Expert Systems With Applications 59 (2016) 101–118

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Building instance graphs for highly variable processes

Claudia Diamantini[a], Laura Genga[a,*], Domenico Potena[a], Wil van der Aalst[b]

[a] Information Engineering Department, Universitá Politecnica delle Marche, via Brecce Bianche, 60131 Ancona, Italy
[b] Faculty of Mathematics and Computer Science, Eindhoven University of Technology, NL-5600 MB, Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Received 24 September 2015
Revised 16 April 2016
Accepted 17 April 2016
Available online 23 April 2016

Keywords:
Process mining
Highly variable processes
Building instance graphs
Process instances

ABSTRACT

Organizations increasingly rely on business process analysis to improve operations performance. Process Mining can be exploited to distill models from real process executions recorded in event logs, but existing techniques show some limitations when applied in complex domains, where human actors have high degree of freedom in the execution of activities thus generating highly variable processes instances. This paper contributes to the research on Process Mining in highly variable domains, focusing on the generation of process instance models (in the form of instance graphs) from simple event logs. The novelty of the approach is in the exploitation of filtering Process Discovery (PD) techniques coupled with repairing, which allows obtaining accurate models for any instance variant, even for rare ones. It is argued that this provides the analyst with a more complete and faithful knowledge of a highly variable process, where no process execution can be really targeted as "wrong" and hence overlooked. The approach can also find application in more structured domains, in order to obtain accurate models of exceptional behaviors. The quality of generated models will be assessed by suitable metrics and measured in empirical experiments enlightening the advantage of the approach.

1. Introduction

Information systems are widely adopted by organizations to support the execution of their business processes. Notable examples are Business Process Management Systems (BPMS), Workflow Management Systems (WMS), Enterprise Resource Planning (ERP), and Customer Relationship Management (CRM). These systems typically record all past process executions in an *event log*. A single execution of a business process is called *process instance*, or *case*, and it can include parallel execution of business activities. However, event logs typically store only the *trace* of a process instance, i.e., the sequence of the activities stored according to their temporal order of occurrence. Typically the start time of the activity is also recorded together with an instance identifier. Depending on the system, other information can be found as well, like the executor, input/output data, and the total duration of the activity (or, equivalently, its end time).

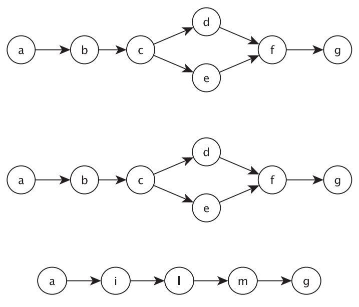
Event logs store invaluable information about organization's behavior that can be exploited by analysts to monitor and improve operations performance. Simple analyses can be performed directly on sequential traces. For instance, average instance lead time, resource usage, event interval analysis (Suriadi, Ouyang, van der Aalst, & ter Hofstede, 2015) can be obtained. However, more insights can be derived only when the control flow structure of the instance is known or, in other terms, when the instance model is known. An instance model explicitly represents parallelisms among activities that are hidden in the sequential trace, e.g., in the form of an *Instance Graph*. Fig. 1 shows a simple example of a set of traces and the corresponding Instance Graphs (IG).

In order to build instance models, causal relations among activities must be known, or they can be inferred from event logs. Process Discovery is the discipline devoted to infer relations from a log and build models. Process Discovery (PD) techniques belong to the broader *Process Mining* discipline, whose goal consists in discovering, monitoring and improving a given process exploiting event log generated during process execution (van der Aalst, 2011). Most of techniques aim to generate a complete process model. However, such a model is not always required, and a good representation of each individual process instance can provide useful insights as well, and can be used in addressing several tasks. As an example, (Lu, Fahland, & van der Aalst, 2015) proposes to exploit instance models in conformance checking applications, i.e., for measuring the extent to which actual executions adhere to a predefined normative process model. (van Beest, Dumas, García-Ba nuelos, & La Rosa, 2015) uses instance models to implement a log delta-analysis technique, which allows to identify relevant differences between the executions recorded in two event logs.

* Corresponding author.
E-mail addresses: c.diamantini@univpm.it (C. Diamantini), l.genga@univpm.it (L. Genga), d.potena@univpm.it (D. Potena), w.m.p.v.d.Aalst@tue.nl (W. van der Aalst).

http://dx.doi.org/10.1016/j.eswa.2016.04.021
0957-4174/© 2016 Elsevier Ltd. All rights reserved.

	a
1	$\langle a, b, c, d, e, f, g\rangle$
2	$\langle a, b, c, e, d, f, g\rangle$
3	$\langle a, i, l, m, g\rangle$



b

Fig. 1. A set of traces (a) and their corresponding instance graphs (b).

Another relevant application is described in our previous work (Diamantini, Genga, & Potena, 2016; Diamantini, Genga, Potena, & Storti, 2013; Diamantini, Potena, & Storti, 2012), where we introduced a methodology aimed at extracting the most relevant common substructures in a set of instance models.

In the literature, some approaches to build instance models exploit information of richly annotated logs like I/O data (an activity that needs certain data in input causally depends from activities producing that data in output), or the overlap between execution intervals (if two intervals overlap then clearly the corresponding activities has been executed in parallel). The applicability of these approaches is thus limited. Others propose the exploitation of some domain knowledge provided by experts (e.g., to publish articles, they must undergo a review cycle) or by a reference process model. In this paper, we deal with building IG from simple event logs, i.e., logs providing the minimal set of information, and without resorting to any domain knowledge. In this setting, to the best of our knowledge, only two approaches have been proposed, both based on the causal relations inference rules embedded in the α-algorithm PD technique (van Beest et al., 2015; van Dongen & van der Aalst, 2004).

It is noteworthy that the α-algorithm, and then the two above mentioned approaches, perform fairly well with well structured processes, but they show limitations when applied to unstructured, highly variable processes. This kind of processes is typical of real-world domains, where human actors have high degree of freedom in the execution of activities. This is the case, for instance, of health care where, although clinical guidelines are given, there may be good reasons to deviate from such guidelines (de Man, 2009; Rovani, Maggi, De Leoni, & Van Der Aalst, 2015). The limits of the application of α-algorithm to highly variable processes are well studied in the literature. They are related to the inference of models affected by *overgeneralization*. In practice, being unable to recognize existing causal relations among activities, the algorithm generates models involving a high number of parallel activities, thus allowing for more instances than the real ones. Similarly, the use of the algorithm to build IG produce poor results, in terms of inaccurate, overgeneralizing IG, accounting for much more traces than those actually recorded in the event log. We provide a detailed discussion of overgeneralization issues and examples in Section 4.

In this paper we propose a novel methodology that elaborates upon (van Beest et al., 2015; van Dongen & van der Aalst, 2004) to improve the quality of generated IG in the presence of highly variable processes. We propose to exploit causal relation inference rules typical of *filtering* techniques, which model only the most frequent process behaviors, thus being able to cut down overgeneralization. However, the application of a filtering technique also presents some drawbacks. In particular, (causal relations of) several traces of the event log are now not represented in the model (i.e., they are *irregular* with respect to the model), impacting negatively on the quality of the corresponding IG. While in loosely variable domains this can still be acceptable, since irregular traces can be considered the result of "wrong" or outlier behaviors, and can then be discarded, this is not the case with highly variable processes, where no execution can be really targeted as wrong and then overlooked, or in the case we like to analyze exactly those outlier behaviors. To address this issue, we also propose to couple the exploitation of filtering techniques with an original technique aimed at *repairing* IG of irregular traces, thus providing the analyst with a more complete and faithful knowledge of the reality registered in the event log.

The quality of generated models will be assessed by suitable metrics and measured in empirical experiments enlightening the advantage of the approach.

Summing up, the main contributions of this work are as follows:

- we throughly investigate and discuss the limitation of existing PD techniques to build IG in highly variable domains;
- we consider two filtering techniques, the Heuristic Miner (Weijters, van der Aalst, & De Medeiros, 2006) and infrequent Inductive Miner (Leemans, Fahland, & van der Aalst, 2014) algorithms, to induce causal relations between activities and we discuss their features in the context of present work;
- we exploit conformance checking techniques to recognize irregular traces and then we introduce a novel algorithm aimed at *repairing* irregular IG, discussing main challenges and showing its behavior by means of several examples;
- we provide experimental evidences on both synthetic and real-world logs, comparing the proposed methodology with existing approaches, and by enlightening the advantage of repairing with respect to simple filtering techniques, using accuracy and generalization capability as figures of merit.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 recalls some basic notions and definitions. Section 4 introduces the issues related to the building of IG in highly variable domains. In Section 5 we delineate our methodology, and in Section 6 experimental results are discussed. Finally, in Section 7 we draw some conclusions and delineate future work.

2. Related work

Our work aims at building an instance model, in the form of IG, for each trace of a given event log. Moreover, since IGs built for highly variable processes are often of poor quality, we introduce a repairing procedure aimed to enhance them. In the following subsections first we review some approaches which deal with building instance models; then we discuss some techniques aimed to enhance process models.

2.1. Building instance models

PD is the process mining branch aimed at extracting models from event logs. A plethora of different techniques has been developed during last years (see De Weerdt et al., 2012; van Dongen, Alves de Medeiros, and Wen, 2009, for an overview). Most of them, however, aim at deriving a process model able to describe the entire process; our proposal, instead, is focused on deriving models describing single process instances. Few approaches have been proposed to deal with this issue, which can be grouped in two categories, i.e., *model-based* and *log-based* approaches, introduced in the following subsections.

2.1.1. Model-based approaches

These approaches instantiate instance models from an a-priori known process model. This is the case, for example, of the VIP Tool (Desel, Juhás, Lorenz, & Neumair, 2003), which, given a Petri net, is able to derive the set of its *runs*, where each run represents a single, complete execution of the net. The obtained set of runs is exploited for analysis and validation of the Petri net. Other examples of building of instance models can be found in some Workflow Management Systems. Among them, we can mention the InConcert tool (van der Aalst & van Hee, 2004), which is interesting from a historic point of view. InConcert aims at ensuring a flexible management of the workflows, by supporting both the classical workflows instantiation from the overall process model and the definition of ad-hoc workflow instances. The ad-hoc instances can be obtained by modifying an instance of the model or by instantiating an empty instance, allowing the user to create the workflow instance as she needs. Furthermore, the routing of completed instances can be exploited to create new instances template. Clearly, in these approaches the definition of the instances is more a design activity, rather than an automatic discovery; indeed, the user has to specify the high level definition of the instance models.

2.1.2. Log-based approaches

Log-based approaches derive instance models starting from a given event log. They first derive the causal relations existing among process activities, and then use these relations to build the instance models. Three kinds of approaches have been proposed in literature, namely i) approaches based on special attributes stored in the event log, ii) approaches based on a-priori knowledge and, finally, iii) approaches based on basic attributes of the event log.

Examples of approaches of the first group are (Hwang, Wei, & Yang, 2004) and (Lu et al., 2015). In (Hwang et al., 2004) authors propose to take into account temporal relationships between activities. For each pair of activities a and b of a trace two types of temporal relationships are modeled, i.e., 1) the “followed” relationship, which states that b is performed after a is terminated, and 2) the “overlapped” relationship, which states that the execution of a and b is temporally overlapping. To this end, authors assume the event log to store for each activity both the starting time end the completion time, using these attributes to derive the set of temporal relationships. These relationships are then used to build for each process instance an instance model named “temporal graph”, which consists in a directed graph where each node corresponds to an activity of the trace, and an edge is inserted between two nodes only if a followed relation exists between corresponding activities. It should be noted that applicability of this approach is limited to event logs storing the actual duration of activities executions. Moreover, temporal relationships are defined locally to each trace of the log. (Lu et al., 2015) proposes an approaches that deals with so-called *Data Annotated Log* (DAL), where each event has a set of input attributes, describing data that are read when executing an activity, and of output attributes, describing data that are written by the activity. As one clearly argue, in a DAL one can define reliable ordering relations, since if an activity a_j has as input an attribute x, which is generated as output by an activity a_i, obviously a_j depends on the execution of a_i and, hence, has to be executed after it. The generation of a DAL requires either a suitable event log management system, or the preprocessing of the log by a domain expert. Many real life event logs do not include any information about data dependencies.

An example of knowledge-based approach is implemented in the ARIS PPM tool [1]. It models each process instance as *instance EPC* (i.e., Event driven Process Chain), another process modeling formalism (see e.g., Forrester, 1999). The tool is able to build sequential instance EPCs and can acquire domain knowledge by experts in order to enrich an instance EPC with parallelisms.

Another example of knowledge-based approach is (Greco, Guzzo, Manco, & Saccà, 2007). Here authors assume to have at disposal an a-priori model of the process, like those used in WMS. They derive for each trace of the log the corresponding process instance by *replaying* the trace on the model, i.e., by identifying the *enacted* subgraph in the overall model, that is the one whose nodes correspond to the activities of the trace. The work also introduces the idea to exploit a model mined from event logs by PD techniques, as in our proposal. However, they do not delve into the issues related with building instances from mined models, assuming a correct model can be generated, which is not the case for highly variable processes. As a further difference, we start from causal relations, instead of using replay. The two approaches are equivalent in the case all traces fully fit the model, which, however, is usually not the case when a filtering PD technique is adopted.

Approaches of group iii) address the most general case, i.e., the building of instance models from a simple event log, storing only basic information for each activity, i.e., its timestamp, its name and the case it belongs to, without exploiting additional domain knowledge. The approach proposed in this work belongs to this group. To the best of our knowledge, the only approaches developed to deal with this issue are (van Dongen & van der Aalst, 2004) and (van Beest et al., 2015), which propose to apply the α-algorithm to derive, respectively, the set of causal relations and the set of parallelisms existing among activities of the process. (van Dongen & van der Aalst, 2004) uses causal relations to determine which activities have to be linked in an IG, generating the graph in such a way that each activity is linked to its closest causal predecessors and to its closest causal successors. Therefore, an edge is inserted between two activities only if a causal relation exists among them. (van Beest et al., 2015) adopts the opposite idea. At the beginning, each activity is linked to all its successors; then, edges between parallel activities, as well as edges linking activities between which another path exists, are removed. Hence, an edge is inserted between two activities only when no explicit parallelism exists among them.

It is noteworthy that both approaches present severe limitations when addressing highly variable processes. Both of them exploit the α-algorithm to derive the causal relations, that is known to have issues in dealing with event logs of highly variable processes. Indeed, in these cases the α-algorithm usually returns imprecise, overgeneralizing models, i.e., models allowing for too much behaviors, from which one obtains, in turn, overgeneralizing IG, which do not provide any aid to the analysis of corresponding process instances.

[1] http://www.softwareag.com/nl/products/aris_platform/aris_controlling/aris_process_performance/overview/default.asp

We use as starting point for the development of this work the approach proposed by (van Dongen & van der Aalst, 2004) in that we exploit causal relations for building IG, adopting a filtering PD technique to infer causal relations instead of α rules. We point out, anyway, that also the exploitation of the parallelisms could be taken into account.

2.2. Model enhancement

The aim of Model Enhancement techniques consists in exploiting information recorded in an event log to improve a process model (van der Aalst, 2011). Among the possible kinds of enhancement, our work is especially related to the the *repairing* of a model. Given a process model and an event log which does not fit the model, the goal of model repairing techniques consists in modifying the original model to better represent the behaviors stored in the event log. An example of model repairing is proposed in (Buijs, La Rosa, Reijers, van Dongen, & van der Aalst, 2013). Here, authors exploit a genetic PD technique (i.e., the ETM algorithm) to infer a process model from the event log. In particular authors propose an extension of the ETM algorithm to take into account the structural similarity between the discovered process model and the reference one, to obtain a model that represents the deviating behaviors remaining as close as possible to the reference model.

A different approach is proposed in (Fahland & van der Aalst, 2015). Here, authors exploit a conformance checking technique to identify possible deviations between the event log and the process model, represented by means of a Petri net. If irregularities exist, they add one or more fragments representing the irregular behaviors to the original net, thus obtaining a new Petri net. More precisely, for each irregular behavior authors detect the corresponding "location", namely the set of places that are marked when the irregularity occurs. Then, they group sequences of irregular activities which are related to the same location. Finally, a new Petri net is inferred from each group that is then integrated into the original net. Our repairing procedure presents some similarities with the idea proposed in (Fahland & van der Aalst, 2015), since we also exploit conformance checking to detect irregularities and repair the model. However, our approach is tailored to analyze and repair directly single instance models. To the best of our knowledge, the approach proposed in this work is the first attempt aimed to deal with instance model enhancement.

3. Preliminaries

This section introduces important concepts used throughout the paper, recalling common definitions of PM literature (e.g., van der Aalst, 2011; 2013). In particular, the first two subsections are devoted to recall basic concepts, e.g., some basic definitions and concepts like *event, trace, Petri net* and so on, while the remaining subsections describe the main concepts related to *conformance checking, model evaluation measures* and *instance graphs*, respectively.

3.1. Basic concepts

The concepts we introduce in this subsection will be used in the definitions provided in the rest of the article.

Given a set A, $\mathcal{B}(A)$ is the set of all *multisets* over A. Some examples: $b_1 = []$, $b_2 = [x, x, y]$, $b_3 = [x, y, z]$, $b_4 = [x, x, y, x, y, z]$, $b_5 = [x^3, y^2, z]$ are multisets over $A = \{x, y, z\}$. b_1 is the empty multiset, b_2 and b_3 both consist of three elements, and $b_4 = b_5$, i.e., the ordering of elements is irrelevant and a more compact notation may be used for repeating elements. Multisets are used to represent the state of a Petri net and to describe event logs where the same trace may appear multiple times. The standard set operators can be extended to multisets, e.g., $x \in b_2$, $b_2 \uplus b_3 = b_4$, $b_5 \setminus b_2 = b_3$, $|b_5| = 6$, etc.

A *relation* $R \subseteq X \times Y$ is a set of pairs. $\pi_1(R) = \{x \mid (x, y) \in R\}$ is the domain of R, $\pi_2(R) = \{y \mid (x, y) \in R\}$ is the range of R, and $\omega(R) = \pi_1(R) \cup \pi_2(R)$ are the elements of R. For example, $\omega(\{(a, b), (b, c)\}) = \{a, b, c\}$. A relation (or function) is *bijective* if there is a one-to-one correspondence between the elements of the domain and range, i.e., it is total, functional, surjective, and injective.

$f : X \nrightarrow Y$ is a *partial function* with domain $dom(f) \subseteq X$ and range $rng(f) = \{f(x) \mid x \in X\} \subseteq Y$. f: $X \rightarrow Y$ is a *total function*, i.e., $dom(f) = X$. A partial function $f : X \nrightarrow Y$ is injective if $f(x_1) = f(x_2)$ implies $x_1 = x_2$ for all $x_1, x_2 \in dom(f)$.

$\sigma = \langle a_1, a_2, \ldots, a_n\rangle \in X^*$ denotes a *sequence* over X of length n. $\langle\ \rangle$ is the empty sequence. Sequences are used to represent paths in a graph and traces in an event log. $\sigma_1 \cdot \sigma_1$ is the concatenation of two sequences.

Functions can also be applied to sequences, such that: if $dom(f) = \{x, y\}$, then $f(\langle y, z, y\rangle) = \langle f(y), f(y)\rangle$.

Definition 1. (Applying functions to sequences)

Let $f : X \nrightarrow Y$ be a partial function. f can be applied to sequences of X using the following recursive definition (1) $f(\langle\rangle) = \langle\rangle$ and (2) for $\sigma \in X^*$ and $x \in X$:

$$f(\langle x\rangle \cdot \sigma) = \begin{cases} f(\sigma) & \text{if } x \notin dom(f) \\ \langle f(x)\rangle \cdot f(\sigma) & \text{if } x \in dom(f) \end{cases}$$

3.2. Events, traces, Petri nets

In this subsection, we introduce definitions related to event logs and their constituents, as well as to Petri nets. In the remainder, we assume the following universes of names, identifiers, and values:

- $\mathcal{C}$ is the set of all possible process instance identifiers, where a *process instance* corresponds to a specific execution of a process;
- $\mathcal{A}$ is the set of all possible activity names, where an *activity* is a well-defined task that has to be performed within the process;
- $\mathcal{E}$ is the set of all possible event identifiers, where an *event* is an instance of an activity.

An *event log* is defined as follows:

Definition 2. (Event log)

$L = (E, C, act, pi, \preceq)$ is an *event log* iff:

- $E \subseteq \mathcal{E}$ is a set of events identifiers,
- $C \subseteq \mathcal{C}$ is a set of identifiers of process instances (or cases),
- $act : E \rightarrow \mathcal{A}$ maps events onto activities,
- pi: $E \rightarrow C$ maps events onto process instances, and
- $\preceq \subseteq E \times E$ defines a total order on events ($\preceq$ is reflexive, antisymmetric, transitive, and total).[2]

The goal of the PD is to elaborate upon a given event log to derive a *process model* which describes the ordering relations existing between activities of the process. Process instances can be mapped onto traces of a process model using the function *trace*.

Definition 3. (Trace)

Let $L = (E, C, act, pi, \preceq)$ be an event log. $trace : C \rightarrow \mathcal{A}^*$ maps process instances onto traces such that for any process instance $c \in C$:

- $trace(c) = \langle a_1, a_2, \ldots, a_n\rangle \in \mathcal{A}^*$ is a sequence of activities, each corresponding to an event,

[2] $e_1 \prec e_2$ if and only if $e_1 \preceq e_2$ and $e_1 \neq e_2$.

- there exists a bijection $f_c \in \{1, 2, \ldots, n\} \to \{e \in E \mid pi(e) = c\}$ such that
  - $act(f_c(i)) = a_i$ for any $i \in \{1, 2, \ldots, n\}$, and
  - $f_c(i) \preceq f_c(j)$ for any $i, j \in \{1, 2, \ldots, n\}$ with $i \leq j$.

Note that there is a one-to-one correspondence between the events of a process instance and positions on the trace. In the context of an event log and a process instance, f_c provides the mapping. Hereafter, given an event log L, we will write that σ is a trace of L if there exists a $c \in C$ such that $\sigma = trace(c)$.

Several modeling formalisms exist to represent process models; in this work, we refer to the *Petri net*, defined as follows:

Definition 4. (Petri net)
A Petri net is a tuple $N = (P, T, F)$ with P the set of places, T the set of transitions, $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ the flow relation.

Places may contain a discrete number of marks called *tokens*; the distribution of tokens is a configuration of the Petri net, which is called a *marking*.

Definition 5. (Marking)
Let $N = (P, T, F)$ be a Petri net. A marking M is a multiset of places, i.e., $M \in \mathcal{B}(P)$.

A Petri net defines a bipartite graph, with nodes $P \cup T$ and egdes F. For every $x \in P \cup T$, $\overset{N}{\bullet} x = \{y \mid (y, x) \in F\}$ is the set of *input nodes* while, similarly, $x \overset{N}{\bullet} = \{y \mid (x, y) \in F\}$ denotes the set of *output nodes*; we drop the superscript N if it is clear from the context. A transition $t \in T$ is said *enabled* only if there is at least one token for each of its input nodes. An enabled transition t may *fire*, i.e., one token is removed from each of the input places $\bullet t$ and one token is produced for each of the output places $t\bullet$. Formally: $M' = (M \setminus \bullet t) \uplus t\bullet$ is the marking resulting from firing enabled transition t in marking M of Petri net N. $(N, M)[t\rangle(N, M')$ denotes that t is enabled in M and firing t results in marking M'. Let $\sigma = \langle t_1, t_2, \ldots, t_n \rangle \in T^*$ be a sequence of transitions. $(N, M)[\sigma\rangle(N, M')$ denotes that there is a set of markings $M_0, M_1, \ldots, M_n$ such that $M_0 = M$, $M_n = M'$, and $(N, M_i)[t_{i+1}\rangle(N, M_{i+1})$ for $0 \leq i < n$. A marking M' is *reachable* from M if there exists a σ such that $(N, M)[\sigma\rangle(N, M')$.

Definition 6. (Firing sequence)
Let (N, M) with $N = (P, T, F)$ be a marked Petri net. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, M) if and only if, for some natural number $n \in \mathbb{N}$, there exist markings M_1, ..., M_n and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = \langle t_1, \ldots, t_n \rangle$ and $\forall i : 1..n((N, M_i)[t_{i+1}\rangle(N, M_{i+1}))$.

For process modeling, typically *labeled* Petri net are used, defined as:

Definition 7. (Labeled Petri net)
A labeled Petri net $N = (P, T, F, l)$ is a Petri net (P, T, F) with labeling function $l : T \nrightarrow \mathcal{A}$. Let $\sigma_v = \langle a_1, a_2, \ldots, a_n \rangle \in \mathcal{A}^*$ be a sequence of activities. $(N, M)[\sigma_v \rhd (N, M')$ if and only if there is a sequence $\sigma \in T^*$ such that $(N, M)[\sigma\rangle(N, M')$ and $l(\sigma) = \sigma_v$ (cf. Definition 1).

We use the τ label for a *silent transition*, namely a transition $t \notin dom(l)$. An example of labeled Petri net is shown in Fig. 2, which will be used as running example throughout the paper. For such a net, $P = \{Start, p_1, \ldots, p_9, End\}$, $T = \{t_1, t_2, \ldots, t_{13}\}$ and $F = \{(Start, t_1), (t_1, p_1), (p_1, t_2), \ldots, (t_{13}, End)\}$. Each transition is labeled, e.g., $l(t_1)$=a, $l(t_3)$=b and so on. Note that t_2 is a silent transition.

For a given process, typically a specific initial state (i.e., the initial configuration of the net) and a specific final state (i.e., the final configuration of the net) exist.

This can be modeled by referring to the concept of *system net*.

Definition 8. (System net)
A system net is a triplet $SN = (N, M_{init}, M_{final})$ where $N = (P, T, F, l)$ is a labeled Petri net, $M_{init} \in \mathcal{B}(P)$ is the initial marking, and $M_{final} \in \mathcal{B}(P)$ is the final marking. $\mathcal{U}_{SN}$ is the *universe of system nets*.

For the labeled Petri net in Fig. 2, $M_{init} = [Start]$ and $M_{final} = [End]$. Each firing sequence which starts in M_{init} and end in M_{final} is a *complete* firing sequence.

Given a system net, $\phi(SN)$ is the set of all possible *visible* traces which can be generated from the net, i.e., complete firing sequences projected onto the set of observable activities.

Definition 9. (Traces of a system net)
Let $SN = (N, M_{init}, M_{final}) \in \mathcal{U}_{SN}$ be a system net. $\phi(SN) = \{\sigma_v \mid (N, M_{init})[\sigma_v \rhd (N, M_{final})\}$ is the set of *visible* traces starting in M_{init} and ending in M_{final}. $\phi_f(SN) = \{\sigma \mid (N, M_{init})[\sigma\rangle(N, M_{final})\}$ is the corresponding set of complete firing sequences.

3.3. Conformance checking

Given an event log L and a Petri net N, conformance checking techniques are aimed at checking whether or not N *fits* the log. N *perfectly fits* L iff each trace can be replayed on the model from the begin to the end.

Definition 10. (Perfectly fitting log)
Let $L = (E, C, act, pi, \preceq)$ be an event log and let $SN = (N, M_{init}, M_{final}) \in \mathcal{U}_{SN}$. SN is perfectly fitting L if and only if $\forall c \in C(trace(c) \in \phi(SN))$.

One of the most common measures used for conformance checking is the *fitness*, which can be measured by *aligning* the traces of the event log with the firing sequences of the net. An *alignment*, here indicated with the symbol γ, shows a possible correspondence between a trace of an event log and a firing sequence of a Petri net. For example, let us consider the trace $\sigma = \langle a, b, d, e, f, b, g \rangle$. A possible *alignment* between σ and a firing sequence of the Petri net in Fig. 2 is:

$$\gamma = \begin{array}{c|c|c|c|c|c|c|c|c} a & \gg & b & \gg & d & e & f & b & g \\ \hline a & \tau & b & c & d & e & f & \gg & g \\ t_1 & t_2 & t_3 & t_6 & t_7 & t_8 & t_9 & & t_{13} \end{array}$$

The first row refers to the event log, while the other two refer to the model. In particular, the sequence of activities in the trace is shown in the first row. The second row shows activities corresponding to transitions of the firing sequence, which are shown in the last row. We need two rows for the model since multiple transactions corresponding to the same activity could exist.

When the Petri net perfectly fits with the log, each activity in the first row occurs also in the second row, in the same position; otherwise, a “no-move” symbol $\gg$ is inserted. For example, in the fourth column we have a no-move in the trace, since according to the firing sequence of the Petri net the transition t_6 should have been fired, but the corresponding activity c is not displayed in the trace. Every pair $(x, (y, t))$, such that the first element belongs to the first line (i.e., it refers to the log) and the second one involves an element from the second and the third line (i.e., the label and the id of a transition of the net, respectively) is called “move”. For example, $(a, (a, t_1))$ states that both the log and the model make an “a” move. Note that the move in the model is caused by the occurrence of the transition t_1, whose label is a. There can be three possible kinds of moves: a) a *synchronous move* if x is an activity in the trace and t is a transition in the model, b) a “move-in-the-log”, if t is $\gg$ and c) a “move-in-the-model”, if x is $\gg$. Note that

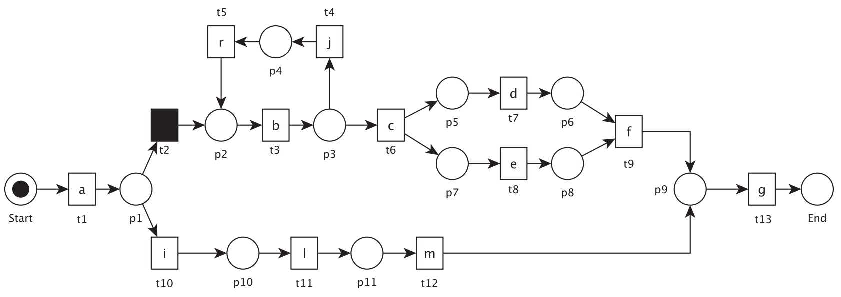


Fig. 2. A labeled Petri net.

a move-in-the-log means that an activity has occurred in a not allowed position, while a move-in-the-model means that a certain activity should have been occurred according to the model and it did not occur in the reality. Such cases are usually indicated as *inserted* and *deleted* activities, respectively. Moves a, b and c are called "legal moves".

Definition 11. (Legal moves)
Let L be an event log and let $SN = (N, M_{init}, M_{final})$ be a system net, with $N = (P, T, F, l)$. $\mathcal{A}_{\mathcal{LM}} = \{(x, (x, t)) \mid x \in (\mathcal{A}) \wedge t \in T \wedge l(t) = x\} \cup \{(\gg, (x, t)) \mid t \in T \wedge l(t) = x\} \cup \{(x, \gg) \mid x \in (\mathcal{A})\}$ is the set of *legal moves*.

Formally, we define an alignment as follows.

Definition 12. (Alignment)
Let σ_L be a trace of an event log L, let $\sigma_{SN} \in \phi_f(SN)$ be a complete firing sequence of the system net SN, and let $\mathcal{A}^*_{\mathcal{LM}}$ be the set of all sequences of legal moves. An *alignment* of σ_L and σ_{SN} is a sequence $\gamma \in \mathcal{A}^*_{\mathcal{LM}}$ such that the projection on the first element of each pair (ignoring $\gg$) yields σ_L and the projection on the last element (ignoring $\gg$) yields σ_{SN}.

It is noteworthy that for a given trace many possible alignments typically exist (possibly, infinitely many). Nevertheless, it is possible to evaluate the quality of alignments, by introducing a cost function:

Definition 13. (Cost of an alignment)
Let $\delta : \mathcal{A}_{\mathcal{LM}} \rightarrow \mathbb{N}$ be a cost function, which assigns each legal move to a specific cost. The *cost of an alignment* $\gamma \in \mathcal{A}^*_{\mathcal{LM}}$ is the sum of all costs: $\delta(\gamma) = \sum_{(x,y) \in \gamma} \delta(x, y)$.

The cost function can be defined in several ways; anyway, in this work we refer to the standard cost function. This function assigns a zero cost to synchronous moves and to moves-in-the-model corresponding to silent transitions, while the cost for each move-in-the-log and each move-in-the-model corresponding to a non silent transition are both equal to one. Hence, in our example we have $\sigma(a, (a, t_1)) = 0$, since the move-in-the-log is mimicked by the move-in-the-model, and $\sigma(\gg, (\tau, t_2)) = 0$, being t_2 is a silent transition. Instead, $\sigma(\gg, (c, t_6)) = 1$, since activity c (caused by the occurrence of the transition t_6) is not mimicked in the event log. Given a trace σ, the alignment with the lowest cost (i.e., the one which allows the most synchronous moves) is called *optimal alignment*. It is noteworthy that for a given trace more than one optimal alignments can exist. To obtain for each trace in the log a unique optimal alignment, a mapping function $\kappa : \mathcal{A}^* \rightarrow \mathcal{A}_{\mathcal{LM}}$ is defined, which deterministically assigns each trace σ to an alignment γ such that γ is optimal. This function can be seen as an "oracle", which provides us with a unique complete firing sequence of the model for each trace in the log (Adriansyah, Munoz-Gama, Carmona, van Dongen, & van der Aalst, 2013). Hereafter, we indicate the optimal alignment provided by κ for a given trace σ as γ^*_σ. Several criteria can be used to define κ; however, the definition of the mapping function is out of the scope of this paper. Here, for computing the alignments we refer to the technique proposed in (Adriansyah, van Dongen, & van der Aalst, 2011), which introduces a cost-based replay technique based on the A^* algorithm.

In order to introduce our technique in Section 5 we will make use of the following definitions that formally specify the notions of irregularity using the notion of alignment cost.

Definition 14. (Irregular traces)
Let $SN = (N, M_{in}, M_{fin})$ be a system net, and let σ be a trace of the event log L. If $\delta(\gamma^*_\sigma) > 0$, then σ is an irregular trace.

Since the cost of the best alignment is greater than zero, an irregular trace involves one or more inserted/deleted activities.

Definition 15. (Inserted activity)
Let $\sigma = trace(c) = \langle a_1, \ldots, a_i, \ldots, a_n \rangle$ be an irregular trace with respect to the system nset SN, with $\delta(\gamma^*_\sigma) = h,\ h \geq 1$. The activity a_i is an *inserted activity* in the ith position of σ iff a trace $\sigma' = \langle a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \rangle$ exists such that $\delta(\gamma^*_{\sigma'}) = h - 1$.

Definition 16. (Deleted activity)
Let $\sigma = trace(c) = \langle a_1, \ldots, a_n \rangle$ be an irregular trace with respect to the system net SN, with $\delta(\gamma^*_\sigma) = h,\ h \geq 1$. The activity $a' \in A$ is a *deleted activity* in the ith position of σ iff a trace $\sigma' = \langle a_1, \ldots, a_{i-1}, a', a_i, \ldots, a_n \rangle$ exists such that $\delta(\gamma^*_{\sigma'}) = h - 1$.

In other words, given a trace which does not fit with a model, an activity is considered an insertion (deletion) if by removing (adding) such activity, a new trace is obtained whose cost of alignment is decreased by one. In general, when a set of inserted (deleted) activities are detected in the same position, conformance checking techniques return a sequence of inserted (deleted) activities.

3.4. Model evaluation measures

Common measures exploited to evaluate a process model with respect to an event log are *fitness, simplicity, generalization*, and *precision*. Fitness, as mentioned above, measures the extent to which the model is able to describe behaviors stored in the event log. Simplicity is used to evaluate to some extent the readability of a process model, and can be measured in several ways (e.g., by counting the number of nodes and edges in the model). The assumption underlying this metric is that the best model is the simplest one which is able to properly represent the behavior in the event log. As regards generalization and precision, both are based on comparing the behaviors allowed by the process model, with the behaviors actually stored in the event log. Generalization regards the capability of the model to generate traces which differ from the ones in the event log, i.e, its capability to fit with future process instances. Indeed, a model which can represent only the behaviors stored in the event log is an *overfitting* model, i.e., it can only describes past process executions, while a good model should be able to represent all process executions, also those which did not occur yet. At the same time, a good process model has also to be precise, namely it does not have to allow for "too many" not observed behavior. Otherwise, the model allows for any order of occurrence of activities in the process, thus resulting useless in analyzing the process. In this case, the model *underfits*, or *overgeneralizes* (De Medeiros et al., 2008) the log. Note that, it is not easy to define which are good values for precision and generalization for a generic process; intuitively, one should look for models with a good trade-off between these metrics. Clearly, the concrete value for such a trade-off is usually not known a-priori, and depends on the context.

3.5. Instance graphs

Although a process usually involves some parallel activities, events that occur are recorded in a trace as a sequence, thus hiding possible parallelism. Hence, in order to identify parallel events within a trace, one needs to know which are the dependencies existing between the activities of the process. To face with such issue, (van Dongen & van der Aalst, 2004), (van Dongen & van der Aalst, 2005) introduce the concept of *Instance Graph* (IG), which is briefly recalled in the following.

Definition 17. (Causal Relation)
A *Causal Relation* (*CR*) is a relation on the set of activities, i.e., $CR \subseteq \mathcal{A} \times \mathcal{A}$. $a_1 \rightarrow_{CR} a_2$ denotes that $(a_1, a_2) \in CR$.

Each element of *CR* represents the order of execution of a pair of activities of a process. For example, $a_1 \rightarrow_{CR} a_2$ states that a_2 cannot be executed until a_1 is terminated; in other words, the execution of a_2 *depends* on the execution of a_1. a_1 is defined as the *causal predecessor* of a_2, and a_2 is the *causal successor* of a_1. Note that the above definition of CR involves also self-loops; indeed, one can find $a_1 \rightarrow_{CR} a_1$.

Given a trace σ, we can build its *Instance Graph* (IG), defined as follows:

Definition 18. (Instance graph)
Let $L = (E, C, act, pi, \preceq)$ be an event log and let $c \in C$ be a process instance. The instance graph of *c* is $IG_c = (V, W, \eta)$ with

- $V = \{e \in E \mid pi(e) = c\}$ is the set of nodes corresponding to events of *c*,
- $W = \{(e_1, e_2) \in V \times V \mid e_1 \preceq e_2 \wedge act(e_1) \rightarrow_{CR} act(e_2) \quad \wedge \quad (\forall e' \in V \quad (e_1 \prec e' \prec e_2 \Rightarrow act(e_1) \not\rightarrow_{CR} act(e')) \quad \vee \quad \forall e'' \in V \quad (e_1 \prec e'' \prec e_2 \Rightarrow act(e'') \not\rightarrow_{CR} act(e_2)))\}$ orders the events,
- $\eta : V \rightarrow \mathcal{A}$ with $\eta(e) = act(e)$ for $e \in V$, labels a node with the corresponding activity.

  $e_1 \rightarrow_W e_2$ denotes that $(e_1, e_2) \in W$.

IG_c is a graph representing the flow of the activities for the process instance *c*. The nodes set *V* contains a node for each event in $\sigma = trace(c)$; each node is labeled, by means of η, with the corresponding activity. *W* is a set of directed edges between nodes v_i, v_j.

Two nodes are connected in IG_c by means of a *path*, defined as:

Definition 19. (Path)
Let $IG_c = (V, W, \eta)$ be an instance graph. Given two nodes e', $e'' \in V$, a *path* from e' to $e''V$ is defined as a sequence $\langle e_1, e_2, ...e_k\rangle \in V^*$ with $k \geq 2$ such that $e_1 = e'$ and $e_k = e''$ and $\forall 1 < i \leq k\ ((e_{i-1}, e_i) \in W)$

We would like to note that, from the Definition 18, if $(e', e'') \in W$, then $\langle e', e''\rangle$ is the only path between e' and e''.

Example 1. Let us consider the trace $\sigma_1 = \langle a, b, c, d, e, f, g\rangle$ of the process instance *c*, for which we assume $CR = \{(a, b), (b, c), (c, d),$ $(c, e), (d, f), (d, g), (e, f), (f, g)\}$. To build the corresponding IG_c, first a node is created for each event; then, each pair of events in the trace is analyzed. The first pair is characterized by the activities *a* and *b*; since $a \rightarrow_{CR} b$ and there are no other events between them in the trace (i.e., *b* is a direct follower of *a*), the corresponding nodes are linked by means of an edge in the graph. All other pairs starting from the first event, (i.e., $(a, c), (a, d), \ldots,$ and (a, g)) are not part of *CR*, so we move to the second event. For this event, we have only $b \rightarrow_{CR} c$, and since *c* is a direct follower of *b*, the corresponding nodes are linked. The third event is the causal predecessor of two events, characterized by activities *d* and *e* respectively. The former is a direct follower of *c*, and hence the edge is drawn. The latter does not directly follow *c*, but we have *d* in the trace between *c* and *e*. Since $c \rightarrow_{CR} d$ but $d \not\rightarrow_{CR} e$, these nodes are also linked. The procedure is repeated until all events have been considered. Fig. 3 shows the instance graph corresponding to σ_1. It should be noted that the set of edges *W* of an IG represents a subset of *CR*. The constraints defined for *W* are aimed to avoid the presence of redundant edges in the graph, namely edges linking nodes that are already linked by a path. In this example, the relation $d \rightarrow_{CR} g$ does not produce any edge, since the path $\langle d, f, g\rangle$ already exists.

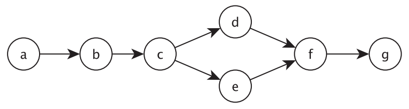


Fig. 3. Instance graph IG_1 for the trace σ_1.

Definition 18 allows to represent only split/join *AND* constructs. Hence, when an IG has parallel branches, it means that in the corresponding process instance, events of the branches have been executed in parallel. In the case a process has split/join *OR* constructs, in the process instance (and hence, in the trace) only the branches actually executed will be reported. Furthermore, IGs are always acyclic; in fact, if a loop exists in the process, in the trace each execution of the loop is represented by a sequence of different events, which correspond to different nodes in the IG.

From the above definitions, it turns out that a process instance *c* generates the trace $\sigma = trace(c)$ and the instance graph IG_c; however, the same Instance Graph could be obtained also from other traces in the event log. Sequences of activities that could generate the same instance graph are called occurrence sequences, as follows:

Definition 20. (Occurrence sequence)
Let $IG_c = (V, W, \eta)$ be an instance graph. The sequence $\sigma = \langle a_1, a_2, \ldots, a_n\rangle$ is an *occurrence sequence* of IG_c if there exists a bijection $g \in \{1, 2, \ldots, n\} \rightarrow V$ such that:

- $\eta(g(i)) = a_i$ for any $i \in \{1, 2, \ldots, n\}$,
- $\nexists e \in V | (e, g(1)) \in W$,
- $\nexists e \in V | (g(n), e) \in W$,
- $g(i) \rightarrow_W g(j)$ implies that $i < j\ \forall i, j \in \{1, 2, \ldots, n\}$.

In other words, in an occurrence sequence of an instance graph the first and the last events correspond to nodes of the graph for which no input and no output edges are defined.

Example 2. Let us consider again the IG shown in Fig. 3. Since it involves a parallelism, two different occurrence sequences can be generated, i.e., $\sigma_1 = \langle a, b, c, d, e, f, g\rangle$ and $\sigma_2 = \langle a, b, c, e, d, f, g\rangle$. It is interesting to note that only one of these occurrence sequences (i.e., σ_1) actually corresponds to the trace from which the graph has been built.

By analyzing the occurrence sequences, it is possible to apply the quality measures regarding the precision and the

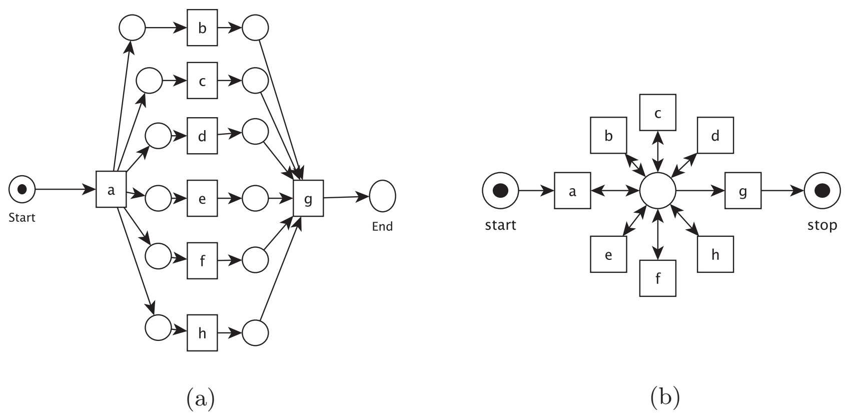


Fig. 4. Examples of overgeneralizing models.

generalization of a process model introduced in the previous subsection also to an IG. We will discuss their use in Section 6, when describing the experiments.

4. Causal relation discovery for highly variable processes

The aim of this section is to provide an in depth analysis of the issues encountered by existing techniques in inferring causal relations and hence IG for highly variable domains. We hasten to note that the problem of inferring causal relations is strictly related to the problem addressed by PD. As a matter of fact, *CR* represents the basic information PD techniques elaborate upon to generate the model, and every PD technique has its own *CR* inference rules. Therefore, to simplify the reading, hereafter, we refer to PD approaches to indicate the corresponding rules to extract the *CR*.

A relevant aspect of a PD technique is its filtering capability. Especially in highly variable contexts, very different results can be obtained when choosing either a *non-filtering* approach, which tries to represent all the behaviors stored in the event log, or a *filtering* approach, which considers only more frequent behaviors. We address both cases in the following subsections, showing the effects on building IGs.

4.1. Non filtering approaches

Non-filtering PD techniques try to build a model able to represent all the behaviors stored in the event log. When applied to event logs involving heterogenous behaviors, non-filtering techniques tend to build models which *overgeneralizes* the process (De Medeiros et al., 2008), i.e., which allow for practically every execution order among activities. Fig. 4 shows two examples of overgeneralizing models.

Fig 4 a shows a model where almost all of the activities, except a and g, are represented in parallel, while Fig. 4b represents a dual situation, where all activities are linked among each other. A well-known approach which can easily lead to model represented in Fig. 4a when applied to highly variable processes is the α-algorithm, which is the technique used in (van Dongen & van der Aalst, 2004) and in (van Beest et al., 2015). It recognizes a causal relation between two activities a and b only if b directly follows a in at least one trace and the opposite never occurs; otherwise, they are considered parallel[3]. Hence, it is apparent that also with just one exception in the log, two activities will be modeled as parallel. The kind of model depicted in Fig. 4b, known in literature as a “Flower Model”, can be obtained by applying the *Heuristic Miner* (HM) (Weijters et al., 2006) in a non-filtering configuration. The HM can be viewed as an extension of the α-algorithm. It redefines and extends the original rules of the α-algorithm, in order to take into account the frequency of follows relations, applying some heuristics to determine causality and parallelisms. In particular, given two activities a and b, HM returns in output $a \rightarrow_{CR} b$ if the difference between the number of times b follows a and the number of times a follows b is above an user-defined threshold; otherwise it is filtered out. Similarly for $b \rightarrow_{CR} a$. When very low thresholds are used for filtering (or no filtering is exploited at all), then both the relations are displayed in the outcome; for highly variable processes, this easily leads to a model where all the activities are connected to each other.

IGs generated by using the *CR* corresponding to models in Fig. 4 cannot properly represent process instances. Let us consider, for example, the trace $\sigma = \langle a, b, c, d, e, f, h, g \rangle$ corresponding to a process instance k. The IG_k built by using the *CR* of the model in Fig. 4a results identical to the model, i.e., all the activities except a and g will be in parallel. Hence, several occurrence sequences of IG_c are allowed and the instance graph overgeneralizes, providing a very poor representation of k. Instead, from the *CR* of the model in Fig. 4b, we can obtain only sequential IGs. Indeed, since each event is linked to its direct follower in the trace, IG_k will be a sequence which mimics the order of the events in the trace; such a graph is clearly a trivial model for k.

4.2. Filtering approaches

Filtering techniques use only the most relevant dependencies to build the model, where a dependency is usually relevant if it occurs more frequently than a given threshold. Two well-known examples of filtering PD algorithms are the already mentioned Heuristic Miner (HM) and the infrequent Inductive Miner (iIM). The iIM algorithm adopts an iterative procedure. At the beginning, it generates a directed graph representing the direct follows relations between activities of the event log. Then, it defines a “cut” of the graph, i.e., a partition between the nodes in disjoint sets such that all the activities in a subset have the same relation with activities in the other subsets. Relations like sequences, loops, parallelism are taken into account. The log is then split to reflect the partitioning of the activities and the procedure is then repeated on each sublog, until sublogs contain only one activity. Similarly to HM, iIM introduces a set of heuristics aimed at filtering infrequent behaviors.

By considering the most common process behavior, filtering approaches usually cut down overgeneralization, at the expenses of the completeness, since some of the traces will not fit the model.

[3] In (van Dongen & van der Aalst, 2004) the improved version of the α-algorithm is used, which takes into account also possible two-length loops when determining causal relations.

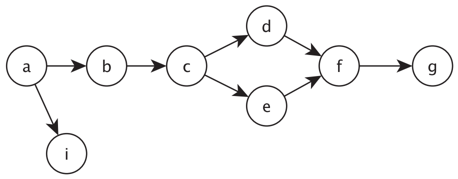


Fig. 5. IG for the trace σ_{in_1}.

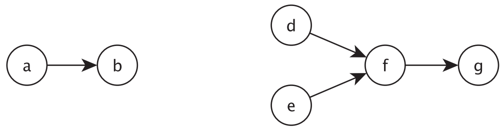


Fig. 6. IG for the trace σ_{del_1}.

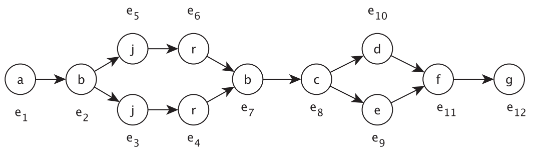


Fig. 7. IG of the trace σ_{del_2}.

According to Definition 14 they are called irregular traces. When applied to an irregular trace Definition 18 leads to the generation of an IG (hereafter irregular Instance Graph, or iIG) providing a very imprecise representation of the corresponding process instance, as shown in the following examples.

Example 3. Let us consider the trace $\sigma_{in_1} = trace(in_1) = \langle a, b, c, \mathbf{i}, d, e, f, g\rangle$. According to the model of Fig. 2, there is an inserted activity i at the 4th position. Fig. 5 shows the IG built from σ_{in_1}, i.e., IG_{in_1}. The node i has only one input edge, and no output edges. In such a way, the only execution constraint for i is to be performed after a; hence, IG_{in_1} can generate several occurrence sequences, where the activity i occurs in very different positions, e.g., $\sigma_1 = \langle a, i, b, c, d, e, f, g\rangle$, $\sigma_2 = \langle a, b, c, d, e, f, g, i\rangle$, and so on.

Example 4. Let us consider the trace $\sigma_{del_1} = \langle a, b, d, e, f, g\rangle$. According to the model of Fig. 2, there is a deleted activity c at the 3rd position. Fig. 6 shows the IG built from such trace, i.e., IG_{del_1}, which consists of two disconnected graphs. This representation is clearly too imprecise; the two components are not linked, thus implying that no ordering relations exist between them. Hence, the graph can generate occurrence sequences like $\sigma_1 = \langle a, d, e, f, g, b\rangle$, $\sigma_2 = \langle d, e, f, g, a, b\rangle$ and so on.

From the previous examples, it turns out that an iIG may provide an overgeneralized representation of the corresponding process instance, which is clearly not desired. Examples 3 and 4 show also that the overgeneralization can be, in many cases, detected also by means of some structural anomalies which affect the graph, e.g., disconnected nodes, presence of more than one starting/ending nodes. It is also possible, however, to obtain graphs corrected from a structural point of view, but which anyway allow for not desired occurrence sequences, as shown by the following example.

Example 5. Let us consider the trace $\sigma_{del_2} = \langle a, b, j, r, j, r, b, c, d, e, f, g\rangle$, where each activity a_i in the ith position corresponds to an event $e_i \mid act(e_i) = a_i$. According to the model of Fig. 2, there is a deletion of the activity b in the 5th position. The resulting graph is shown in Fig. 7. Note that for this example we also report the event identifiers, in order to avoid possible misunderstandings due to the multiple occurrences of some activities. As we can see, this IG has no structural anomalies; however, there is a parallelism between the two occurrences of the activities j and r, which is due to the deletion of b but is not allowed by the model. Again, the instance graph overgeneralizes returning four occurrence sequences.

While in structured processes the number of irregular traces is not significant and these traces could be discarded, in higly variable processes irregular traces have to be properly handled. As a matter of fact, these kind of processes usually represent human-intensive processes, where irregular behaviors are intentional variations of regular behaviors, hence their analysis turns to be interesting. Hence, rather than removing iIGs, our goal is to *repair* them. In next section we detail the proposed algorithms for graph repairing that, starting from models generated by filtering techniques, transform an iIG in order to reduce overgeneralization.

5. Building instance graphs

Algorithm 1 provides an overview of the Building Instance Graphs (*BIG*) algorithm. Given an event log L, first the filtering technique PD is applied to obtain the process model M, with its corresponding CR (step 1). Then, for each trace $\sigma_k = trace(k), k \in C$ the corresponding instance graph IG_k is built by referring to Definition 18 (step 3). Step 4 checks whether σ_k is an irregular trace, by checking the conformance with the model. This check is done by evaluating the fitness of the trace to the model. In this work, we refer to the conformance checking approach proposed by (Adriansyah et al., 2011), which minimizes the standard cost function defined in Section 3. If a trace is irregular, then the lists of inserted (I) and deleted (D) activities are returned. Indeed, each element of D and I is a consecutive sequence of irregular activities. For instance, if at the ith position there is only one inserted activity and from jth to hth position there are $h - j + 1$ consecutive inserted activities, we have $I = \{\langle a_i\rangle, \langle a_j, \ldots, a_h\rangle\}$. Irregular traces are repaired in step 6, by using the repairing algorithm reported in Algorithm 2.

Given a trace, its instance graph IG_k and the lists of its inserted/deleted activities, Algorithm 2 returns a repaired graph IG'_k having same nodes and a few difference in the set of edges. Note that varying edges returns a graph that is not fully compliant with Definition 18; nevertheless, since the graph is still the flow of activities of the trace, for the sake of simplicity we call it IG as well.

Algorithm 1 Building Instance Graphs Algorithm.

```
Let $L = (E, C, act, pi, \preceq)$ be an event log
1: $[M, CR]$=ApplyProcessDiscovery($L$, $PD$);
2: for $k = 1$ to $|C|$ do
3:     $IG_k$=ExtractInstanceGraph($\sigma_k$,$CR$);
4:     $[D, I]$=CheckTraceConformance($\sigma_k$,$M$);
5:     if $D \cup I \neq \emptyset$ then
6:         $IG_k$=IrregularGraphRepairing($IG_k$,$\sigma_k$, $D$, $I$, $CR$);
```

Algorithm 2 The Irregular Graph Repairing Algorithm.

```
Let $IG_k = (V, W, \eta)$ be the irregular instance graph for the trace $\sigma_k$
Let $D$ be the set of deleted activities of $\sigma_k$
Let $I$ be the set of inserted activities of $\sigma_k$
Let $CR$ be the causal relation
1: $W' = W$
2: for all $a \in D$ do
3:     $W' = DR(W', a, CR)$
4: for all $a \in I$ do
5:     $W' = IR(W', a, CR, \sigma'_k)$
6: $IG' = (V, W', \eta)$
```

Algorithm 3 The Deletion Repairing Algorithm.

```
Let $\sigma_{del} = \langle a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n\rangle$ be an irregular trace
Let $\langle a_{d_1}, \ldots, a_{d_n}\rangle$ be the sequence of deleted activities in the $i$th position of $\sigma_{del}$
Let $IG_{\sigma_{del}} = (V, W, \eta)$ be the iIG for the trace $\sigma_{del}$, with $a_i = \eta(e_i)$
Let $CR$ be the causal relation
1: $W_{r_1} = \{(e_k, e_i) \mid k < i \wedge ((e_k, e_i) \in W) \wedge (\exists e_h, k \le h < i \mid \eta(e_h) \rightarrow_{CR} a_{d_1}) \wedge (a_{d_n} \rightarrow_{CR} \eta(e_i))\}$
2: $W_{r_2} = \{(e_k, e_j) \mid k < i \wedge j > i \wedge ((e_k, e_j) \in W) \wedge (\eta(e_k) \rightarrow_{CR} a_{d_1}) \wedge (a_{d_n} \rightarrow_{CR} \eta(e_i)) \wedge \exists e_l, i < l < j \mid (e_l, e_j) \in W\}$
3: $W' = W \setminus (W_{r_1} \cup W_{r_2})$
4: for $k = i - 1$ to 1 do
5:   for $j = i$ to $n$ do
6:     if $(\eta(e_k) \rightarrow_{CR} a_{d_1}) \wedge (a_{d_n} \rightarrow_{CR} \eta(e_j)) \wedge (\langle e_k, \ldots, e_j\rangle \notin W') \wedge ((\nexists e_l, k < l < j \mid (e_k, e_l) \in W') \vee (\nexists e_m, k < m < i \mid (e_m, e_j) \in W')))$ then
7:       $W' = W' \cup \{(e_k, e_j)\}$
```

The algorithm repairs first all the deleted activities, and then all the inserted activities, calling the *Deletion Repairing Algorithm* (DR) and the *Insertion Repairing Algorithm* (IR) respectively.

5.1. Deletion repairing algorithm

The idea underlying the DR algorithm consists in connecting activities occurred before and after the deleted activities, and in removing those edges which should not have been created according to the model. Algorithm 3 describes the general case when a sequence of deleted activities is detected in the same position of the trace. The algorithm starts by identifying the sets W_{r_1}, W_{r_2}, which contain edges generated by the deleted activities. Indeed, the deletion of one or more activities could lead to an irregular IG where some events are linked although they should not be linked according to the model (e.g., see Example 5). These edges have to be recognized and deleted. In particular, W_{r_1} contains edges between events occurring before the position of the deletion and the ith event. Edges in W_{r_2} link events before and after the ith position. Then, a set of edges to be added to the graph is computed (steps 4–7). These edges link those events occurring before and after the deleted sequence, such that a dependency exists between them and the first and the last events in the sequence respectively. Note that only direct edges are considered, namely edges between events which are no linked by other paths in the new graph. As output we obtain the transformed set W', which is needed to define the repaired graph.

Example 6. Let us consider the trace $\sigma_{del_1} = \langle a, b, d, e, f, g\rangle$ of the Example 4. We have only one deleted activity c in the 3rd position. At the beginning of the procedure, we have $W = \{(a, b), (d, f), (e, f), (f, g)\}$[4]. In this case, both the sets W_{r_1}, W_{r_2} are empty; while from steps 4–7 it results that the edges (b, d) and (b, e) have to be added. At the end, we obtain $W' = \{(a, b), (d, f), (e, f), (f, g), (b, d), (b, e)\}$; the resulting graph is shown in Fig. 8.

Example 7. Let us consider the trace $\sigma_{del_2} = \langle a, b, j, r, j, r, b, c, d, e, f, g\rangle$, defined in Example 5. The deleted activity is b in the fifth position. In the original instance graph we have $W = \{(e_1, e_2), (e_2, e_3), (e_2, e_5), (e_3, e_4), (e_5, e_6), (e_4, e_7), (e_6, e_7), (e_7, e_8), (e_8, e_9), (e_8, e_{10}), (e_9, e_{11}), (e_{10}, e_{11}), (e_{11}, e_{12})\}$. It is straightforward to verify that $W_{r_1} = \{(e_2, e_5)\}$ and $W_{r_2} = \{(e_4, e_7)\}$. Then, we derive

[4] We use activities instead events identifiers with an abuse of notation in case this creates no misunderstandings.

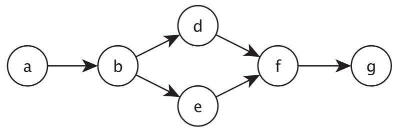


Fig. 8. Repaired graph for $\sigma_{del_1.}$

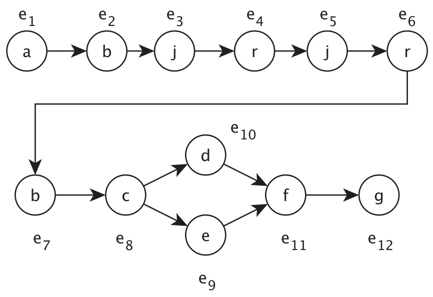


Fig. 9. Repaired graph for $\sigma_{del_2.}$

that the edge (e_4, e_5) has to be added to W'. The repaired graph is shown in Fig. 9.

5.2. The insertion repairing algorithm

Algorithm 4 describes the IR algorithm, which is aimed at restructuring an iIG when a sequence of inserted activities is detected.

The algorithm starts by detecting sets W_{r_1}, W_{r_2} and W_{r_3} of edges to be deleted. W_{r_1} (W_{r_2}) involves edges which link events before (after) the inserted sequence to activities belonging to the irregular trace. W_{r_3} contains edges connecting inserted activities. After

Algorithm 4 The Insertion Repairing Algorithm.

```
Let $\sigma_{ins} = \langle a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_j, \ldots, a_n\rangle$ be an irregular trace
Let $\langle a_i, \ldots, a_j\rangle$ be the sequence of inserted activities from position $i$ to $j$ of $\sigma_{ins}$
Let $I$ be the set of inserted activities of $\sigma_{ins}$
Let $IG_{\sigma_{ins}} = (V, W, \eta)$ be the iIG for the trace $\sigma_{ins}$, with $a_i = \eta(e_i)$
Let $CR$ be the causal relation
1: $W_{r_1} = \{(e_k, e_l) \mid k < i \wedge i \le l \le j \wedge ((e_k, e_l) \in W)\}$
2: $W_{r_2} = \{(e_k, e_l) \mid i \le k \le j \wedge l > j \wedge ((e_k, e_l) \in W)\}$
3: $W_{r_3} = \{(e_k, e_l) \mid i \le k \le j \wedge i \le l \le j \wedge (e_k, e_l) \in W)\}$
4: $W' = W \setminus (W_{r_1} \cup W_{r_2} \cup W_{r_3})$
5: for $k = j + 1$ to $n$ do
6:   if $\eta(e_k) \notin I \wedge ((\eta(e_{i-1}) \rightarrow_{CR} \eta(e_k)) \vee ((e_{i-1}, e_k) \in W)) \wedge (\langle e_j, \ldots, e_k\rangle \notin W')$ then
7:     $W' = W' \cup \{(e_j, e_k)\} \wedge W_{a_1} = W_{a_1} \cup \{(e_j, e_k)\}$
8: if $\eta(e_{i-1}) \nrightarrow_{CR} \eta(e_{i+1})$ then
9:   $W' = W' \cup \{(e_{i-1}, e_i)\} \wedge W_{a_2} = W_{a_2} \cup \{(e_{i-1}, e_i)\}$
10: else
11:   for $k = i - 1$ to 1 do
12:     if $\eta(e_k) \notin I \wedge ((\eta(e_k) \rightarrow_{CR} \eta(e_{j+1})) \vee ((e_k, e_{j+1}) \in W)) \wedge (\langle e_k, \ldots, e_i\rangle \notin W')$ then
13:       $W' = W' \cup \{(e_k, e_i)\} \wedge W_{a_2} = W_{a_2} \cup \{(e_k, e_i)\}$
14: $W_{a_3} = \{(e_k, e_{k+1}) : i \le k \le (j - 1)\}$
15: $W' = W' \cup W_{a_3}$
16: $W_{r_4} = \{(e_k, e_l) \mid (e_k, e_h) \in W_{a_2} \wedge i \le h \le j \wedge (e_p, e_l) \in W_{a_1} \wedge i \le p \le j\}$
17: $W' = W' \setminus W_{r_4}$
18: if $\eta(e_{i-1}) \nrightarrow_{CR} \eta(e_{i+1})$ then
19:   $W_{r_5} = \{(e_{i-1}, e_k) \in W' \mid k > i\}$
20:   $W' = W' \setminus W_{r_5}$
```

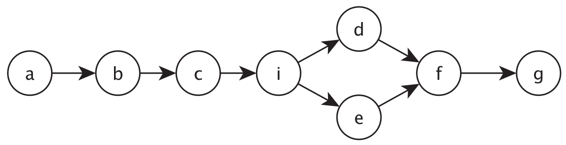


Fig. 10. Repaired graph for σ_{in_1}.

the removal of edges, the algorithm extracts the sets W_{a_1}, W_{a_2} and W_{a_3} of edges to be added. W_{a_1} has edges linking the last inserted activity to subsequent events. The building of the set W_{a_2} depends on the presence of inserted activities between two parallel events in the graph. Note that in both cases we don't insert an edge between two inserted activities, i.e., edges of the sets W_{a_1}, W_{a_2} link a regular activity and an inserted one. The case in which the statement (in step 8) is true will be detailed at the end of the subsection. Otherwise, in the case of insertion between non-parallel activities, W_{a_2} contains edges linking the first inserted activity to previous events. Note that the iIG on which the algorithm works could be already transformed to repair previous irregularities, so we might have edges between events which correspond to activities for which CR is not defined. For this reason in W_{a_1} as well as W_{a_2} we need to specify that an edge is built if either the CR is defined between corresponding activities, or an edge exists in the graph. Note that an edge can be added to W_{a_1} or W_{a_2} only if no paths, connecting events in the edge, already exist. In fact, if an edge between e_i and e_j exists, then $\langle e_i, e_j\rangle$ is the only path between the two events (see Definition 18). Then W_{a_3} simply adds an edge between each consecutive activities within the sequence. Finally, the set W_{r_4} is defined to remove edges which link nodes connected to the inserted activities in previous steps.

Example 8. Let us consider $\sigma_{in_1} = \langle a, b, c, \mathbf{i}, d, e, f, g\rangle$, of Example 3, where $W = \{(a, i), (a, b), (b, c), (c, d), (c, e), (d, f), (e, f), (f, g)\}$. In this case, the algorithm returns $W_{r_1} = \{(a, i)\}$; $W_{r_2} = W_{r_3} = \emptyset$; $W_{a_1} = \{(i, d), (i, e)\}$; $W_{a_2} = \{(c, i)\}$; $W_{a_3} = \emptyset$; $W_{r_4} = \{(c, d), (c, e)\}$. The repaired graph is the one shown in Fig. 10.

The statements in steps 8-9 and 18–20 of the algorithm define a variant for the set W_{a_2} and introduce the set W_{r_5}. These steps allow to handle a special case of insertion, namely activities inserted between parallel activities. In the remaining of this subsection, we briefly show the motivations for the definition of this variant.

Example 9. Let us consider $\sigma_{in_2} = \langle a, b, c, d, \mathbf{x}, e, f, g\rangle$, where x is the inserted activity. The initial edges set is $W = \{(a, b), (b, c), (c, d), (c, e), (d, f), (e, f), (f, g)\}$. The iIG is reported in Fig. 11(a). Since x is not in the model, the graph has a disconnected node for x. If we do not consider statements in steps 8 and 18, the removing edges sets are $W_{r_1} = W_{r_2} = W_{r_3} = \emptyset$, while the other sets are $W_{a_1} = \{(x, f)\}$, $W_{a_2} = \{(c, x)\}$, $W_{a_3} = \emptyset$. The resulting repaired graph is shown in Fig. 11(b).

It is noteworthy that the inserted activity x has been considered as parallel to d and e; indeed, since CR is not defined for these two activities, x is linked to the predecessor and the follower of both of them. The adding of another parallel branch increases the number of occurrence sequences, hence the overgeneralization of the IG. In order to reduce this phenomenon, we add the inserted activity in one of the already existing parallel branches. To this end, in W_{r_5} the new parallel branch is removed, and in W_{a_2} the inserted activity is moved to an existing path originates from its direct predecessor in the trace. Fig. 12 shows the effect of the variant of the algorithm.

It is important to note that when a trace involves different kinds of irregular behavior, first all the deleted activities are repaired, then the inserted ones (see Algorithm 2). This repairing order depends on the fact that when one or more deletions occur, the IR algorithm could be not able to correctly connect the inserted activities. This issue is discussed through the following example.

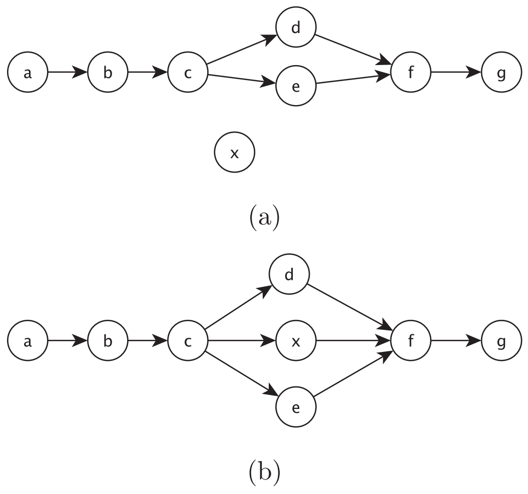


Fig. 11. iIG (a) and the partially repaired graph (b) for σ_{in_2}.

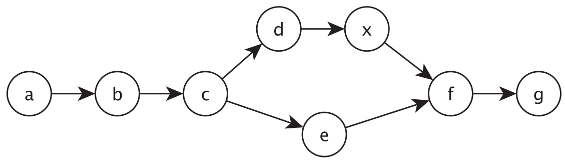


Fig. 12. Repaired IG for l_{in_2}.

Example 10. Let us consider the trace $\sigma_{in-del} = \langle a, b, i, d, e, f, g\rangle$; here we have an inserted event i in the 3rd position and a deleted event c in the 4th position. The initial edges set is $W = \{(a, b), (a, i), (d, f), (e, f), (f, g)\}$; the graph is shown in Fig. 13a. First, let us repair the graph by following only the order in which the irregular behaviors have occurred, i.e., first we repair the insertion and then the deletion. The removing edges set are $W_{r_1} = \{(a, i)\}$, $W_{r_2} = \emptyset$, $W_{r_3} = \emptyset$. As regards adding edges, $W_{a_1} = \emptyset$ and $W_{a_2} = \emptyset$; in fact, both the activities occurred before and after i should have been linked to c, which however has been skipped. As a result, i cannot be linked to anything. At the end of the insertion repairing, we have $W' = \{(a, b), (d, f), (e, f), (f, g)\}$. As regards the deletion repairing, we have: $W_{r_1} = W_{r_2} = \emptyset$. The edges to add are (b, d) and (b, e). Hence, the final edge set is $W' = \{(a, b), (b, d), (b, e), (d, f), (e, f), (f, g)\}$. The resulting graph is shown in Fig. 13b. The presence of a disconnected node leads to undesired overgeneralization. Let us now perform the repairing by following the Algorithm 2; first we consider the deletion and then the insertion. Now, after the deletion repairing we have $W' = \{(a, b), (a, i), (b, d), (b, e), (d, f), (e, f) (f, g)\}$. With respect to the previous case, the insertion repairing now returns $W_{a_1} = \{(i, d), (i, e)\}$ and $W_{a_2} = \{(b, i)\}$. Hence, $W' = \{(a, b), (b, i), (i, d), (i, e), (d, f), (e, f), (f, g)\}$; the repaired graph is shown in Fig. 13c, which shows a considerably reduced overgeneralization.

6. Experiments

In this section we evaluate the IGs obtained by our Building Instance Graph methodology adopting the Heuristic Miner (HM) and the infrequent Inductive Miner (iIM) to derive *CR*. We compare the obtained IGs with (1) IGs obtained by adopting the techniques proposed in (van Dongen & van der Aalst, 2004) and in

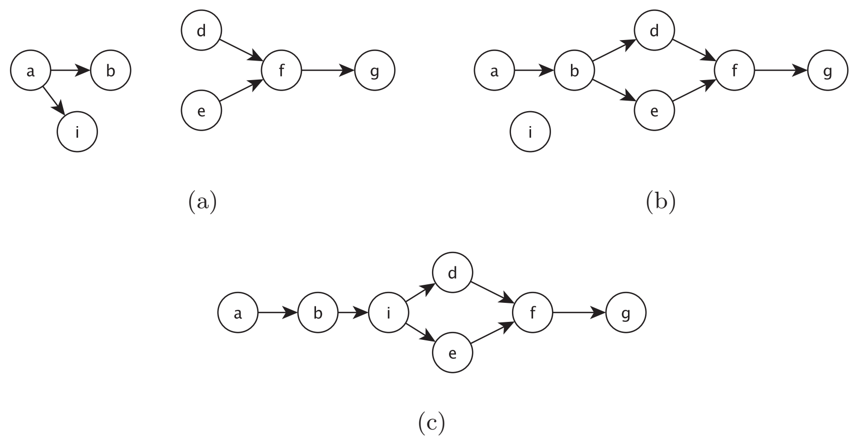


Fig. 13. (a) iIG for σ_{in-del}; (b) repaired graph when insertion repairing is executed before deletion repairing; (c) repaired graph following the execution order of Algorithm 2.

(van Beest et al., 2015), and (2) IGs obtained by using the HM and iIM without repairing. Experiments are performed both on synthetic (Subsection 6.1) and real-world (Subsection 6.2) event logs. In order to evaluate IGs, the following metrics are introduced:

- The *accuracy* (Acc), i.e., the number of graphs correctly reconstructed. We report both the numbers of correct IGs, and the percentage of correct IGs with respect to the overall IGs set. Note that, we evaluate accuracy only for the synthetic experiments, since the set of true IGs is unknown in real-world scenarios;
- The *matching cost* (MC), i.e., the distance between repaired and true IGs, which allows to evaluate the error made in building the IG. In order to compute MC, we refer to typical measures exploited to compare graphs. In particular, given an instance graph IG_σ, the execution of which has generated the trace σ, the difference between IG_σ and the reconstructed IG'_σ is evaluated as the number of transformations that are needed to make them isomorphic. Such transformations include a) adding or deleting an edge, b) adding or deleting a node, c) changing a label of an edge or a node and d) reversing the direction of an edge. In our case, all of these transformations are considered as having cost 1. When MC is 0 the IG has been correctly generated; the greater the MC the greater the error made in building the IG. Note that, like accuracy, MC can be measured only in the case of synthetic experiments;
- The *average generalization* (AG) of the IG set, which evaluates the number of occurrence sequences which can be generated on average by each IG. On the one hand, if AG is 1 then IGs are tailored to represent only the corresponding traces; on the other hand high values means overgeneralizing graphs, allowing for several different occurrence sequences. We compute the average generalization over the whole IG set (AG_{all}), the set of IGs obtained from regular traces (AG_{reg}) and the set of IGs obtained from irregular traces (AG_{irr}). When a non-filtering approach is adopted (e.g., for vD and vB) all traces fit the model, thus $AG_{reg} = AG_{all}$ and AG_{irr} can not be computed.

A metric commonly used in PM literature to evaluate the quality of mined models is the fitness, which evaluates to what extent the model is able to represent the set traces from which it was derived (see Section 3.3). Although in principle it is possible to compute the fitness also for an IG, by checking whether its corresponding trace is among its occurrence sequences, this metric turns out to be not useful to evaluate the IG quality. Indeed, the algorithms for building IGs always return an IG capable of representing its original trace. We have anyway computed the fitness for each IGs, to test the correctness of the algorithm; as expected, each trace fits its corresponding built IG.

6.1. Synthetic experiments

The first set of experiments have been performed on event logs generated by simulating the Petri net shown in Fig. 14. In this net, there are some “regular” (i.e., frequent) behaviors, represented by the paths involving solid edges; but several “irregular” (i.e., not frequent) behaviors can occur, whose paths are characterized by dotted edges. As an example, the activity *a* is usually followed by either *b* or *i*, but in rare cases *c* is executed immediately after *a*.

The adopted experimental procedure involves three main steps: 1) generating a set of process instances from the given Petri net and, hence, their *true* IGs and the event log; 2) building the IGs adopting different approaches; 3) evaluating the built IGs, comparing them with the true ones.

As regards the event log generation, we consider process instances with at most one irregular path; hence, irregular behaviors are variants of regular ones and differ only for the irregular path. This constraint limits the variability of the experiments, without affecting the goal.

From the model, it turns out that there are thirteen irregular behaviors. As regards the regular behaviors, instead, there are only two possible paths: I_1, that moves from the event *a* to the event *b* (i.e., the upper path in Fig. 14) and I_2, that follows the lower path from *a* to *g* through *i*. Note that I_2 can generate only one kind of trace, while I_1 can generate four possible kinds of traces, depending on the order in which activities *d* and *e* are performed and by the execution of the cycle. We grouped traces from I_1 into two groups: *G1* involves traces having *d* performed before *e*, while the second one, *G2* having *e* performed before *d*; each group can be instantiated with or without cycle execution (whatever the number of executed cycle instances is). The groups of traces of I_1 are shown in Fig. 15. Note that, same considerations hold also for any irregular behaviors which are variants of I_1; also in this case there are four irregular traces.

We performed several experiments, varying the event log in order to increase the number of regular traces. First, we randomly generated one trace for each irregular behavior, thus obtaining thirteen irregular traces; then, at each step we added an increasing number of regular traces, randomly generated by the regular behaviors. Since I_2 corresponds only to one kind of trace, while I_1 corresponds to two different groups of traces, we added 1/3 of the regular traces from the instance I_2, and the remaining 2/3 from I_1. We started with a log involving 16 traces (i.e., 13 irregular traces plus 2 traces from I_1 and 1 from I_2), until obtaining a log

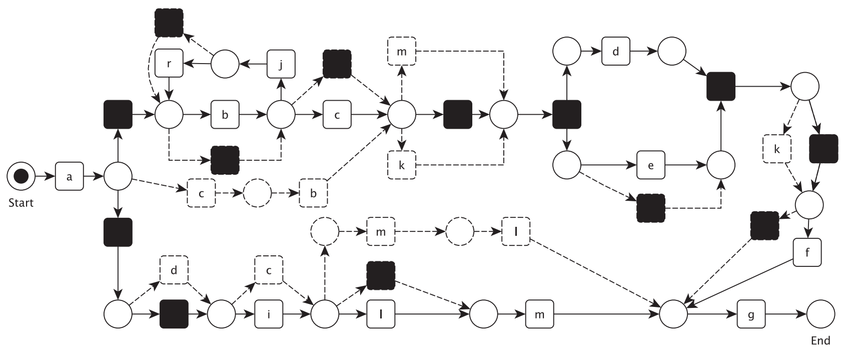
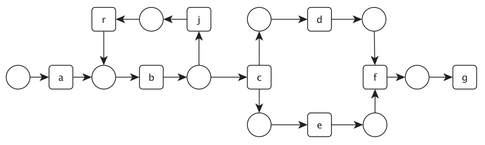


Fig. 14. The Petri net used for synthetic experiments.



G1	abcdefg abjrbcdefg
G2	abcdfeg abjrbcedfg

(a)

Fig. 15. Group of traces corresponding to I_1.

Table 1
Accuracy (number and percentage of correctly reconstructed IGs) obtained by varying the percentage of irregular traces (IrT) in the synthetic log. Results of 6 experiments: vD, vB, BIG^-_{HM}, BIG_{HM}, $\text{BIG}^-_{\text{iIM}}$, and BIG_{iIM}.

Trace	IrT(%)	Acc	vD	vB	BIG^-_{HM}	BIG_{HM}	BIG^-_{iIM}	BIG_{iIM}
16	81.3%	#IG	2	3	3	12	5	12
		%IG	12.5%	18.8%	18.8%	75.0%	31.3%	75.0%
19	68.4%	#IG	2	3	5	14	7	14
		%IG	10.5%	15.8%	26.3%	73.7%	36.8%	73.7%
22	59.1%	#IG	2	3	10	22	9	16
		%IG	9.1%	13.6%	45.5%	100%	40.9%	72.7%
27	48.2%	#IG	2	3	15	27	12	20
		%IG	7.4%	11.1%	55.6%	100%	44.4%	74.1%
34	38.2%	#IG	2	3	21	34	16	24
		%IG	5.9%	8.8%	61.8%	100%	47.1%	70.6%
49	26.5%	#IG	2	3	36	49	26	34
		%IG	4.1%	6.1%	73.5%	100%	53.1%	69.4%
99	13.1%	#IG	2	3	86	99	60	68
		%IG	2.0%	3.0%	86.9%	100%	60.6%	68.7%
199	6.5%	#IG	2	3	186	199	126	134
		%IG	1.0%	1.5%	93.5%	100%	63.3%	67.3%
298	4.4%	#IG	2	3	285	298	191	199
		%IG	0.7%	1.0%	95.6%	100%	64.1%	66.8%

Table 2
Matching Cost (Total and Average) obtained by varying the percentage of irregular traces (IrT) in the synthetic log. Results of 6 experiments: vD, vB, BIG^-_{HM}, BIG_{HM}, $\text{BIG}^-_{\text{iIM}}$, and BIG_{iIM}.

Trace	IrT(%)	MC	vD	vB	BIG^-_{HM}	BIG_{HM}	BIG^-_{iIM}	BIG_{iIM}
16	81.3%	Tot	105	88	63	20	62	20
		Avg	7.5	6.8	4.8	5.0	5.6	5.0
19	68.4%	Tot	118	103	68	25	67	25
		Avg	6.9	6.4	4.9	5.0	5.6	5.0
22	59.1%	Tot	131	118	50	0	72	30
		Avg	6.6	6.2	4.2	0.0	5.5	5.0
27	48.2%	Tot	152	143	50	0	80	35
		Avg	6.1	6.0	4.2	0.0	5.3	5.0
34	38.2%	Tot	183	187	51	0	95	50
		Avg	5.7	6.0	3.9	0.0	5.3	5.0
49	26.5%	Tot	248	354	51	0	120	75
		Avg	5.3	7.7	3.9	0.0	5.2	5.0
99	13.1%	Tot	464	604	51	0	200	155
		Avg	4.8	6.3	3.9	0.0	5.1	5.0
199	6.5%	Tot	898	1308	51	0	370	325
		Avg	4.6	6.7	3.9	0.0	5.1	5.0
298	4.4%	Tot	1327	1814	51	0	540	495
		Avg	4.5	6.1	3.9	0.0	5.0	5.0

with 298 traces (i.e., 13 irregular plus 190 from I_1 and 95 from I_2). Clearly, as the number of the regular traces in the log increases, the percentage of irregular traces (IrT) decreases. Hence, we move from IrT=81.25%, to IrT=4.36%. Details are reported in Tables 1 and 2.

Tables also report accuracy and matching cost for the 6 different approaches analyzed: van Dongen (vD), van Beest (vB), the BIG algorithm using Heuristic Miner (BIG_{HM}) and infrequent Inductive Miner (BIG_{iIM}) and the BIG^- algorithm, which corresponds to the BIG algorithm without applying the repairing (i.e., without performing steps 4–6 in Algorithm 1), using the same PD techniques (BIG^-_{HM}, $\text{BIG}^-_{\text{iIM}}$). Note that, since both vD and vB adopt the α-algorithm, that is a non-filtering PD technique (see Section 4), no repairing strategy is needed. Table 1 reports the accuracy of each set of built IGs, in terms of number and percentage of correctly reconstructed IGs; while Table 2 reports both the total MC, i.e., the sum of the MC computed on the whole log, and the average MC, which is computed only on the wrong graphs, since for correct graphs the value is zero.

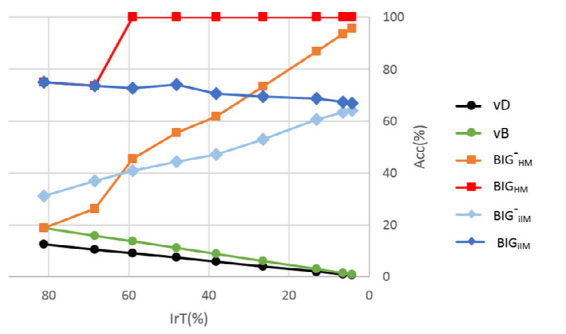


Fig. 16. Accuracy Vs. percentage of irregular traces.

Fig. 16 provides a graphical representation of the accuracy trends with respect to the percentage of irregular traces in the event log.

First, we can note that the worst results are provided by vD, that is able to correctly build only two IGs of the entire set. vB obtains only slightly better results, being able to correctly build three IGs of the set. These values do not change by reducing the percentage of irregular traces, resulting in a decay of the accuracy for both the approaches; indeed, the corresponding lines in Fig. 16 tend to an accuracy of 0%. This trend is due to the presence of swaps in the irregular traces. The α-algorithm considers the pairs b and c, and l and m as parallels; hence, it is not able to properly build the IGs corresponding to regular traces. Indeed, the only graphs that are properly reconstructed by vD are those which correspond to traces involving the deletion of l and the deletion of b, respectively; while vB was able to reconstruct also the IG corresponding to the trace involving the deletion of c. This is due to the fact that vB connects the activity b to both the activities d and e, since no explicit parallelism is detected between them, while vD cannot connect b and d, since no causal relation exists between them. As regards $\text{BIG}^{-}_{\text{HM}}$, it provides better results than vD and vB, as expected. However, for the first event log it shows the same performance of vB, since it is able to reconstruct three IGs. As a matter of facts, the model mined by HM describes almost all regular behavior of Fig. 14, but the activities l and m are considered as parallel, with the result that the IGs involving these activities cannot be properly reconstructed. As IrT reduces, HM is able to correctly represent regular traces, thus the $\text{BIG}^{-}_{\text{HM}}$ error tends to decrease (see Fig. 16); in particular, from 70% to 60% there is an improvement in accuracy of the 20%. Then, as the number of regular traces increases, the outcome of the algorithm stabilizes and the accuracy trends becomes linear.

Similar considerations hold for $\text{BIG}^{-}_{\text{iIM}}$. At the beginning it obtains poor results, because of the high percentage of noise, which leads to induce incorrect parallelisms; by reducing the noise in the log, accuracy increases. However, it is noteworthy that for IrT less than 60%, $\text{BIG}^{-}_{\text{HM}}$ obtains better results than $\text{BIG}^{-}_{\text{iIM}}$. The accuracy of $\text{BIG}^{-}_{\text{HM}}$ tends to 100%, while for $\text{BIG}^{-}_{\text{iIM}}$ the accuracy tends to 65%. This depends again on the identification of a wrong parallelism between the activities l and m, which is kept also for low percentage of irregular traces; the result is that the IGs built from traces involving these activities turn out to be incorrect.

When the repairing procedure is applied, the accuracy increases significantly. Both BIG_{HM} and BIG_{iIM} outperform results obtained with same models without repairing. For the first log the improvement is 43.75% using the iIM and 56.25% using the HM. Decreasing IrT, the accuracy of BIG_{iIM} shows a linear trend with a little slope (from 73% to 67%); while BIG_{HM} moves to 100%, namely it is able to reconstruct all IGs. When IrT is very low, irregular traces are very rare and the scenario moves from unstructured to structured one. In this scenario, the effect of repairing is almost negligible.

As regards the MC, results are consistent with the considerations done for accuracy. Indeed, both vD and vB obtain for all the event logs the highest values of total MC. This was expected, since they turned out to be able to correctly build only two IGs in all the cases, respectively. , $\text{BIG}^{-}_{\text{HM}}$ and , $\text{BIG}^{-}_{\text{iIM}}$ obtain similar values at the beginning, but , $\text{BIG}^{-}_{\text{HM}}$ improves significantly its values by decreasing the percentage of irregular traces. The repaired approaches are the best ones; in particular BIG_{HM} obtains a cost equal to zero when the IrT decreases under 68%, and BIG_{iIM} obtains a total MC much lower than $\text{BIG}^{-}_{\text{iIM}}$. Let us now consider the average MC, which provides an indication about the magnitude of the error, i.e., how many errors each approach has done on average when building a graph. Note that this value has to be evaluated with respect to the average number of elements (i.e., nodes and edges) per graph, which results in [15; 17] for all event log. Again, vD and vB are the approaches that return the highest number of errors per graph, ranging from 4.48 to 7.5, and from 5.9 to 7.69 respectively. $\text{BIG}^{-}_{\text{iIM}}$ and BIG_{iIM} return for all the event logs an average MC around 5; it depends on errors in the model mined by the iIM, as we discussed before. Finally, approaches based on HM

Table 3
Average Generalization over the whole IG set (AG_{all}), the set of regular IGs (AG_{reg}) and the set of irregular IGs (AG_{irr}) obtained by varying the percentage of irregular traces (IrT) in the synthetic log. Results of 6 experiments: vD, vB, $\text{BIG}^{-}_{\text{HM}}$, BIG_{HM}, $\text{BIG}^{-}_{\text{iIM}}$, and BIG_{iIM}.

Trace	IrT(%)	AG	vD	vB	BIG^{-}_{HM}	BIG_{HM}	BIG^{-}_{iIM}	BIG_{iIM}
16	81.3%	AG_{all}	7.8	3.6	23.7	1.9	19.4	1.9
		AG_{reg}	7.8	3.6	2.0	2.0	2.0	2.0
		AG_{irr}	–	–	28.7	1.9	23.5	1.9
19	68.4%	AG_{all}	7.2	3.5	20.3	1.9	16.7	2.0
		AG_{reg}	7.2	3.5	2.0	2.0	2.0	2.0
		AG_{irr}	–	–	28.7	1.9	23.5	1.9
22	59.1%	AG_{all}	6.7	3.5	16.5	1.7	14.7	2.0
		AG_{reg}	6.7	3.5	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	26.8	1.7	23.5	1.9
27	48.2%	AG_{all}	6.3	3.5	13.8	1.7	13.2	2.0
		AG_{reg}	6.3	3.5	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	26.7	1.7	25.2	1.9
34	38.2%	AG_{all}	5.8	3.4	11.6	1.7	10.9	2.0
		AG_{reg}	5.8	3.4	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	27.7	1.7	25.2	1.9
49	26.5%	AG_{all}	5.2	3.4	8.6	1.7	8.2	2.0
		AG_{reg}	5.2	3.4	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	27.7	1.7	25.2	1.9
99	13.1%	AG_{all}	4.6	3.4	5.1	1.7	5.1	2.0
		AG_{reg}	4.6	3.4	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	27.7	1.7	25.2	1.9
199	6.5%	AG_{all}	4.3	3.4	3.4	1.7	3.5	2.0
		AG_{reg}	4.3	3.4	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	27.7	1.7	25.2	1.9
298	4.4%	AG_{all}	4.2	3.3	2.8	1.7	3.0	2.0
		AG_{reg}	4.2	3.3	1.7	1.7	2.0	2.0
		AG_{irr}	–	–	27.7	1.7	25.2	1.9

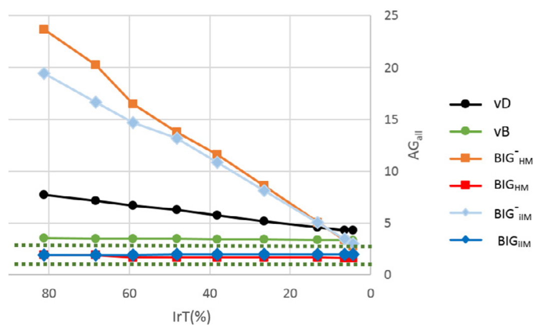


Fig. 17. Average generalization Vs. percentage of irregular traces. Synthetic event log.

algorithm show the best values ($\text{BIG}^{-}_{\text{HM}}$ in [3.92; 4.84], while BIG_{HM} is 5 for IrT > 60%, 0 otherwise), demonstrating that the model obtained by using the HM algorithm is the best one.

Table 3 shows the AG values obtained by the performed experiments.

In order to evaluate the values of the AG, we need a reference value for IGs with "good" generalization. To this end, we refer to AG_{reg} obtained by filtering approaches. As a matter of fact, since in these experiments irregular traces are variants of regular traces they differ each other for few changes; hence AG_{reg} and AG_{irr} should assume similar values. We emphasize that to compute AG we do not use a-priori knowledge about the true process model; hence, regular and irregular traces are the ones which perfectly fit and not fit the model respectively. From Table 3, it turns out that the AG_{reg} for BIG_{HM} is 1.7 on average, while for BIG_{iIM} is 2; hence, we consider the interval [1.7; 2] as reference.

Fig. 17 shows the AG_{all} obtained by the six experimented approaches with respect to IrT; the interval of reference values is represented by dotted lines. Both $\text{BIG}^{-}_{\text{HM}}$ and $\text{BIG}^{-}_{\text{iIM}}$ show worse results than vD and vB. This is due to the presence of deleted

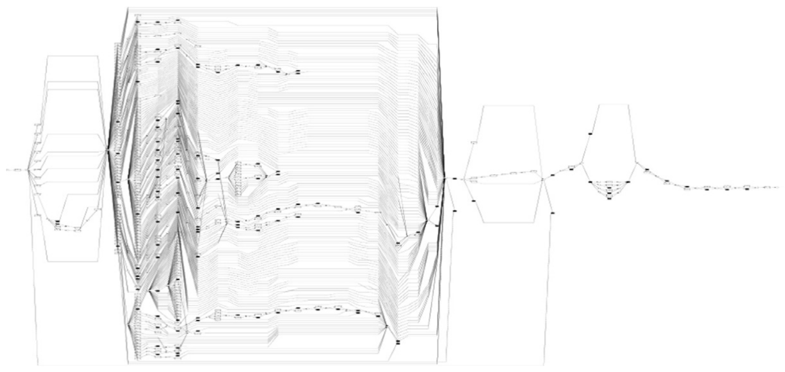


Fig. 18. Petri Net mined by the iIM from the WABO4 event log.

activities, which leads to disconnected graphs and in turn to high overgeneralization (see Example 4). This statement is supported by results in Table 3, where AG_{irr} is significantly higher than AG_{reg}. We can note that vB obtains better results than vD. Indeed, the latter is also affected by a relevant overgeneralization ranging in [4.2; 7.75], that is more than double of the reference values; while vB obtains quite stable results, around 3 traces per graph, that is closer, but still outside the reference interval.

Finally, by applying the repairing procedure we obtain the best results, both for BIG_{HM} and for BIG_{iIM}; their lines overlap almost completely in Fig. 17. The resulting AG_{all} and AG_{irr} are in the reference interval, hence showing the capability of the proposed repairing procedure of preventing overgeneralization.

6.2. Real-world event log

This subsection shows the results obtained on *WABO4*, which is one of the real-world event logs of the CoSeLoG project[5]. WABO4 refers to the process of environmental permit application of a Dutch municipality. The log involves 332 different activities in 787 traces. Fig. 18, which shows the Petri net mined by the iIM, allows us to perceive the highly variable nature of the process.

We preprocessed the event log by adding artificial start and end activities to all traces.

For this experiment we could not use the HM. It is worth noting that HM does not guarantee to obtain a sound process model (i.e., a model where all process steps can be executed and the final marking is always reachable), leading to problems in conformance checking (Leemans et al., 2014). In particular, using HM to mine a model for WABO4, the conformance checker is able to define an alignment only for 5 of 787 traces; in other words, for most of traces we cannot know if they are regular or irregular, hence, we cannot apply the proposed repairing procedure. For this reason, we used only iIM as filtering PD technique, that always returns a sound process model. It is worth noting that there are 485 irregular traces with respect to the model mined by iIM, that is around 62% of the whole log. Nevertheless, the model has a high fitness (i.e., the ability to reproduce traces in the log), which is around 98%. This can be explained by considering that the ratio between the number of inserted/deleted activities and the number of activities per trace is low (we have around two inserted/deleted activities for trace on average, and each trace is formed by 46 events on average); hence, a relevant portion of each trace can be actually replied by the model. Table 4 shows the *AG* values obtained on the WABO4 event log by vD, vB, BIG^{-}_{iIM} and BIG_{iIM} approaches. Note that the values reported for vD and vB represent lower bounds

Table 4
AG results obtained on WABO4 event log.

Trace	IrT(%)	AG	vD	vB	BIG^{-}_{iIM}	BIG_{iIM}
787	61.3%	AG_{all}	$\gg$ 99.14	$\gg$ 92.82	15.4	9.5
		AG_{reg}	$\gg$ 99.14	$\gg$ 92.82	3.8	3.8
		AG_{irr}	–	–	22.7	9.5

of actual *AG* values. Indeed, applying vD and vB we obtained very complex IGs, with a high number of edges, which made generating the entire occurrence sequences set for each IG time-consuming. Therefore, for vB and vD we limited the occurrence sequences generation to the first 100. Noteworthily, in both cases we found few IGs generating less than 100 occurrence sequences; therefore, we had that most of the IGs (691 for vB, 776 for vD) generated a number of occurrence sequences at least equal (but very likely much bigger) than 100, thus motivating the high AG_{all} values computed for these algorithms.

Applying iIM, we obtain significantly better generalization results. In particular, it turns out that BIG_{iIM} outperforms the BIG^{-}_{iIM}. Indeed, BIG^{-}_{iIM} shows still a quite high AG_{all} value (i.e., 15.4 occurrence sequences per graph); while applying the repairing the AG_{all} decreases to 9.5. The improvement achieved by the repairing is especially evident for the irregular traces, where we move from $AG_{irr} = 22.7$ of BIG^{-}_{iIM} to $AG_{irr} = 13$ when the repairing is applied.

The results achieved in terms of generalization values prove that the proposed approach was actually able to obtain IGs significantly more accurate than other techniques, especially when the repairing is applied.

In order to better appreciate the effect of the proposed approach, in Fig. 19 we show an example of iIG built by BIG^{-}_{iIM} and BIG_{iIM} for a trace σ_1 extracted from the WABO log, involving 78 events; for the sake of space, in the Figure only the portions of IGs including irregularities are shown, black dots represent the remaining parts of the process. The trace has a deleted activity, i.e., *01_HOOFD_510_2a*, and an inserted activity, i.e., *16_LGSD_010*. Fig. 19a shows an evident structural problem: besides *START* node the process has two other nodes without predecessors, namely *01_HOOFD_510_3* and *01_HOOFD_515*, and two nodes without successors, namely *END* and *16_LGSD_010*. Hence, BIG^{-}_{iIM} returns an IG having a very high number of occurrence sequences. Fig. 19b shows the IG obtained by BIG_{iIM}; the repairing procedure corrects the structural problems and provides a clear and not ambiguous representation of the execution order of corresponding process instance. Moreover, in the dotted box of Fig. 20a and b we show the same portion of IGs as returned by applying vD and vB approaches. Differing from IGs obtained by BIG_{iIM}, vB and vD approaches return IGs with a high number of parallel behaviors; hence revealing a significant overgeneralization; the number of

[5] http://data.3tu.nl/repository/collection:event_logs_real

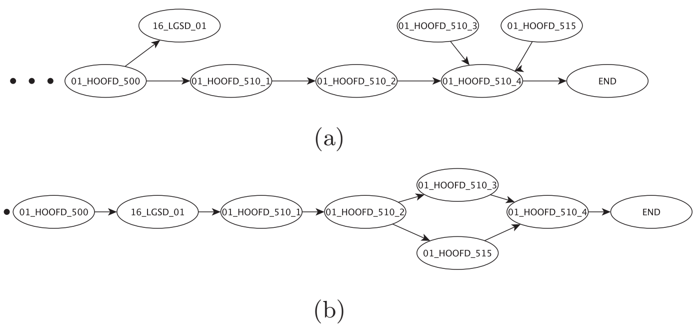


Fig. 19. Instance Graph obtained by BIG^{-}_{iIM} (a) and BIG_{iIM} (b) for σ_1.

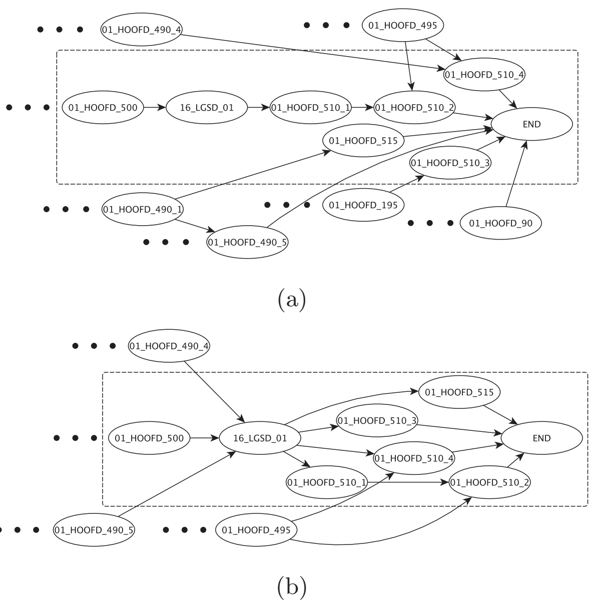


Fig. 20. Instance Graph obtained by vD (a), vB (b) for σ_1.

occurrence sequences is four for BIG_{iIM} and more than 100 for other approaches.

Figs. 19 and 20 clearly show the advantage of using our approach with respect to other ones; indeed, it is apparent that overgeneralizing IGs do not provide useful insights on the execution of the process.

7. Conclusions

This work introduced a methodology for building Instance Graphs for highly variable processes starting from a simple event log. IGs offer a different perspective on the process, focusing on the analysis of individual process instances. The proposed approach bases the generation of IGs on casual relations among activities, which are inferred from the log by resorting to PD techniques. In particular filtering PD techniques are adopted, which represent only the most relevant casual relations, thus cutting down overgeneralization. However, filtering leads to irregular traces; hence, Instance Graphs built for irregular traces are not accurate. To address this issue, we couple the exploitation of filtering techniques with an original technique aimed at repairing graphs.

We performed several experiments, both on synthetic and real-world event logs, to assess the advantages of the approach. It turns out that our proposal outperforms other approaches both in terms of accuracy and average generalization. In particular, in the synthetic experiments we were able to evaluate the the correctness of the methods with respect to reference IGs. Results show that techniques based on the α-algorithm were not able to generate the correct IGs in most cases even on event logs with a very low percentage of irregularities, since also few irregular traces led to overgeneralizing models. Adopting filtering approaches instead, the effects of the irregularities were significantly reduced, resulting in more IGs correctly reconstructed. However, from experiments it is also apparent that filtering approaches without repairing perform really bad in terms of average generalization, obtaining worse results than van Dongen's and van Beest's approaches. Such a result pointed out the importance of the repairing procedure, which corrects structural anomalies and improves accuracy and generalization. The results obtained in the synthetic case were confirmed in the real-world case. Here, the α-based approaches obtained really high values of average generalization, thus meaning that the corresponding IGs involved many parallelisms, as also shown in examples from which it was apparent that these IGs do not provide much aid to the analysis of process instances. Similar considerations also hold for filtering approaches without repairing; while the introduction of repairing allowed to obtained significantly better performance.

Experiments also enlightened that the feasibility of the proposed repairing technique is constrained to the availability of sound models, due to the conformance checking technique adopted. This in turn put a constraint on the PD technique to be used to mine the model. Nevertheless, we like to note that iIM guarantees to always return a sound model, so that we can state that our approach can be applied to any event log, although the model mined by iIM could not be the best in terms of generalization performance. Hence the possibility to detect irregular traces also on unsound models could provide more flexibility in selecting the most accurate technique for the data at hand. We plan to investigate the usage of different conformance checking techniques to detect irregular traces in future work. We plan also to investigate the effects of the usage of Process Discovery techniques other than Heuristic Miner and infrequent Inductive Miner. Furthermore, we intend to further develop the repairing algorithm for special cases. For example, we intend to analyze and evaluate alternative strategies to deal with the insertion of activities occurring in the middle of parallel branches.

References

Adriansyah, A., van Dongen, B. F., & van der Aalst, W. (2011). Conformance checking using cost-based fitness analysis. In *Proceedings of the 15th IEEE international enterprise distributed object computing conference* (pp. 55–64). Helsinki, Finland: IEEE.

Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B. F., & van der Aalst, W. M. (2013). Alignment based precision checking. In M. La Rosa, & P. Soffer (Eds.), *Business process management workshops*. In *Lecture notes in business information processing.: Vol. 132* (pp. 137–149). Berlin Heidelberg: Springer.

Buijs, J. C. A. M., La Rosa, M., Reijers, H. A., van Dongen, B. F., & van der Aalst, W. M. P. (2013). Improving business process models using observed behavior. In P. Cudre-Mauroux, P. Ceravolo, & D. Gašević (Eds.), *Data-driven process discovery and analysis*. In *Lecture notes in business information processing: Vol. 162* (pp. 44–59). Berlin Heidelberg: Springer.

De Medeiros, A. K. A., Guzzo, A., Greco, G., van der Aalst, W., Weijters, A. J. M. M., van Dongen, B. F., & Saccà, D. (2008). Process mining based on clustering: A quest for precision. In A. ter Hofstede, B. Benatallah, & H.-Y. Paik (Eds.), *Business process management workshops*. In *Lecture notes in computer science: Vol. 4928* (pp. 17–29). Berlin Heidelberg: Springer.

De Weerdt, D., De Backer, M., Vanthienen, J., & Baesens, B. (2012). A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. *Information Syst, 37*, 654–676.

Desel, J., Juhás, G., Lorenz, R., & Neumair, C. (2003). Modelling and validation with viptool. In W. Van der Aalst, & M. Weske (Eds.), *Business process management. Lecture notes in computer science: Vol. 2678* (pp. 380–389). Berlin Heidelberg: Springer.

Diamantini, C., Genga, L., & Potena, D. (2016). Behavioral process mining for unstructured processes. *Journal of Intelligent Information System*.

Diamantini, C., Genga, L., Potena, D., & Storti, E. (2013). Pattern discovery from innovation processes. In *Collaboration technologies and systems (CTS), 2013 international conference on* (pp. 457–464). San Diego, California, USA: IEEE.

Diamantini, C., Potena, D., & Storti, E. (2012). Mining usage patterns from a repository of scientific workflows. In *Proceedings of the 27th annual ACM symposium on applied computing* (pp. 152–157). ACM.

Fahland, D., & van der Aalst, W. (2015). Model repair—aligning process models to reality. *Information System, 47*, 220–243.

Forrester, P. (1999). Business process engineering: Reference models for industrial enterprises. *Integrated Manufacturing System, 10*.57–57

Greco, G., Guzzo, A., Manco, G., & Saccà, D. (2007). Mining unconnected patterns in workflows. *Information System, 32*, 685–712.

Hwang, S., Wei, C., & Yang, W. (2004). Discovery of temporal patterns from process instances. *Computers in Industry, 53*, 345–364.

Leemans, S. J., Fahland, D., & van der Aalst, W. (2014). Discovering block-structured process models from event logs containing infrequent behaviour. In N. Lohmann, M. Song, & P. Wohed (Eds.), *Business process management workshops*. In *Lecture notes in business information processing: Vol. 171* (pp. 66–78). Springer International Publishing.

Lu, X., Fahland, D., & van der Aalst, W. (2015). Conformance checking based on partially ordered event data. In F. Fournier, & J. Mendling (Eds.), *Business process management workshops*. In *Lecture notes in business information processing: Vol. 202* (pp. 75–88). Springer International Publishing.

de Man, H. (2009). Case management: A review of modeling approaches. *BPTrends, January, 2009*. http://www.ww.bptrends.com/publicationfiles/01-09-ART-%20Case%20Management-1-DeMan.1165%20doc--final.pdf. Last Access: 16/09/2015.

Rovani, M., Maggi, F. M., De Leoni, M., & Van Der Aalst, W. M. (2015). Declarative process mining in healthcare. *Expert System with Applications, 42*, 9236–9251.

Suriadi, S., Ouyang, C., van der Aalst, W. M., & ter Hofstede, A. H. (2015). Event interval analysis: Why do processes take time? *Decision Support System, 79*, 77–98.

van Beest, N., Dumas, M., García-Ba nuelos, L., & La Rosa, M. (2015). Log delta analysis: Interpretable differencing of business process event logs. In H. R. Motahari-Nezhad, J. Recker, & M. Weidlich (Eds.), *Business process management*. In *Lecture notes in computer science: Vol. 9253* (pp. 386–405). Springer International Publishing.

van der Aalst, W. (2011). *Process mining: Discovery, conformance and enhancement of business processes*. Springer-Verlag Berlin Heidelberg.

van der Aalst, W. (2013). Decomposing petri nets for process mining: A generic approach. *Distributed and Parallel Databases, 31*, 471–507.

van der Aalst, W., & van Hee, K. M. (2004). *Workflow management: Models, methods, and systems*. Cambridge, MA, USA: MIT press.

van Dongen, B., & van der Aalst, W. (2004). Multi-phase process mining: Building instance graphs. In P. Atzeni, W. Chu, H. Lu, S. Zhou, & T.-W. Ling (Eds.), *Conceptual modeling- ER 2004*. In *Lecture notes in computer science: Vol. 3288* (pp. 362–376). Berlin Heidelberg: Springer.

van Dongen, B., Alves de Medeiros, A., & Wen, L. (2009). Process mining: Overview and outlook of petri net discovery algorithms. In K. Jensen, & W. van der Aalst (Eds.), *Transactions on petri nets and other models of concurrency II*. In *Lecture notes in computer science: Vol. 5460* (pp. 225–242). Berlin Heidelberg: Springer.

van Dongen, B. F., & van der Aalst, W. (2005). Multi-phase process mining: Aggregating instance graphs into EPCs and petri nets. In D. Marinescu (Ed.), *Proceedings of the second international workshop on applications of petri nets to coordination, workflow and business process management* (pp. 35–58). Miami, Florida,USA: Florida International University.

Weijters, A., van der Aalst, W., & De Medeiros, A. A. (2006). Process mining with the heuristics miner-algorithm. *Technische Universiteit Eindhoven, Technical Report WP, 166*.1–34

Claudia Diamantini is associate professor at the Department of Information Engineering, Università Politecnica delle Marche, where she coordinates the degree courses in computer and automation engineering and leads the Knowledge Discovery & Management research group. She received the PhD degree in artificial intelligent systems from the University of Ancona in 1995. At present her main research interests are in business analytics and data mining, and in semantic models for integrated interoperable analytics and mining in distributed settings. She has been working on these topics within national and international projects. She is author of about 100 technical papers in refereed journals and conferences. She is a member of the IEEE and ACM.

Laura Genga is a PhD student of the Department of Information Engineering of the Università Politecnica delle Marche, where she graduated in computer and automation engineering in 2012. Her research interests are in the areas of data mining and knowledge discovery, process mining and innovation support.

Domenico Potena received the PhD in information systems engineering from the Università Politecnica delle Marche, Italy, in 2004. From 2005 to 2008, he was post-doctoral fellow at the same university. At present, he is an assistant professor at the Università Politecnica delle Marche, Department of Information Engineering. His research interests include knowledge discovery in databases, data warehousing, data semantics, innovation management systems.

Wil van der Aalst is a full professor of information systems at the Technische Universiteit Eindhoven (TU/e). He is also an adjunct professor at Queensland University of Technology (QUT). His research interests include workflow management, process mining, Petri nets, business process management, process modeling, and process analysis. He is an elected member of the Royal Holland Society of Sciences and Humanities (Koninklijke Hollandsche Maatschappij der Wetenschappen) and the Academy of Europe (Academia Europaea).