can be simply transformed from the graph in Figure 1(a) by replacing each edge with a place. Every place in the net is attached with at most one flow (since only one of the alternatives can be executed in XOR-split).

The work is accomplished when the flow successfully executes from start to end exactly following the AND/XOR constraints on event (type) names specified by the specification.

With structural information, we can directly ignore the repair candidate σ3, which corresponds to the revise division without parallel tasks. Repairing efficiency is thus improved compared with the simple sequence-based approaches.

In general, both the Name labeling and the structural Operator/Successor may involve dirty information, known as 1) inconsistent labeling and 2) unsound structure. (A structure is said unsound if it cannot find any labeling conforming to the specification, see Example 3 for instance). Unsound structure may be raised owing to ad-hoc re-assignment of operators, e.g., a task is assigned to a successor J. Zhe but actually executed by the operator O. Chu. Such structural inconsistency needs business actors to manually handle. Inconsistent labeling of event names, however, typically occurs owing to mistakenly recording. Therefore, it is highly desirable to efficiently detect unsound structure, and repair the inconsistent labeling with sound structure. According to our survey in a real dataset (see details in Table I in the experiments), among traces with detectable inconsistencies3, about 5.42% are raised by unsound structure, while the others (about 94.58%) are structurally sound but with inconsistent labeling.

In this paper, we study the two problems of cleaning event data, 1) detecting unsound structure; or 2) returning a repair of event names if the structure is sound. That is, while reporting all detectable inconsistencies, we also attempt to remedy the majority (94.58%) of inconsistencies as accurate as possible.

Challenges: The major challenges of detecting and repairing dirty event data originate from coupling of data and logic.

Existing database repairing techniques [19] cannot handle the complex structural relationships, e.g., t2[Successor]=J. Zhe & O. Chu denoting the follow-up relationships among t2, t3, t4. Moreover, the constraints specified by process specifications are very different from integrity constraints in relational data. In particular, data dependencies declare relationships in tuple pairs, while process specifications indicate constraints on events with flow directions, AND/XOR semantics.

Adapting the existing graph relabeling technique [27], by treating execution and specification as simple graphs, falls short in three aspects: 1) the AND/XOR semantics are not considered; 2) the vertex contraction technique in [27] modifies the structure of execution and thus cannot detect unsound structure; 3) [27] proposes only approximate algorithms for large graphs, while event traces are usually small and exact computation may apply.

Contributions: To the best of our knowledge, this is the first study on detecting and repairing inconsistencies in event logs with structural information. Our major contributions in this paper are summarised as follows.

1) We propose an exact repairing algorithm to either provide the optimal repair of an execution trace or conclude unsound structure. Branch and bound algorithms, together with several efficient pruning techniques, are devised.

2) We develop an efficient PTIME approximation algorithm, by only one pass through the transitions in the execution trace. Although it may generate false negative regarding the detection of unsound structure and may not be able to guarantee the optimal repairing, the performance studies show that it can achieve good accuracy while keeping time cost extremely low.

3) We report an extensive experimental evaluation to demonstrate the performance of proposed methods. Repairing accuracies of our exact approaches (greater than 90% in all the tests) are significantly higher than the state-of-the-art sequence-based [11] and graph-based [27] methods. The one pass approximation also shows a good accuracy higher than 70% in most tests, with relative approximation ratio less than

3Other errors, that are consistent w.r.t. the specification, are unlikely to be detected without further knowledge and are not in the scope of this study.
1.5. For time performance, the one pass algorithm can achieve an improvement of at most 3 orders of magnitude compared with exact approaches, in both real and synthetic data sets.

The rest of the paper is organized as follows. We introduce preliminaries in Section II. Major results of two detecting/reparing algorithms are presented in Sections III and IV, respectively. Section V provides an experimental evaluation. Finally, we discuss related work in Section VI and conclude the paper in Section VII.

II. PROBLEM STATEMENT

We first formalize syntax and definitions for process specifications and executions. Conformance checking is then introduced, which raises the detecting and repairing problems.

A. Preliminary

For a function \( f \) and a set \( A \), let \( f(A) \) denote \( \{ f(x) \mid x \in A \} \).

**Definition 1.** A Petri net is a triplet \( N = (P, T, F) \), where \( i) \) \( P \) is a finite set of places, \( ii) T \) is a finite set of transitions, \( P \cap T = \emptyset \), \( iii) F \subseteq (P \times T) \cup (T \times P) \) is a set of directed arcs, namely flow relation.

A net is a bipartite directed graph, with set \( F \) of edges between nodes in \( P \) and \( T \). Each \( (x, y) \in F \) is a directed arc from node \( x \) to node \( y \). For any \( x \in P \cup T \), let \( \text{pre}_F(x) = \{ y \mid (y, x) \in F \} \) be the set of all input nodes of \( x \) and \( \text{post}_F(x) = \{ y \mid (x, y) \in F \} \) denote the set of all output nodes of \( x \).

**Definition 2.** A process specification is a Petri net \( N(P, T, F) \) such that \( i) P \) contains a source place having \( \text{pre}_F(\text{start}) = \emptyset \), \( ii) P \) contains a sink place having \( \text{post}_F(\text{end}) = \emptyset \).

**Definition 3.** A causal net is a Petri net \( N = (P, T, F) \) such that for every \( p \in F \), \( |\text{pre}_F(p)| \leq 1 \) and \( |\text{post}_F(p)| \leq 1 \).

It is easy to see that there will be no XOR-split or XOR-join in a causal net (according to the maximum in/out degree 1 of places), while AND-split and AND-join are allowed. If we interpret places as edges connecting two transitions, the net is indeed a directed acyclic graph of transitions [18].

**Definition 4.** An execution of a process specification \( N_s(P_s, T_s, F_s) \) is denoted by \( (N_s, \pi) \), where \( N_s(P_s, T_s, F_s) \) is a causal net and \( \pi \) is a labeling \( \pi : P_s \cup T_s \rightarrow P_s \cup T_s \) such that \( \pi(P_s) \subseteq P_s \) and \( \pi(T_s) \subseteq T_s \).

We use \( y : Y \) to denote \( \pi(y) = Y \) for short, where \( y \) is a transition/place in \( N_s \) mapping to a transition/place \( Y \) in \( N_s \) via \( \pi \), e.g., \( \pi(1) = \text{submit} \) denoted by \( t_1 \).\text{submit} in Figure 1.

**Definition 5.** We say an execution \( (N_s, \pi) \) conforms to a process specification \( N_s \), denoted by \( (N_s, \pi) \models N_s \), if only if \( i) \) \( \pi(P_s) \subseteq P_s \) and \( \pi(T_s) \subseteq T_s \); \( ii) \) for any \( t \in T_s \), \( \pi(\text{pre}_{F_s}(t)) = \text{pre}_{F_s}(\pi(t)) \) and \( \pi(\text{post}_{F_s}(t)) = \text{post}_{F_s}(\pi(t)) \); \( iii) \) for any \( p \in P_s \), \( \pi(\text{pre}_{F_s}(p)) = \emptyset \) implies \( \pi(p) = \text{start} \) and \( \text{post}_{F_s}(\pi(p)) = \emptyset \) indicates \( \pi(p) = \text{end} \).

That is, there is a bijection between \( \text{pre}_{F_s} \) and \( \text{pre}_{F_s} \) for each transition \( t \) in the execution \( (N_s, \pi) \), and similarly, for \( \text{post}_{F_s} \) and \( \text{post}_{F_s} \).

<table>
<thead>
<tr>
<th>Event</th>
<th>Name</th>
<th>Operator</th>
<th>Successor</th>
<th>Conf</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>submit</td>
<td>A</td>
<td>B</td>
<td>1.0</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>design</td>
<td>B</td>
<td>C &amp; D</td>
<td>0.6</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>insulation proof</td>
<td>C</td>
<td>E</td>
<td>0.7</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>check inventory</td>
<td>D</td>
<td>E</td>
<td>0.1</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>evaluate</td>
<td>E</td>
<td>F</td>
<td>0.9</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>archive</td>
<td>F</td>
<td>—</td>
<td>0.8</td>
</tr>
</tbody>
</table>

(a) Example of an execution trace

(b) Representing execution as causal net

Fig. 2: Example of conformance

B. Execution Trace

In practice, execution is stored as execution trace \( \sigma \), with schema (Event, Name, Operator, Successor,...). Each tuple in \( \sigma \), a.k.a. an event, denotes a transition in execution \( t_i \in T_\sigma \), ordered by execution timestamp, e.g., the \( i \)-th executed event/transition \( \sigma(i) = t_i \).

By the labeling \( \pi \), each event \( t_i \) in \( T_\sigma \) is associated with a name \( \pi(t_i) \), which usually corresponds to a type in the specification \( N_s \), i.e., \( \pi(t_i) \in T_s \).

Execution trace also records the net structure of execution. As there is no XOR-split or XOR-join in the causal net of an execution, each place \( p_j \) in \( \text{pre}_{F_s}(t_i) \) corresponds to exactly one transition, say \( \text{pre}_{F_s}(p_j) = \{ t_j \} \). Combining \( t_j \) of all \( p_j \) forms \( \text{pre}_{F_s}(\pi(t_i)) \), namely the prerequisite of \( t_i \).

**Proposition 1.** For any \( \sigma(j) = t_j, \sigma(i) = t_i, j < i \) in a trace \( \sigma \), it always has \( t_i \notin \text{pre}_{F_s}(\pi(t_j)) \) and \( t_j \notin \text{post}_{F_s}(\pi(t_i)) \).

Thus, no \( t_i \) can appear before its prerequisite \( t_j \) in a trace \( \sigma \).

Conformance of execution trace can be checked by recovering its corresponding causal net, i.e., recovering places (and labeling) between a transition and its prerequisite (as places are not recorded in execution trace).

Example 2 (Example 1 continued). Consider another execution trace in Figure 2(a) over the specification in Figure 1(a). We represent the corresponding causal net in Figure 2(b) as follows. For the first \( t_1 \) without any prerequisite, we put a place \( p_0 \) with \( \pi(p_0) = \text{start} \) as the pre set. The second \( \pi(2) \) of \( t_2 \) has prerequisite \( \text{pre}_{F_s}(\pi(t_2)) = \{ t_1 \} \). We recover the labeling of the place \( p_1 \) between \( t_2 \) and its prerequisite \( t_1 \) to the place between \( \pi(t_2) \) and \( \pi(t_1) \) in the specification, i.e., \( \pi(p_1) = \text{check} \). Similarly, considering the prerequisites of \( t_5 \), \( \text{pre}_{F_s}(\pi(t_5)) = \{ t_3, t_4 \} \), we obtain \( \pi(p_5) = \text{check} \). For the last \( t_6 \), which is not prerequisite of any others, a place \( p_7 : \text{end} \) is appended as \( \text{post}_{F_s}(t_6) \).

Referring to conformance definition, for any transition, say \( t_1 \) for instance, we have \( \pi(\text{pre}_{F_s}(t_1)) = \pi(p_0) = \{ \text{start} \} = \text{pre}_{F_s}(\text{submit}) = \text{pre}_{F_s}(\pi(t_1)) \) and \( \pi(\text{post}_{F_s}(t_2)) = \pi(\{ p_2, p_4 \}) = \{ b,d \} = \text{post}_{F_s}(\text{design}) = \text{post}_{F_s}(\pi(t_2)) \).

We consider two types of inconsistencies, unsound structure
and inconsistent labeling, which injure the conformance.

C. Unsound Structure Detection Problem

We say that a causal net $N$ is unsound w.r.t. the specification $N_s$, if there does not exist any labeling $\pi$ such that $(N, \pi)$ forms an execution conforming to $N_s$.

**Problem 1.** Given an execution $(N, \pi)$ over the specification $N_s$, the unsound structure detection problem is to determine whether there exists a labeling $\pi'$ such that $(N, \pi') \models N_s$.

We say that structure is sound if there exists at least one labeling $\pi'$ to make the conformance.

**Example 3** (Example 1 continued). Consider another execution trace in Figure 3 over the specification in Figure 1(a). As shown in the recovered causal net, the second $t_2$ involves inconsistency that $\pi(p_3) = \pi(p_1) \neq \{b,d\}$.

To accomplish the work specified in Figure 1(a), at least two transitions should be processed which take $t_2$ as prerequisites. However, only one transition $t_3$ in the causal net in Figure 3 has prerequisite $t_2$. It is impossible to find any labeling $\pi'$ that can make conformance to the specification.

While we can detect unsound structure, as mentioned, handling unsound structure is not the focus of this study.

D. Inconsistent Labelling Repair Problem

For an execution trace with sound structure, we can repair the inconsistent labeling of events. Repairing the execution can be viewed as the relabeling of transitions (and places) from $N$ to the specification $N_s$. The new labeling, say $\pi'$, should meet the conformance requirement.

**Cost Model:** As discussed in the introduction, along the same line of database repairing [7], a typical principle is to find a repair that minimally differs from the original data.

Let $(N, \pi')$ be a repaired execution of the original $(N, \pi)$ by changing the labeling function from $\pi$ to $\pi'$ such that $(N, \pi') \models N_s$. The event name repairing cost is given by

$$\Delta(\pi, \pi') = \sum_{t \in T_s} \delta(\pi(t), \pi'(t)),$$

where $\pi'(t)$ is the new (type) name of the transition (event) $t$ in the repaired $(N, \pi')$, and $\delta(\pi(t), \pi'(t))$ denotes the cost of replacing $\pi(t)$ by $\pi'(t)$.

As shown in Figure 2(a), additional information may also be attached in the table of execution trace, e.g., the confidence of an event being correctly recorded by the executor. The confidence field is optional and analogous to the confidence of each tuple in database repairing [7]. Intuitively, a higher confidence $conf(t)$ indicate a larger cost of $t$ being repaired.

Frequency is an observation of “user behaviors” and may help in repairing. For instance, in Figure 1 (a), if insulation proof appears much more frequently than electrician proof in the database of all execution traces, we may repair $t_2$ by insulation proof. The cost of repairing a high frequency $freq(\pi(t))$ to a low frequency $freq(\pi'(t))$ is large.

Thereby, the cost $\delta$ can be defined as, but not limited to,

$$\delta(\pi(t), \pi'(t)) = conf(t) \cdot dis(\pi(t), \pi'(t)) \cdot \frac{freq(\pi(t))}{freq(\pi'(t))}$$

where $conf(t)$ is the confidence associated to transition $t$ as the example illustrated in Figure 2(a), $dis(\pi(t), \pi'(t))$ denotes the metric distance between two names $\pi(t)$ and $\pi'(t)$, e.g., edit distance, and $freq(\pi(t))$ and $freq(\pi'(t))$ are the frequencies of $\pi(t)$ and $\pi'(t)$, respectively, appearing in different execution traces in the event database.

**Problem 2.** Given an execution $(N, \pi)$ over the specification $N_s$, the inconsistent labeling repairing problem is to find a relabeling $\pi'$ with the minimum repairing cost $\Delta(\pi, \pi')$ such that $(N, \pi') \models N_s$, if it exists.

III. EXACT ALGORITHM

Both detecting and repairing problems can be solved by an algorithm of attempting to find the minimum repair. If no valid repair is found, the input execution trace is detected as unsound structure. In this section, a practical branch and bound algorithm is developed for computing exact solutions. We also propose advanced bounding functions and pruning techniques to further improve the efficiency.

A. Branch and Bound

We first briefly describe the idea of computing repairs. For each transition in a given execution trace, there may be multiple candidates for repairing. In order to generate the optimal repair, we should theoretically consider all the repairing alternatives, each of which leads to a branch of generating possible repairs. The repairing must roll back to attempt the other branches in order to find the minimum cost one. Intuitively, by trying all the possible branches, we can find the exact solution.

**Overview:** Starting from the first transition in the execution trace $\sigma$, in each step, we will consider all the possible repairs for a transition $t_k$, each of which leads to a repairing branch.

For any node in the branching graph, let $\sigma_k$ denote the first $k$ transitions in $\sigma$ that have been repaired by $\pi'$. As we will present soon, a lower bound of least cost for repairing the remaining transitions in $\sigma \setminus \sigma_k$ can be computed, to form a valid repair. That is, we can compute a bound of repairing cost $LB(\sigma_k, \pi')$ for all the possible repairs generated in the branches w.r.t. $\sigma_k$. A simple bounding function can be $LB(\sigma_k, \pi') = \Delta(\pi, \pi')$, i.e., the cost that has already been paid in the repairing $\pi'$ for the first $k$ transitions in the trace.
Lemma 2. For the current branching for each repairs (Line 14) of \( Q \) in \( \pi \), will not be modified in the current branching. As illustrated in \( \sigma \) satisfies \( \pi \) of the current solution, the result is the first valid solution with the determination of \( Q \) element left in remaining nodes in \( Q \) repair, in Line 6, we directly return it as the result. As the \( \sigma \) say repairing the trace, i.e., \( \pi \). The branching continues in the remaining transitions of \( \sigma \) until it forms a valid repair (e.g., node 4 for all 6 transitions in \( \sigma \) or no further repairing can be applied such as node 3. Suppose that the repairing cost \( \Delta(\pi, \pi') \) of node 4 is 30 (computed by string edit distance on event names). Then, all the branches on node 5 with bound 31 can be safely pruned.

Algorithm: Algorithm 1 presents the procedure of branch and bound repairing. We maintain a priority queue \( Q \), where each element \( (\sigma_k, \pi') \) denotes a node or state of branching. As shown in Line 3, each step fetches an element from \( Q \), say \( \sigma_{k-1} \) together with its repair \( \pi' \), which has the minimum \( LB(\sigma_{k-1}, \pi') \). If the current \( \pi' \) has already formed a valid repair, in Line 6, we directly return it as the result. As the remaining nodes in \( Q \) must have a lower bound no less than the current solution, the result is the first valid solution with the minimum cost.

Otherwise, we keep on branching to the next transition \( t_k \). According to Proposition 1, it ensures that all the prerequisites of \( t_k \) are in \( \pi_{k-1} \).

Lemma 2. For the current branching for each \( t_k \), it always satisfies \( \pre_F(\pi(\pre_F(\pi(\tau))) \subseteq \pi_{k-1} \).

That is, the transitions in \( \pi_{k-1} \) has already been repaired and will not be modified in the current branching. As illustrated in Figure 5, the prerequisites of \( t_k \) determine the possible assignments of places in \( \pre_F(\pi(\tau)) \), i.e., Lines 10-12 in Algorithm 1. The determination of \( \pi'(p_i) \) for each \( p_i \in \pre_F(\pi(\tau)) \) will be presented below. Consequently, for each labeling \( \pi'(p_i) \) on places in \( \pre_F(\pi(\tau)) \), we can enumerate the corresponding possible repairs (Line 14) of \( t_k \) for branching (Line 18).

Finally, the while iteration terminates when there is no element left in \( Q \). The returned results can be either the optimal repair or the identification of unsound structure. The correctness of conformance of the returned repair is guaranteed by Line 5 in Algorithm 1.

Algorithm 1: EXACTBB(\( N_s, \pi, N_b \))

Input: An execution \( (N_s, \pi) \) and a specification \( N_b \)

Output: An optimal repair \( \pi' \) with the minimum repairing cost such that \( (N_s, \pi') \models N_b \)

1: \( Q := \{\emptyset, \pi\} \)
2: while \( Q \neq \emptyset \) do
3: \( (\sigma_{k-1}, \pi') := \min_{\sigma, \pi} LB(\sigma, \pi') \in Q \)
4: \( Q := Q \setminus \{(\sigma_{k-1}, \pi')\} \)
5: if \( (N_s, \pi') \models N_b \) then
6: return \( \pi' \)
7: else
8: \( t_k := \sigma(k) \) the k-th transition in the execution trace
9: \( \sigma_k := \sigma_{k-1} \cup \{t_k\} \)
10: for each \( p_i \in \pre_F(\pi(\tau)) \) do
11: \( P_i := \) all valid labeling \( \pi'(p_i) \) of \( p_i \).
12: \( \Lambda := P_i^1 \times \cdots \times P_i^{\pre_F(\pi(\tau))} \)
13: for each labeling \( \pi'(p_i) \) in \( \Lambda \) on places \( \pre_F(\pi(\tau)) \) do
14: \( T_c := \{p_i \in \pre_F(\pi(\tau)) | \post_F(\pi'(p_i)) \}
15: for each \( X \in T_c \) do
16: if \( \pi'(p_i(\tau)) = \post_F(\pi'(p_i)) \) then
17: \( \pi'(t_k) := X \)
18: \( Q := Q \cup \{(\sigma_k, \pi')\} \)
19: return unsound structure

B. Generating Branches

Recall that each branch w.r.t. the current transition \( t_k \) corresponds to a possible repairing \( \pi'(t_k) \). As illustrated in Figure 5, to determine \( \pi'(t_k) \), we need to first identify the labeling on the places in the pre set of \( t_k \).

Let us consider any \( p_i \in \pre_F(\pi(\tau)) \). Referring to the definition of causal net, we have a unique transition, say \( t_i \), in the pre set of \( p_i \), denoted as \( \pre_F(\pi(\tau)) \). This \( t_i \) must belong to \( \sigma_{k-1} \) according to Lemma 2, where the repair \( \pi'(t_i) \) has been given. As illustrated in Line 11 in Algorithm 1, we can find a set \( P_i^{\pre_F(\pi(\tau))} \) of all valid labeling \( \pi'(p_i) \) of \( p_i \) that are consistent with \( \pi'(t_i) \). There are several scenarios to consider for determining \( P_i^{\pre_F(\pi(\tau))} \):

Case 1. If \( \post_F(\pi(\tau)) \subseteq \sigma_k \), then we have \( P_i^{\pre_F(\pi(\tau))} := \post_F(\pi'(t_i)) \). That is, there exists at least one transition, whose prerequisite is \( t_i \), but not belonging to \( \sigma_k \), e.g., \( t_{k+1} \) following \( t_k \) in Figure 5 that has not been repaired in \( (\sigma_{k-1}, \pi') \). We can assign any \( \pi'(p_i) \) in \( \post_F(\pi'(t_i)) \) without introducing inconsistencies to \( t_i \) in the current stage.

Case 2.1. If \( \post_F(\pi(\tau)) \subseteq \sigma_k \), and

\[ \pi'(t_i) \setminus \{p_i\} = \post_F(\pi'(t_i)) \]

then we have \( P_i^{\pre_F(\pi(\tau))} := \post_F(\pi'(t_i)) \). In this (and following 2.x) case, we have all the transitions, whose prerequisite is \( t_i \), belonging to \( \sigma_k \). In other words, all the transitions, e.g., \( \post_F(\pi(\tau)) \) of \( t_{k-1} \) in Figure 5, are repaired in \( (\sigma_{k-1}, \pi') \) except \( t_k \). Moreover, the condition \( \pi'(t_i) \setminus \{p_i\} = \post_F(\pi'(t_i)) \) ensures the conformance on \( t_i \) if we ignore \( p_i \). Consequently, any assignment \( \pi'(p_i) \) in \( \post_F(\pi'(t_i)) \) will not introduce inconsistencies to \( t_i \).

Case 2.2. If \( \post_F(\pi(\tau)) \subseteq \sigma_k \), and

\[ |\pi'(t_i) \setminus \{p_i\}| = |\post_F(\pi'(t_i))| - 1 \]
then we have $P^c_i := \text{post}_{F_s}(\pi'(t_i)) \setminus \pi'(\text{post}_{F_s}(t_i) \setminus \{p_i\})$. This case differs from Case 2.1 in the variance between $\pi'(\text{post}_{F_s}(t_i) \setminus \{p_i\})$ and $\text{post}_{F_s}(\pi'(t_i))$. It states that there is only one choice of $\pi'(p_i)$, i.e., $\text{post}_{F_s}(\pi'(t_i)) \cap \pi'(\text{post}_{F_s}(t_i) \setminus \{p_i\})$, in order to make the conformance on $t_i$.

**Case 2.3.** If $\text{post}_{F_s}(\text{pre}_{F_s}(p_i)) \leq \sigma_k$, and

$$|\text{post}_{F_s}(\pi'(t_i))| - |\pi'(\text{post}_{F_s}(t_i) \setminus \{p_i\})| > 1,$$

then we have $P^c_i := \emptyset$. In this case, the difference between $\pi'(\text{post}_{F_s}(t_i) \setminus \{p_i\})$ and $\text{post}_{F_s}(\pi'(t_i))$ is at least 2. It is impossible to achieve the conformance on $t_i$ by simply repairing one place $p_i$. We ignore this case by setting $P^c_i := \emptyset$.

Thus far, we have presented the assignment of each place $p_i$ in pre$_{F_s}(t_k)$. Considering all the $r = |\text{pre}_{F_s}(t_k)|$ places, we can enumerate all the labeling $\pi'$ on places $\text{pre}_{F_s}(t_k)$, i.e., $\Lambda := P^c_{\pi} \times \cdots \times P^c_{\text{pre}_{F_s}(t_k)}$ in Line 12 in Algorithm 1. For each labeling $\pi'$ in $\Lambda$, there is a set of candidate repairs for $\pi'(t_k)$, denoted by $T_c := \bigcap_{p_i \in \text{pre}_{F_s}(t_k)} \text{post}_{F_s}(\pi'(p_i))$ in Line 14. Finally, any $\pi'(t_k)$ in $T_c$, which satisfies the conformance requirement $\pi'(\text{pre}_{F_s}(t_k)) = \pi'(\text{post}_{F_s}(t_k))$, generates a possible branch $(\sigma_k, \pi')$, and is added into $Q$ in Line 18.

**Example 5** (Example 1 continued). Let $t_5$ in Figure 1 be the currently considered $k$-th transition in Line 8 of Algorithm 1. Line 11 computes all valid labeling (w.r.t. pre-requisites of $t_5$) for the places in $\text{pre}_{F_s}(t_4)$, by considering the aforesaid possible cases, i.e., $\{C\}$ for $p_3$ referring to $\pi'(t_3)$ = installation proof and $\{E\}$ for $p_5$ referring to $\pi'(t_4)$ = check inventory. The labeling of places $p_3, p_5$ suggests possible candidates for branching $t_5$, in Line 14, having $T_c = \text{post}_{F_s}(C) \cap \text{post}_{F_s}(E) = \{\text{evaluate}\}$. By considering the next branching step iteratively, on $t_6$, since there is no violation left, the program returns the result in Line 6.

**Algorithm Analysis:** Note that pre and post sets of a transition lead to parallel processes. In most processes, the number of parallel flows of a transition is often small and can be regarded as a constant$^4$ [25]. Let $b$ and $d$ be the maximum sizes of the pre/post set of any node in the specification and execution, respectively. We have $O(b^d)$ possible labelings in $\Lambda$, each of which corresponds to $b$ repairing candidates in $T_c$, i.e., total $O(b^{d+1})$ repairs for $t_k$. Consider the branches of possible combinations on $n$ transitions. The worst case complexity of Algorithm 1 is $O(b^{d+1}n)$, exponential to $n$.

$^4$Although a process may theoretically consist of a large number of parallel flows, in practice, techniques are often applied to keep the process as simple as possible, such as minimize the routing paths per element [25]. According to our survey, the maximum number of parallel flow is 4 in the dataset from SAP Reference Models [10], which includes 69 typical workflow specifications.

### C. Pruning Invalid Branches

It is worth noting that not all the branches can eventually generate a valid repair (e.g., node $\mathcal{C}$ in Figure 4). We call the branches that cannot form a valid repair invalid branches. The earlier the algorithm could identify invalid branches, the better the repairing performance will be. However, the aforesaid repairing method will not terminate branching until the last step, i.e., no further repairing can perform on a transition.

The intuition of early termination for invalid branches comes from the scenario of unsound structure. If the maximum length path from the current transition $t_k$ to end in the causal net is shorter than the minimum length path from $\pi'(t_k)$ to end in the specification, modifying transitions after $t_k$ will form an invalid repair.

Pruning of invalid branches can be deployed before Line 18 in Algorithm 1. Specifically, in the preprocessing, for each transition $t_j$ in the specification, we can find a shortest path from $t_j$ to end, denoted by $SP_{t_j}(t_k)$. Moreover, since any causal net can be viewed as a directed acyclic graph, we can find the longest path from any transition $t_i$ to end, say $LP_{t_i}(t_k)$. It can be computed by running a shortest-path finding algorithm with negative weights.

**Proposition 3.** A branch $(\sigma_k, \pi')$ with $LP_{t_i}(t_k) < SP_{t_i}(\pi'(t_k))$ is an invalid branch that cannot form any valid repair with the current labeling $\pi'$ on $\sigma_k$.

According to the proposition, for any $t_k$ in Line 18 in Algorithm 1, if $LP_{t_i}(t_k) < SP_{t_i}(\pi'(t_k))$, we will not add this $(\sigma_k, \pi')$ to $Q$. That is, $(\sigma_k, \pi')$ is pruned as an invalid branch.

**Example 6** (Example 3 continued). In Figure 3, let $t_1$ be the current transition with $\pi'(t_1) = \text{submit}$. The maximum length from $t_1$ to the end place $p_3$ is 2 (2 transitions), while the specification shown in Figure 1(a) needs to process at least 5 transitions (with a minimum length 5) to go to the end place from design. Consequently, we can directly conclude that the branch with respect to $\pi'(t_1) = \text{submit}$ is invalid, i.e., unable to find a repair $\pi''$ for the causal net with $\pi''(t_1) = \text{submit}$.

### D. Advanced Bounding Function

The lower bound of repairing cost $LB(\sigma_1, \pi)$ is essential in pruning branches. Before introducing the advanced bounding function $LB$, we first investigate the lower bound of cost for repairing an execution. Let $LC(N_\sigma, \pi)$ denote the least cost of repairing $(N_\sigma, \pi)$. As mentioned, a naive bound is $LC(N_\sigma, \pi) = 0$, as any repair $\pi'$ must have $\Delta(\pi, \pi') \geq 0$. Indeed, as discussed below, such a naive bound will yield a bounding function $LB$ with weaker pruning power.

To obtain a reasonable bound of least cost for repairing $(N_\sigma, \pi)$, we build a conflict graph $G$ with transitions in $T_\sigma$ as vertexes. For any place $p \in P_\sigma$, let $\text{pre}_{F_p}(p) = \{t_j\}$ and $\text{post}_{F_p}(p) = \{t_j\}$. If $\text{post}_{F_p}(\pi'(t_i)) \cap \text{pre}_{F_p}(\pi'(t_j)) = \emptyset$, i.e., at least one of the transitions $t_i, t_j$ needs to be repaired, we put a conflict edge $(t_i, t_j)$ in $G$. Each vertex $t_i$ is associated with a weight $\omega(t_i) = \min_{x \in T_\sigma} \delta(\pi'(t_i), x)$, i.e., the minimum cost on all possible repairs of $t_i$. 

![Fig: Generating a branch](image-url)
To eliminate inconsistencies, at least one transition of each edge in $G$ should be repaired. The minimum weighted vertex cover of $G$ with total weight $VC^*(G)$ can be interpreted as a lower bound of least cost $LC(N_\sigma, \pi)$, i.e., $VC^*(G) \leq \Delta(\pi, \pi')$ for any repair $\pi'$. As computing the exact minimum vertex cover is unlikely to be efficient, we relax the bound as follows. Consider a set $E = \emptyset$ initially. We repeatedly add an edge say $(t_i, t_j)$ of $G$ into $E$, and remove $t_i, t_j$ and all the edges incident on $t_i$ or $t_j$, until there is no edge left in $G$. Consequently, no two edges in $E$ share the same vertex. As each edge should be covered by at least one vertex from the minimum vertex cover, we have $\sum_{(t_i, t_j) \in E} \min\{w(t_i), w(t_j)\} \leq VC^*(G)$. Considering the relationship between vertex cover and repairing, it follows:

**Lemma 4.** For any valid repair $\pi'$, we have

$$\sum_{(t_i, t_j) \in E} \min\{w(t_i), w(t_j)\} \leq VC^*(G) \leq \Delta(\pi, \pi').$$

Hence, we define the lower bound of the least cost for repairing by $LC(N_\sigma, \pi) = \sum_{(t_i, t_j) \in E} \min\{w(t_i), w(t_j)\}$. Note that each $(\sigma, \pi')$ divides the transitions into two parts, $\sigma_i$ and its complement $\sigma \setminus \sigma_i$, denoted as $\bar{\sigma}$. We consider $N_{\sigma_i}(P_{\sigma}, T_{\sigma}, F_{\sigma_i})$ as a projection or partition of the net on transitions $T_{\sigma_i} \subseteq T_\sigma$ corresponding to the remaining execution trace $\bar{\sigma}$. As $\pi'$ only specifies the repairing of the current $\sigma_i$ transitions in $\bar{\sigma}$, they have not been reassigned by $\pi'$ yet.

**Lemma 5.** We have $\pi'(t) = \pi(t), \forall t \in \sigma \setminus \sigma_i$.

Finally, the lower bound is defined as

$$LB(\sigma_i, \pi') = \Delta(\pi, \pi') + LC(N_{\sigma_i}, \pi),$$

which consists of the repairing cost $\Delta(\pi, \pi')$ that has been made on $\sigma_i$, and the least cost of repairing the remaining $\bar{\sigma}$. The larger the lower bound, the higher the power will be in pruning branches. We call this $LB(\sigma_i, \pi')$ with $LC(N_{\sigma_i}, \pi) = \sum_{(t_i, t_j) \in E} \min\{w(t_i), w(t_j)\}$ the advanced bounding function. It is not surprising that the aforesaid simple bounding function with the naive bound $LC(N_{\bar{\sigma}}, \pi) = 0$ shows weaker pruning power.

**Example 7** (Example 1 continued). Let $\sigma_1$ with one transition $t_1$ in Figure 1 be the currently repaired transitions. Since no transition is changed so far, we have $\Delta(\pi, \pi') = 0$. For the remaining transitions $t_2t_3t_4t_5t_6$, i.e., $\bar{\sigma}$, a conflict graph is constructed with edges $(t_2, t_3), (t_2, t_4), (t_3, t_5), (t_4, t_5)$. Suppose that $(t_2, t_3)$ and $(t_4, t_5)$ is chosen to $E$ for Lemma 4, and $t_2, t_3$ has smaller minimum cost, say $w(t_2) = 3$ and $w(t_3) = 0$. By removing $(t_2, t_3)$ and $(t_4, t_5)$, there is no edge left in the conflict graph. We have $LB(\sigma_1, \pi') = LC(N_{\bar{\sigma}}, \pi) = w(t_2) + w(t_3) = 3$ higher than the simple bound 0.

**IV. APPROXIMATION**

Although several advanced pruning techniques are proposed, the exact algorithm is still too costly to meet the fast repairing requirement, such as in online systems. As event data are continuously generated, the online repairing may only allow one pass through the events (transitions) in executions. In this section, to support fast repairing, we introduce several heuristics for approximation and present a one-pass algorithm.

The idea of approximate repairing is to repair one transition at a time, and repeat until all violations are eliminated or no repairing can be further conducted. As each step of repairing a transition may introduce inconsistencies to others, we heuristically choose a revision that will have least violations to others. Let us first investigate this intuition on how to repair one transition regarding violation elimination.

**A. Heuristic for Repairing a Transition**

Consider any transition $t_k \in T_\sigma$ in an execution $(N_\sigma, \pi)$ whose current labeling $\pi(t_k)$ is inconsistent with the specification $N_\pi$, that is, having either $\pi(\text{pre}_{t_k}(t_k)) \neq \pi(\text{pre}_{t_k}(t_k))$ or $\pi(\text{post}_{t_k}(t_k)) \neq \pi(\text{post}_{t_k}(t_k))$. The repairing is to find a new labeling $\pi'(t_k)$ that can eliminate the inconsistency to $t_k$. We consider possible candidates for repairing $t_k$. Intuitively, in order to reduce the repairing cost, we prefer the repairing of $t_k$ with least inconsistencies introduced to other transitions.

Given any labeling $\pi'$, we define the number of violations to a transition $t_k$ as follows. Recall that any place $p$ in a causal net always has $|\text{pre}_p(p)| \leq 1$ and $|\text{post}_p(p)| \leq 1$. For any $p_i \in \text{pre}_{t_k}(t_k)$, we have either $\pi(\text{pre}_{t_k}(p_i)) = \emptyset$ (i.e., $\text{start}$) or $\text{pre}_{t_k}(p_i) = \{t_1\}$, a unique transition (prerequisite) in the pre set of $p_i$. We count place $p_i \in \text{pre}_{t_k}(t_k)$ as one violation to $\pi'(t_k)$ if $\pi'(p_i) \notin \text{post}_{t_k}(\pi'(t_k))$. For the case of $\text{pre}_{t_k}(p_i) = \emptyset$, $\pi'(p_i)$ can only be mapped to $\text{start}$ with no violation introduced. By considering the symmetric violations to the post set of $t_k$, the total violation count introduced by $\pi'$ on $t_k$ is given by $\tau(t_k, \pi') = |\{p_i \in \text{pre}_{t_k}(t_k) \mid \pi'(p_i) \notin \text{post}_{t_k}(\pi'(t_k)), \pi'(p_i) \neq \text{start}\}| + |\{p_j \in \text{post}_{t_k}(t_k) \mid \pi'(p_j) \notin \text{pre}_{t_k}(\pi'(t_k)), \pi'(p_j) \neq \text{end}\}|$.

Consequently, we need to find a $\pi'$ such that $\tau(t_k, \pi')$ is minimized. If $\tau(t_k, \pi') = 0$, the repair $\pi'$ is a perfect repairing without introducing any new inconsistencies to others.

![Fig. 6: Repairing one transition $t_k$](image)

**Example 8** (Example 1 continued). Consider $t_2$ in Figure 1 with $\pi(t_2) = \text{do revise}$. For $p_1 \in \text{pre}_{t_2}(t_2)$, we cannot find any labeling $\pi'(p_1)$ such that $\pi'(p_1) \in \text{pre}_{t_2}(\pi'(t_2))$. Thereby, $p_1$ is counted as a violation towards $\pi(t_2)$, Similarly, $p_2$ and $p_4$ are also counted having $\tau(t_2, \pi) = 3$. Suppose that $t_2$ is repaired by $\pi'(t_2) = \text{design}$. We can assign $\pi'(p_1) = \text{a}$ which belongs to $\text{pre}_{t_2}(\pi'(t_2))$, i.e., $p_1$ is no longer a violation. It reduce the violation count of $t_2$ to $\tau(t_2, \pi) = 0$. [\textcolor{red}{\blacksquare}]

**B. One Pass Algorithm**

We present an one pass algorithm of repairing one transition at a time from start to end in the execution trace $\sigma$. In each step, we determine the repair $\pi'(t_k)$ of a transition $t_k, k = 1, \ldots, |\sigma|$, and its corresponding $\pi'(p_j)$ of $p_j \in \text{post}_{t_k}(t_k)$.
Following the order of execution trace, we show that $\pi'(p_i)$ of all places $p_i \in \text{pre}_{F_c}(t_k)$ must have been determined. According to Proposition 1, all the prerequisite transitions of the current $t_k$, say $t_i \in \text{pre}_{F_c}(\text{pre}_{F_c}(t_k))$, should have been repaired. When previously repairing $t_i$, the corresponding $p_i \in \text{post}_{F_c}(t_i)$, having $\text{post}_{F_c}(p_i) = \{t_k\}$, is assigned.

Initially, only one place is processed in the causal net, i.e., the start place. In each step of $t_k$, as shown in Figure 6, all the place $p_{k-1}, \ldots, p_{k-1}$ in the pre set of $t_k$ are already determined. After repairing the transition $t_k$ (if necessary), we assign all the places $p_{k+1}, \ldots, p_{k+s}$ in the post set of $t_k$.

Finally, the program terminates when it reaches the last transition in the execution trace.

Algorithm 2 presents the pseudo-code of one pass repairing. As illustrated in Line 1, we start from the first transition $t_1 = \sigma(1)$ directly following the start place. In each iteration, Line 5 selects a transition $t_k$, i.e., the $k$-th transition $\sigma(k)$ in the execution trace $\sigma$. If there is no inconsistency with respect to $t_k$, we directly move to the next transition (Line 23); otherwise, $t_k$ needs to be repaired (Lines 7-22). As discussed, all the places in the pre set of $t_k$ must have been recovered (initially, the start place in $\text{pre}_{F_c}(t_1)$ leaves unchanged).

Hence, the repairing is to determine two aspects: $\pi'(t_k)$ in Line 11 and $\pi'(\text{post}_{F_c}(t_k))$ in Line 13. Possible candidates for these two aspects will be discussed soon. $\tau_{\text{min}}$ in Line 15 records the repairing $\pi'$ with the minimum $\tau(t_k, \pi')$, i.e., the minimum violations introduced by repairing $t_k$.

**Algorithm 2 ONEPASS($N_{\sigma}, \pi, N_s$)**

**Input:** An execution $(N_{\sigma}, \pi)$ and a specification $N_s$  
**Output:** A repair $\pi'$ such that $(N_{\sigma}, \pi') \models N_s$

1: $k:= 1$, $\pi':= \pi$
2: while $(N_{\sigma}, \pi') \not\models N_s$ do
3: \hspace{1em} if $k > |T_{\sigma}|$ then
4: \hspace{2em} return unsound structure
5: \hspace{1em} $t_k:= \sigma(k)$ the $k$-th transition in the execution trace
6: \hspace{1em} if $\pi'(\text{pre}_{F_c}(t_k)) \neq \text{pre}_{F_c}(\pi'(t_k))$ or $\pi'(\text{post}_{F_c}(t_k)) \neq \text{post}_{F_c}(\pi'(t_k))$ then
7: \hspace{2em} $T_c := \cap_{p_i \in \text{pre}_{F_c}(t_k) \text{post}_{F_c}(\pi'(p_i))}$
8: \hspace{2em} $\tau_{\text{min}} := $ a large positive integer $d$
9: \hspace{2em} for each $x \in T_c$ do
10: \hspace{2em} if $\pi'(\text{pre}_{F_c}(t_k)) = \text{pre}_{F_c}(x)$ then
11: \hspace{3em} $\pi'(t_k) := x$
12: \hspace{3em} for each complete labeling $\pi_p : \text{post}_{F_c}(t_k) \rightarrow \pi'(t_k)$ do
13: \hspace{4em} $\pi'(\text{post}_{F_c}(t_k)) := \pi_p(\text{post}_{F_c}(t_k))$
14: \hspace{4em} if $\tau(t_k, \pi') < \tau_{\text{min}}$ then
15: \hspace{5em} $\tau_{\text{min}} := \tau(t_k, \pi')$
16: \hspace{5em} $\pi_{\text{min}}(t_k) := \pi'(t_k)$
17: \hspace{5em} $\pi_{\text{min}}(\text{post}_{F_c}(t_k)) := \pi'(\text{post}_{F_c}(t_k))$
18: \hspace{4em} if $\tau_{\text{min}}$ equals to the original value $d$ then
19: \hspace{5em} return unsound structure
20: \hspace{2em} else
21: \hspace{3em} $\pi'(t_k) := \pi_{\text{min}}(t_k)$
22: \hspace{3em} $\pi'(\text{post}_{F_c}(t_k)) := \pi_{\text{min}}(\text{post}_{F_c}(t_k))$
23: \hspace{2em} $k++$
24: return $\pi'$

Correctness of conformance in the returned $\pi'$ is ensured by the condition on pre set in Line 10 and the complete labeling (defined below) for post set in Line 12 for each transition.

**Candidates for Transition $\pi'(t_k)$:** Since the places in $p_i \in \text{pre}_{F_c}(t_k)$ have already been determined, we can only choose candidates for repairing $t_k$ without introducing any inconsistency to $p_i$. For $\pi'(p_i)$ of each $p_i \in \text{pre}_{F_c}(t_k)$, we can find a set of valid post transition, $\text{post}_{F_c}(\pi'(p_i))$, in the specification $N_s$. A candidate $x$ appearing as the valid post transition of all $\pi'(p_i)$ can be a possible repairing of $t_k$, which will be consistent with all $p_i$. Hence, the candidates for repairing $t_k$ are given by $T_c := \cap_{p_i \in \text{pre}_{F_c}(t_k) \text{post}_{F_c}(\pi'(p_i))}$ as illustrated in Line 7 in Algorithm 2.

**Candidates for Places $\pi'(\text{post}_{F_c}(t_k))$:** Next, given a candidate $x \in T_c$ for $\pi'(t_k)$, we aim to determine the assignment of places $p_j \in \text{post}_{F_c}(t_k)$ such that $\tau(t_k, \pi')$ is minimized. Again, the assignment of $p_j$ should be inconsistent with $t_k$. For a fixed $\pi'(t_k) = x$, it is equivalent to find a labeling from $\text{post}_{F_c}(t_k)$ to $\text{post}_{F_c}(x)$, denoted by $\pi_p$. We say a labeling $\pi_p : \text{post}_{F_c}(t_k) \rightarrow \text{post}_{F_c}(x)$ is complete, if $\pi_p(\text{post}_{F_c}(t_k)) = \text{post}_{F_c}(x)$. This complete labeling $\pi_p$, as a candidate labeling of $\pi'$, ensures the consistency on $t_k$, i.e., $\pi'(\text{post}_{F_c}(t_k)) = \text{post}_{F_c}(\pi'(t_k))$.

All the possible complete labeling $\pi_p$ can be enumerated by considering the combination of $\text{post}_{F_c}(x)$ with repetition. Let $b$ and $d$ be the maximum sizes of the pre/post set of any node in the specification and execution, respectively. Each $p_i \in \text{post}_{F_c}(t_k)$ has $b$ choices of $\pi'(p_i)$ for repairing. Considering all $d$ places, the total number of possible labelings $\pi_p$ is bounded by $O(b^d)$.

**Example 9** (Example 1 continued). Consider the first transition $t_1$ in Figure 1. Its name submit is already consistent with $p_0 : \text{start} \in \text{pre}_{F_c}(t_1)$. For $t_2$, we can find a repairing $\pi'(t_2) = \text{design}$, and find a complete labeling $\pi_p$, e.g., $\pi_p(p_2) = b$ and $\pi_p(p_4) = d$, for $p_2, p_4 \in \text{pre}_{F_c}(t_2)$ such that the violation count on $t_2$ is minimized. Similarly, for the next $t_3$, we can find a repairing, say $\pi'(t_3) = \text{insulation}$ proof for example, and its corresponding $\pi'(p_3) = c$ in the post set, following the minimum violation count heuristic. Repairing carries on by one pass through the execution trace, and yields $\pi'(t_4) = \text{check inventory}$, $\pi'(p_5) = e$, $\pi'(t_5) = \text{evaluate}$, $\pi'(p_6) = s$.

**Algorithm Analysis:** As each place $p_i \in \text{pre}_{F_c}(t_k)$ can suggest $|\text{post}_{F_c}(\pi'(p_i))|$ (at most $b$) repairs for $t_k$, the total number of candidates in $T_c$ is bounded by $b$. Considering all the $O(b^d)$ possible labelings, we have cost $O(b^{d+1})$ for repairing one transition. The while iteration repeats at most $n$ times, $n = |T_{\sigma}|$ the number of transitions in the execution. Hence, the complexity of Algorithm 2 is $O(b^{d+1}n)$.

We select one of the alternatives for repairing a transition in the one pass algorithm, which is heuristically good but might not be optimal. The repairing results could be possibly bad, i.e., significantly differ from the original one compared to the optimal solution, as each transition may lead to a completely different flow in execution. Nevertheless, the one pass solution offers an alternative of trading time efficiency.
from repairing cost. As shown in the experiments, the one pass algorithm needs extremely low time cost while the observed approximation ratio is still considerable.

V. EXPERIMENTS

In this section, we first report a survey on the quality of a real event data set. Then, the performance of proposed repairing methods is evaluated on both effectiveness and efficiency. All programs are implemented in Java, and experiments run on a computer with 2.67GHz CPU and 8GB memory.

A. Survey on Quality of Real Data

We employ a real data set collected from a bus manufacturer. The event data are extracted from processes related to the bus design and customization affairs. The specification considered in the experiments consists of 22 transitions and 24 places with the maximum size of pre/post set 3 (the maximum parallel flows). There are 4,722 traces collected during the execution of the process. Most of the traces are small in size, in the range of 6 to 20. The maximum size observed in all the traces is no greater than 75. According to our observation, the maximum size of pre/post set in the execution is 3 as well.

In order to evaluate the quality of the collected execution traces, we observe the percentage of traces with inconsistencies. According to the statistics reported in Table I, only 14.08% of traces are consistent, i.e., exactly conform to the specification, while most traces are inconsistent, either inconsistent labelling or unsound structure. In particular, 3.85% of execution traces are irrelevant to the specification, i.e., with all the event names not from the specification.

Among 4,050 execution traces with inconsistencies (82.07%), we apply our exact repairing method. According to the results, there are 3,665 traces (77.62%) that can be repaired, while the remaining 210 traces (4.45%) are identified with unsound structure. That is, there does not exist any labeling π w.r.t. the observed causal net structure that would conform to the specification. In the following, we focus on illustrating the performance of the proposed repairing approaches, in terms of both effectiveness and efficiency.

B. Repairing Performance Evaluation

In order to study the accuracy of repairing inconsistent labelling by proposed methods, we employ 665 traces in the dataset that are correct (conforming to the specification). We randomly change event names in the traces as faults, e.g., for fault size 3, we randomly alter 3 event names in each trace (if the trace size is less than the fault size, we alter all the event names). The repairing methods are then applied to modify the execution traces to eliminate violations. We study the accuracy of the repairing results via comparison with the truth of faulty data previously replaced. For each trace, we conduct the random insertion of faults 1000 times and compute the average accuracy. Meanwhile, the repairing time performance is also reported, including the repairing over the remaining 4,050 execution traces with inconsistencies in real data.

Criteria: Let truth be the set of original correct events \((t, \pi_o(t))\) that are randomly replaced in an execution trace. Let found be the set of \((t, \pi'(t))\) that are repaired in \(\pi'\), i.e., the repairing results. To evaluate the accuracy, we use the f-measure of precision and recall [31], given by \(\text{precision} = \frac{|\text{truth}\cap\text{found}|}{|\text{found}|}, \text{recall} = \frac{|\text{truth}\cap\text{found}|}{|\text{truth}|}\), and \(f\)-measure = \(2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}\). It is natural that a higher f-measure is preferred.

Recall that the one pass method returns an approximate repairing with cost \(\Delta(\pi, \pi')\), while the exact approaches compute the optimal solutions with repairing cost \(\Delta(\pi, \pi')\), denoted by \(\Delta^*\). To study the difference between optimal and approximate solutions, we report the relative performance \((\Delta/\Delta^*)\). The closer the relative ratio to 1, the better is the approximation performance.

Besides the effectiveness evaluation, we also observe the time cost of the repairing approaches to study the efficiency performance. In particular, we have two bounding functions and a pruning technique for the exact algorithms, both of which can reduce possible repairing branches. To study the pruning power, we observe the total number of elements that have ever been put in \(Q\), i.e., elements (nodes) in the branching graph in Figure 4. Less the processed elements in the branches, the higher is the pruning power.

C. Comparison with Existing Methods

This experiment compares our proposed Exact algorithm (Exact) and approximate One Pass algorithm (OP) with the state-of-the-art techniques Graph Relabel [27] (Graph) and trace Alignment [11] (Alignment). The comparison is performed on various sizes of inserted faults in Figure 7 and various trace sizes in Figure 8.

As illustrated in Figure 7(c), the accuracies of both the Exact and OP are considerable, with f-measure no less than 0.8. Remarkably, the Exact approach have f-measures as high as 0.9. The accuracy performance of the OP method is not as stable as the exact ones. The rationale is that OP determines a heuristically good assignment as the repair of a transition without trying other alternatives like the exact algorithm. Consequently, by choosing a wrong assignment in a step, the repairing may vary in the following steps.

Accuracy of Alignment and Graph drops quickly on large fault sizes. The reason is that Alignment without exploiting structural information always chooses wrong XOR choices. Graph is originally designed for repairing simple graphs, which do not consider AND and XOR semantics. As shown in Figure 8(a), our Exact and OP keep high accuracies when the size of trace grows up, with f-measure no less than 0.8.

Figures 7(d) and 8(b) report the efficiency evaluation. It is not surprising that the repairing time cost of Exact increases
with the increase of inserted faults in Figure 7(d). According to our analysis, the Exact algorithm has exponential complexity in the number of events (transitions). Therefore, its time costs increase heavily with the increase of trace sizes in Figure 8(b). Nevertheless, OP algorithm shows significantly lower time costs (comparable to Alignment, but with higher accuracy than Alignment, especially in large fault sizes and trace sizes).

D. A Closer Look at Proposed Techniques

We compare our proposed repairing techniques in Figures 9-11, including the Exact algorithm with the Simple bounding function (ES), the Exact algorithm with the Advanced bounding function (EA), the Pruning of Invalid branches for the exact algorithm (PI), and the One Pass algorithm (OP).

In Figure 9(a), we demonstrate that the advanced bounding function (EA) can reduce the repairing time significantly compared with the simple one (ES). In order to illustrate the pruning power of different bounding functions, in Figure 9(b), we show that EA needs less elements of repairing states to be processed (i.e., the total number of nodes in Figure 4). The effectiveness of pruning on invalid branches is limited, since the traces with sound structure in this experiment have lower chance to involve invalid branches.

Similarly, as illustrated in Figure 10(a), EA method with the advanced bounding function can reduce time cost considerably, compared with ES. Again, OnePass algorithm keeps a significantly lower time cost. Indeed, the time cost in Figure 10(a) is proportional to the size of processed elements of branching states in Figure 10(b). The processed elements as well as the pruning power may not increase strictly with the trace size, owing to the structural difference in the process. Referring to the property of bounding functions, the pruning power of the advanced pruning bound is at least no worse than that of the simple one, which is also observed in Figure 10(b).

Figure 11 presents the relative performance $\Delta/\Delta^*$ of the approximate result $\Delta$ by OP and the optimal solution $\Delta^*$ by the Exact algorithm. As illustrated, the approximate result is very close to the optimal one, with relative difference no greater than 1.4. With the increase of inserted faults, both the optimal and approximate solutions have to repair most transitions. Hence, their repairing cost difference becomes small and the relative ratio $\Delta/\Delta^*$ decreases close to 1.

E. Scalability

We also report the repairing time performance over the 4050 execution traces with inconsistencies. In Figure 12(a), each point denotes the size of a trace v.s. its repairing time cost. As illustrated, the $EA$ approach with advanced bounding function can significantly reduce time costs, especially when the trace size is large. Similar to the aforesaid results, the repairing time cost is closely related to the corresponding processed elements of branching states, as shown in Figure 12(b). It is notable that the pruning method of invalid branches does not show significant improvement. The reason is that our currently employed real data set has a small portion of unsound structure traces, i.e., only 4.45% as shown in Table I. The opportunity of pruning on invalid branches is thus limited during repairing.
In order to evaluate the scalability of the proposed methods, Figure 13 reports the experiment on larger synthetic data. The synthetic event data are generated following the method in [28] by using the commonly used workflow patterns, such as parallel, sequential, and so on. Note that we can find a valid repair for most execution traces in the previous real data. In order to study the performance of unsound structure cases, the synthetic data contains 20% traces that do not exist any valid labeling. As illustrated in the results, the advanced bounding function (EA) can always show better pruning power and needs only half of the time cost of ES.Remarkably, the pruning method performs well together with both ES and EA, since it can prune the invalid branches especially in those traces that contain unsound structures. Again, the one pass algorithm can always keep significantly lower time cost.

F. Experiments on OA Dataset

We employ another real data set collected from the OA systems (implemented by Lotus Notes) of two subsidiaries in a telecom company. The specification considered in the experiment consists of 32 transitions and 31 places with the maximum size of pre/post set 3 (maximum parallel flows).

As illustrated in Figure 14(c), both the Exact and OP still achieve very high accuracies with f-measures no less than 0.95. In contrast, the f-measure of Alignment falls heavily compared with its performance in Figure 7(c). The rationale is that the dataset collected from the OA systems contains more AND splits and AND joins in specification which prevent Alignment from determining the truth execution paths. Similar with Figure 7, the accuracy of Alignment and Graph drops quickly on large fault sizes here. Figures 14(d) report the efficiency evaluation. The time cost of Exact still increases with the increase of inserted faults in Figure 14(d), and OP algorithm shows significantly lower time costs.

VI. RELATED WORK

The cooperation of business processes and data management has been emphasized for various workflow networks involving both data and flow, e.g., in Web applications, e-governance, and electronic patient records [15]. In particular, workflow techniques are useful for data management tasks such as data lineage and data provenance [5], [28]. Instead of repairing inconsistencies, the existing study assumes event data to be clean and is dedicated to improving the execution performance, i.e., optimize/accelerate the execution [22]. As described, repairing event data is indeed highly demanded and non-trivial.

Process Data Management: Studies on process data conducted by the data management community mainly focus on processing queries over workflow executions [4], [12], [13], [16]. A typical query inputs a process specification and a pattern of execution, and tries to identify all the executions that have the structure specified by the pattern. Additional conditions may be added in the query, such as type information [12] or probability [14]. Moreover, as an important application, provenance queries on workflows are well investigated [23], [1], [2], [3]. A typical provenance query calculates the transitive closure of dependencies of an event in the process data. In particular, Bao et al. [1] studied the difference provenance, i.e., computing the structural difference of two executions. Note that the repairing studied in this work employs the modification of names in events (transitions) without changing the structure. Our approaches either identify the executions with unsound structures or repair them for conformance. As the prerequisite of execution is not changed in repairing, the repairing cost is directly computed by modification count.

Database Repairing: Data dependencies or integrity constraints are often employed to eliminate inconsistencies in databases [19]. Most previous works consider equality constraints such as inclusion dependencies, functional dependencies or conditional functional dependencies [6]. The repairing aims to modify a minimum set of tuple values in order to make the revised data satisfy the given constraints [33], [7]. Although we adopt the same modification repairing, the constraints are very different between data dependencies and process specifications. In particular, the equality based data dependencies specifies groups of tuples with equal values,
which do not exist among transitions in event data. Approaches are also proposed that do not follow the minimality, such as fix with master data and edit rules [21], partial currency orders [20], or accuracy rules [8], etc. To cooperate with these art techniques, extra information is often needed such as master data or additional rules.

VII. CONCLUSIONS

To the best of our knowledge, this is the first study on considering structural information for cleaning event data. Rather than simply repairing event names in an unstructured sequence of events, the structure-based cleaning concerns inconsistencies in both structure and labeling (names) of events. While unsound structure is usually not for automatically repairing (which needs business actors to manually handle), it is highly desirable to repair inconsistent event names (as also performed by the existing sequence-based cleaning).

In this paper, we study the problems of efficiently detecting unsound structure and repairing inconsistent event names. Firstly, to repair event data with inconsistent labeling but sound structure, we follow the widely used minimum change principle to preserve the original information as much as possible. Then, we devise a novel, practically efficient exact algorithm to conduct detection and repairing dirty event data simultaneously so that it either 1) reports unsound structure or 2) gives the minimum repair of inconsistent event names. Moreover, we also present a PTIME one-pass algorithm to approximately deliver the results. Effective bounding functions and pruning techniques are carefully designed to achieve high repairing performance.

Experiments on both real and synthetic data demonstrate the effectiveness and efficiency of proposed methods. In particular, the repair accuracy of our proposal is significantly higher than the existing sequence-based repair [11] and the direct application of graph repair [27]. According to the survey on real datasets, among the execution traces with detectable inconsistencies (82.07%), most are structurally sound with repairable event names (77.62%). After detecting unsound structures by this proposal, an interesting future study is to automatically suggest possible structural explanations during the manual consultation by business owners.

ACKNOWLEDGEMENT

This work is partly supported by China NSFC under Grants 61325008, 61202008 and 61232006; ARC DP150102728, ARC DP140103578 and ARC DP120104168; an NSERC Discovery grant and a BCIC NRAS Team Project. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES