Knowledge and Information Systems (2020) 62:1143–1197
https://doi.org/10.1007/s10115-019-01376-9

REGULAR PAPER

Information-preserving abstractions of event data in process mining

Sander J. J. Leemans[1] · Dirk Fahland[2]

Received: 26 April 2018 / Revised: 18 June 2019 / Accepted: 23 June 2019/Published online: 20July 2019

Abstract
Process mining aims at obtaining information about processes by analysing their past executions in event logs, event streams, or databases. *Discovering* a process model from a finite amount of event data thereby has to correctly infer infinitely many unseen behaviours. Thereby, many process discovery techniques leverage *abstractions* on the finite event data to infer and preserve behavioural information of the underlying process. However, the fundamental information-preserving properties of these abstractions are not well understood yet. In this paper, we study the information-preserving properties of the "directly follows" abstraction and its limitations. We overcome these by proposing and studying two new abstractions which preserve even more information in the form of finite graphs. We then show how and characterize when process behaviour can be unambiguously recovered through characteristic footprints in these abstractions. Our characterization defines large classes of practically relevant processes covering various complex process patterns. We prove that the information and the footprints preserved in the abstractions suffice to unambiguously rediscover the exact process model from a finite event log. Furthermore, we show that all three abstractions are relevant in practice to infer process models from event logs and outline the implications on process mining techniques.

Keywords Process mining · Information preservation · Language abstraction · Model abstraction · Rediscoverability · Directly follows · Minimum self-distance · Inclusive choice

1 Introduction

Process mining comprises methods and techniques for understanding possibly unknown processes from recorded events. Input to process mining is usually an *event log* containing sequences of events that describe activity occurrences observed in the past. An event log is always a finite sample of the possibly infinite behaviour of the underlying process. *Process*

✉ Sander J. J. Leemans
s.leemans@qut.edu.au

1 Queensland University of Technology, Brisbane, Australia

2 Eindhoven University of Technology, Eindhoven, The Netherlands

discovery has the aim of learning a process model that can describe the behaviour of this unknown process [1].

Process discovery is typically studied in terms of formal languages as illustrated in Fig. 1: process S has unknown behaviour $\mathcal{L}(S)$ from which some process executions are recorded in a log $L \subseteq \mathcal{L}(S)$—a *finite language* over a set of activities. A process discovery algorithm then shall infer from finite L the remaining, infinite, unobserved behaviour of S and encode them in a model M, describing possible process executions as language $\mathcal{L}(M)$. Ideally, the discovered model M describes precisely the behaviour of the underlying process S, that is, $\mathcal{L}(M) = \mathcal{L}(S)$. In practice, the underlying process S is unknown, so finding the "right" model M is often seen as a pareto-optimization problem between fitness (maximize $L \cap \mathcal{L}(M)$), precision (minimize $\mathcal{L}(M) \setminus L$), generalization (maximize behaviour not seen in L but likely to occur in the future), and simplicity of M [2].

1.1 Information preservation in process discovery

In many time-efficient discovery algorithms [3–6], a function α first *abstracts* the known behaviour seen in L into abstract behavioural information $B = \alpha(L)$ on the activities recorded in L, thereby generalizing from the sample L. Common abstractions are the "directly follows" [3] and "eventually follows" [4] relations and their variants [7], and "behavioural profiles" [8] that also include "conflict" and "concurrency". The discovery algorithm then synthesizes a model $M = \gamma(B)$ from B using a *concretization* function γ, after optionally processing B [4,9,10], thereby interpreting information in abstraction B.

This renders the abstraction central to information preservation and inference. If for two logs (two different samples) L_x and L_y, the abstraction yields $\alpha(L_x) = \alpha(L_y)$, then any discovery algorithm using α will return the same model $\gamma(\alpha(L_x)) = \gamma(\alpha(L_y))$, even if L_x and L_y come from different processes. For example, the two different processes in Fig. 2a, b may yield logs $L_1 = \{\langle a, b, c\rangle, \langle a, c, b\rangle\}$ and $L_2 = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle a, b\rangle, \langle a, c\rangle\}$ both

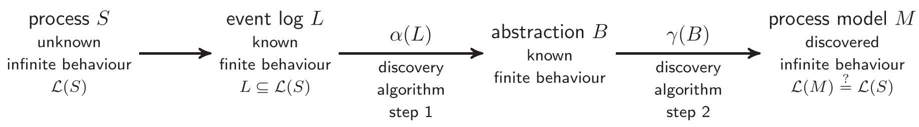
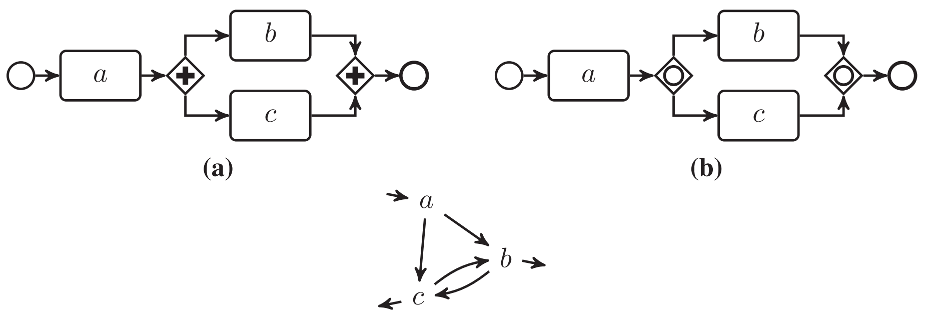


Fig. 1 Illustration of the context of process discovery with an abstraction function α



(c) Directly follows abstraction.

Fig. 2 Two process models in BPMN [11], (**a**, **b**), with a different language but the same directly follows abstraction (**c**)

having the same directly follows abstraction shown in Fig. 2c. The directly follows abstraction is not preserving all information of Fig. 2a, b, and any discovery algorithm working with this abstraction alone cannot distinguish these logs.

Beyond event logs, stream process discovery techniques require information-preserving abstraction functions to retain information from event streams in a memory-efficient representation [12], and event databases use indexes based on abstractions [13] to efficiently discover models from very large event data sets [14]. This importance of α gives rise to two questions:

- When does α extract the “right” behavioural information about the underlying process S so that γ can use this information to obtain the “right” model, M, with $\mathcal{L}(S) = \mathcal{L}(M)$?
- What information from a log has to be used and stored in a finite abstraction α to allow inferring the “right” model M?

1.2 State of the art

Although considerable progress has been made in recent years by using different abstraction and concretization functions in process discovery, more complex behaviour can still not be accurately discovered from event data [10,15]. At the same time, the inherent information-preserving properties of the used abstractions have received considerable less attention. That is, the class of models that can be (re-)discovered has been precisely characterized for specific algorithms [3,16–22]; however, a systematic study into the information preserved by the abstractions that these algorithms use has not been performed.

Information preservation and recovery for behavioural profiles independent of the alpha-algorithm are only known for acyclic processes, while behavioural profiles cannot preserve information of regular languages in general [8]. Information-preserving “footprints” of directly follows abstraction have been studied [5,23] and used in multiple discovery algorithms [5,6] for basic process operators. No existing study considers information preservation in abstractions for practically relevant and frequent process patterns such as unobservable skip paths, interleaving or the inclusive gateways of Fig. 2b.

1.3 Contribution

In this paper, we study how and under which circumstances event data abstractions preserve behavioural information in a way that allows to unambiguously recover basic and complex process patterns for process discovery. Specifically, we consider sequence, exclusive choice (split/join), structured loops, concurrency (split/join), critical section and inclusive choice (split/join), and *unobservable* paths for skipping [24]. The last three patterns have not been studied before systematically in process mining.

In Sect. 2, we expose the problem in more detail against related work and we contribute a *framework* to systematically investigate whether for a specific class C of languages (that are generated by process models having a particular set of operators), we can define an abstraction function α that can abstract any language to a finite representation and is *injective*: $\alpha(L_1) = \alpha(L_2)$ implies $L_1 = L_2$ (for languages L_1 and L_2), or $\alpha(\mathcal{L}(M_1)) = \alpha(\mathcal{L}(M_2))$ implies $\mathcal{L}(M_1) = \mathcal{L}(M_2)$. In these cases, discovering $\gamma(\alpha(L_1)) = \gamma(\alpha(L_2))$ is correct. Central to our proof is the definition of *footprints* that allow to precisely recover behavioural information from an abstraction. As the problem has no general solution, even on regular languages [8], we restrict ourselves to languages that can be described syntactically by the relevant subclass of *block-structured* process models in which each activity is represented at most once (that is, no duplicate activities), which we explain in Sect. 3.

We then use our framework to consolidate previous results on the information-preserving properties and to sharply characterize the limitations of the well-known directly follows abstraction α_{dfg} in Sect. 5. To overcome these limitations, we then propose and study two new abstractions: the *minimum self-distance* abstraction and its footprints discussed in Sect. 6 preserve information about loops in the context of concurrency, and our results close a gap left in earlier works [5]. The *concurrent-optional-or* abstraction discussed in Sect. 7 preserves information about optional behaviour, and we are the first to show that unobservable paths for skipping and inclusive choices are preserved and can be recovered from footprints in finite abstractions of languages, and hence event logs. The operational nature of the footprints allows applying them in process discovery algorithms. We report on experiments showing the practical relevance and applicability of these information-preserving abstractions for process discovery on real-life event logs in Sect. 8. Further, we discuss practical implications of our results on process mining research, mining from streams, and index design for event databases in Sect. 9.

2 Background, problem exposition, and research framework

We first discuss existing abstractions of processes and their applications. We then work out the research problem of information preservation in behavioural abstractions and propose a framework to research this problem systematically.

2.1 Background

Behavioural abstractions of process behaviour have been studied previously, generally with the aim to obtain a finite representation structure for reasoning about and comparing process behaviours. The *directly follows graph* (DFG) or *Transition-adjacency relations* [3,5,25] are derived from behaviour (a language or log) and relate two activities ‘A’ and ‘B’ if and only if ‘A’ can be directly followed by ‘B’; this relation can be enriched with arc weights (based on frequency of observed behaviour) [9] or semantic information from external sources [26]. *Causal footprints* [27] abstract model structures to graphs where multi-edges between activities (visible process steps) represent ‘and’ (concurrent) and ‘xor’ (exclusive choice) pre- and post-dependencies between activities. *Behavioural profiles* (BPs) [8] derived from behaviour or models combine DFGs and causal footprints by providing binary relations for directly-following, concurrent, and conflicting (mutually exclusive) activities. Further, binary relations between activities can be defined to preserve more subtle aspects of concurrency [28]. *Principal transition sequences* abstract behaviour as a truncated execution tree of the model that contains all acyclic executions and truncates repetitions and unbounded behaviour [29].

Most behavioural abstractions have been used to *measure similarity* between process models [30]; BPs [31] and abstraction in [26] preserve more information and allow to define a metric space in order to perform searches in process model collections. In this context also non-behaviour preserving abstractions have been studied, such as projections to “relevant” subsequences [32] and *isotactics* for preserving concurrency of user-defined sets of activities in true-concurrency semantics [33]. BPs have also been used to *propagate changes* in behaviour in process model collections to other related process models [34]. Various sets of binary relations in BPs give rise to implications between relations structured along a lattice, allowing to reason within the abstractions [23,28]. Most process discovery techniques use BPs or DFGs in several variations to abstract and generalize behavioural information from event logs [3–6,35].

Information preservation for BPs has been studied in various ways. van der Aalst et al. [3] characterize the *rediscoverable* models S where from a "complete" log $L \subseteq \mathcal{L}(S)$ the discovered model M describes the same process executions: $\mathcal{L}(M) = \mathcal{L}(S)$. Their proof (1) sufficiently characterizes the rediscoverable model structures that remain distinguished under abstraction α into BPs, and (2) shows that the model synthesis γ step of the alpha-algorithm reconstructs the original model. Badouel et al. [16] also provide necessary conditions. While the alpha-algorithm preserves complex structures including non-free choice constructs, behaviours such as unobservable steps, inclusive choices or interleaving are not preserved, and information is only preserved under the absence of deviations or noise [15]. Furthermore, the characterization and proofs are tied to the alpha-algorithm and cannot be applied to other contexts and discovery algorithms. Independent of an algorithm, BPs are not expressive enough to preserve even trace equivalence for any general class of behaviour (within regular languages) [8]. BPs preserve behaviour for acyclic, *unlabelled* models, i.e. where each activity in the model is observable and distinct from all other activities [8,36]. For cyclic models, only for unlabelled, free-choice workflow nets a variant of BPs including the "up-to-k" successor relation preserves trace equivalence, though k is model specific [7]; for large enough k, this resembles the eventually follows relation.

Information preservation for DFGs has been studied as rediscoverability for the Inductive Miner [5] and the alpha-algorithm [3,16] and its variants. For instance, several variants of the alpha-algorithm have been introduced to include short loops (alpha$^+$ [17]), to include long-distance dependencies [18] (using the eventually follows relation, but only for acyclic processes), to include non-free-choice constructs (alpha^{++} [19]) and to include certain types of silent transitions (alpha$^{\#}$ [20,21], alpha$^{\$}$ [22]). For these algorithms, it was shown that they could obtain enough information from an event log to reliably rediscover certain constructs; however, these characterizations and proofs are tied to the specific algorithms and cannot be applied to other contexts and discovery algorithms. Information preservation in Inductive Miner [5] was proven by showing that behavioural information can be completely retrieved from the abstraction itself, in the form of *footprints* in the DFG, independent of a specific model synthesis algorithm, and characterizing the block-structured models for which this information is preserved in the DFG. However, these footprints allow reusing information abstraction and recovery in other (non-block-structured) contexts such as the Split Miner [6], which uses footprints from the DFG in a different way to synthesize a model, or Projected Conformance Checking [37], which uses abstractions to provide guarantees in approximative fast conformance checking measures. Other process discovery techniques might be able to obtain enough information from event logs to reconstruct a model, such as (conjectured) the ILP miner [35] and several genetic techniques [38,39], but no insights into guarantees and limits of information preservation are available.

2.2 Problem exposition

This paper particularly addresses information-preserving capabilities of abstractions of languages in general, that hold in *any* application using these abstractions, rather than just a single specific algorithm. To approach the problem in an algorithm-independent setting, we generalize prior work on rediscoverability to the following sub-problems:

RQ1 For which class C of languages is each language L *uniquely* defined by its syntactic model M with $L = \mathcal{L}(M)$, i.e. $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ implies $M_1 = M_2$?

RQ2 For which class C of languages does an abstraction α distinguish languages on their abstractions alone, that is, when does $\alpha(L_1) = \alpha(L_2)$ imply $L_1 = L_2$ for $L_1, L_2 \in$ C?

RQ3 For which class C of languages does α preserve the information about the entire model syntax, i.e. when does $\alpha(\mathcal{L}(M_1)) = \alpha(\mathcal{L}(M_2))$ imply $M_1 = M_2$?

RQ4 For which classes C of languages and which abstractions α can syntactic constructs of model M be recognized in $\alpha(\mathcal{L}(M))$?

The specific objective of RQ4 is to not only ensure that an abstraction α can distinguish different types of behaviour, but that information about the behaviour can be *recovered* from the abstraction, so the abstraction can be exploited by different process discovery algorithms.

2.3 Research framework

As discussed in Sect. 1, we will answer the above questions for three different abstractions: directly follows α_{dfg}, minimum self-distance α_{msd}, and concurrent-optional-or α_{coo}. Each abstraction function α_x returns a finite abstraction $\alpha_x(L)$ for any language L in the form of *finite graphs* over the alphabet Σ of L. As RQ2 has no general answer [8], we limit our model class to a very general class of block-structured models $C_{\mathcal{B}}$ defined in Sect. 3. We then employ the following framework to answer RQ1–RQ4 for each abstraction:

– Using a system of sound rewriting rules on block-structured process models [40, p. 113] explained in Sect. 4, we obtain a unique syntactic normal form $NF(M)$ for each model $M \in C_{\mathcal{B}}$.
– We show that each abstraction $\alpha_x(L)$ of *any* language L of some model $M \in C_{\mathcal{B}}$, $L = \mathcal{L}(M)$ preserves not only the behaviour but also *syntax* of M in an implicit form: we can recover from $\alpha_x(L)$ the top-level operator of the normal form $NF(M)$ through specific characteristic features in $\alpha_x(L)$, called *footprints*, answering RQ4. Specifically, much of the information preserved in BPs is already contained in the more basic α_{dfg} from which not only sequence, choice, loops and concurrency but also interleaving (Sect. 5) can be recovered. The *minimum self-distance* abstraction α_{msd} preserves information about loops in the context of concurrency (Sect. 6) allowing to preserve information from a larger model class. The new *concurrent-optional-or* abstraction α_{coo} (Sect. 7) preserves information about unobservable skipping and inclusive choices not considered before.
– We then characterize for each abstraction α_x the class of models $C_x \subset C_{\mathcal{B}}$ for which the *entire* recursive structure of $M \in C_x$ is *uniquely* preserved in the abstraction of its language $\alpha_x(\mathcal{L}(M))$; each C_x partially constrains the nesting of operators, most notably concurrency.
– As our core result, we then show that the languages $\mathcal{L}(M_1)$, $\mathcal{L}(M_2)$ of any two models $M_1, M_2 \in C_x$ with different normal forms $NF(M_1) \neq NF(M_2)$ can always be distinguished in their finite abstractions $\alpha_x(\mathcal{L}(M_1)) \neq \alpha_x(\mathcal{L}(M_2))$ based on the footprints in α_x. Our characterization is sharp in the sense that for any necessary condition, we provide a counterexample, answering RQ3.
– As the normal form $NF(M)$ is unique, we can then conclude RQ2: any two different *languages* $L_1 \neq L_2$ of any two models $L_i = \mathcal{L}(M_i)$, $M_i \in C_x$ can always be distinguished by their finite abstractions $\alpha_x(L_1) \neq \alpha_x(L_2)$.
– We finally can answer RQ1: by RQ3 two different syntactic normal forms $NF(M_1) \neq NF(M_2)$ have different abstractions of their languages $\alpha_x(L_1) \neq \alpha_x(L_2), L_i =$

$\mathcal{L}(NF(M_i))$ and have therefore different languages $L_1 \neq L_2 \in C_x$ (as α_x is a deterministic function).

3 Preliminaries

3.1 Traces, languages, partitions

A *trace* is a sequence of *events*, which are executions of activities (i.e. the process steps). The empty trace is denoted with ϵ.

A *language* is a set of traces, and an *event log* is a finite set of traces. For instance, the language $L_3 = \{\langle a, b, c\rangle, \langle a, c, b\rangle\}$ consists of two traces. For the first trace, first activity a was executed, followed by b and c. Traces can be concatenated using $\cdot$.

In this paper, we limit ourselves to *regular languages*, that is, languages that can be defined using the regular expression patterns sequence, exclusive choice and Kleene star.

We refer to the activities that appear in a language or event log as the *alphabet* Σ of the language or event log. A *partition* of an alphabet consists of several sets, such that each activity of the alphabet appears in precisely one of the sets, and each activity in the sets appears in the alphabet. For instance, the alphabet of L_3 is $\{a, b, c\}$ and a partition of this alphabet is $\{\{a, b\}, \{c\}\}$.

3.2 Block-structured models

A *block-structured* process model can be recursively broken up into smaller pieces. These smaller pieces form a hierarchy of model constructs, that is, a tree [41].

A *process tree* is an abstract representation of a block-structured workflow net [1]. The leaves are either unlabelled or labelled with activities, and all other nodes are labelled with process tree operators. A process tree describes a language: the leaves describe singleton or empty languages, and an operator describes how the languages of its subtrees are to be combined. We formally define process trees recursively:

Definition 3.1 (*process tree syntax*) Let Σ be an alphabet of activities, then

– *activity* $a \in \Sigma$ is a process tree;
– the *silent activity* τ ($\tau \notin \Sigma$) is a process tree;
– let $M_1 \dots M_n$ with $n > 0$ be process trees and let $\oplus$ be a process tree *operator*; then, the *operator node* $\oplus(M_1, \dots M_n)$ is a process tree. We also write this as

$$\begin{array}{c} \oplus \\ \diagup \quad \diagdown \\ M_1 \ \dots \ M_n \end{array} .$$

Each process tree describes a language: an activity describes the execution of that activity, a silent activity describes the empty trace, while an operator node describes a combination of the languages of its children. Each operator combines the languages of its children in a specific way.

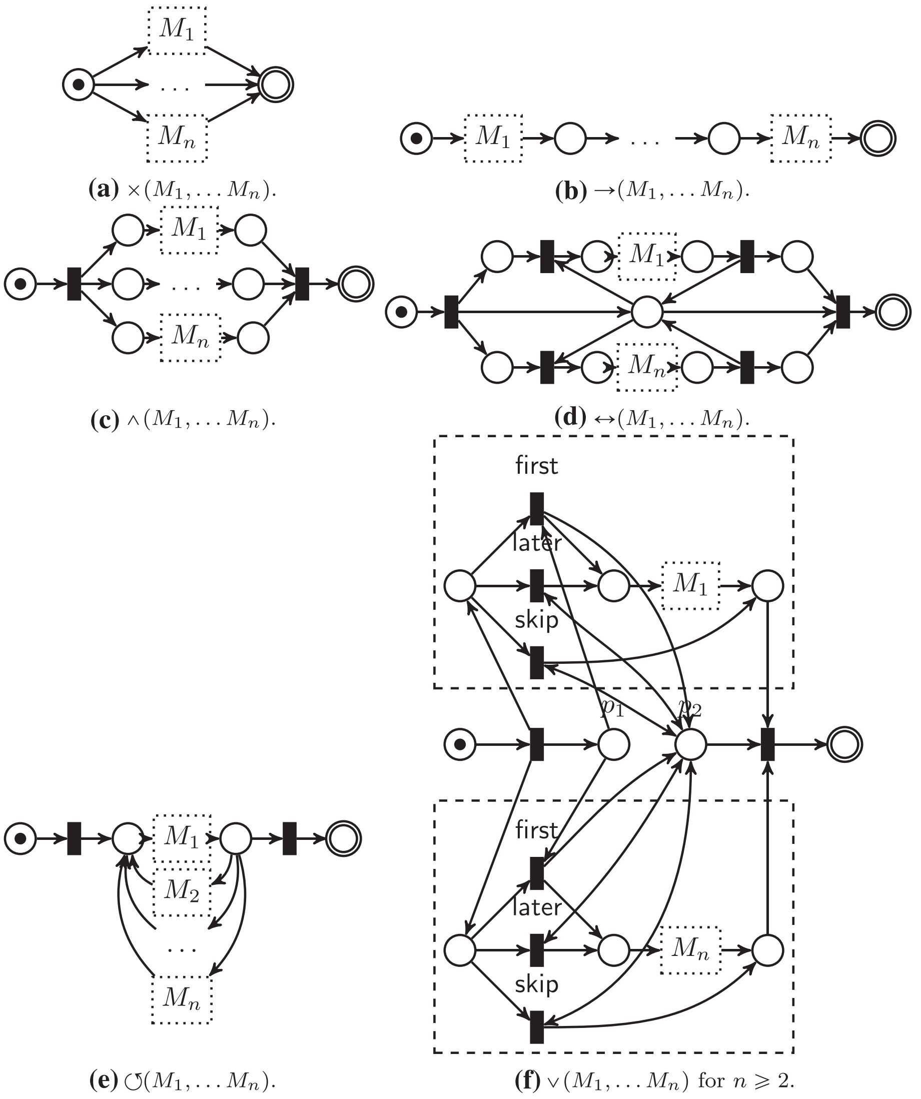
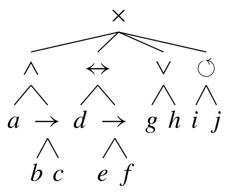

M_1
M_n
...
M_2
first
later
skip
p_1
p_2
(a) $\times(M_1, \ldots M_n)$.
(b) $\rightarrow(M_1, \ldots M_n)$.
(c) $\wedge(M_1, \ldots M_n)$.
(d) $\leftrightarrow(M_1, \ldots M_n)$.
(e) $\circlearrowleft(M_1, \ldots M_n)$.
(f) $\vee(M_1, \ldots M_n)$ for $n \geqslant 2$.

Fig. 3 Recursive Petri-net translations of process trees

In this paper, we consider six process tree operators:

1. exclusive choice $\times$,
2. sequence $\rightarrow$,
3. interleaved (i.e. not overlapping in time) $\leftrightarrow$,
4. concurrency (i.e. possibly overlapping in time) $\wedge$,
5. inclusive choice $\vee$ and
6. structured loop $\circlearrowleft$ (i.e. the loop body M_1 and alternative loop back paths $M_2 \ldots M_n$).

For instance, the language of



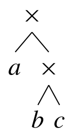
is $\{\langle a,b,c\rangle$, $\langle b,a,c\rangle$, $\langle b,c,a\rangle$, $\langle d,e,f\rangle$, $\langle e,f,d\rangle$, $\langle g\rangle$, $\langle h\rangle$, $\langle g,h\rangle$, $\langle h,g\rangle$, $\langle i\rangle$, $\langle i,j,i\rangle$, $\langle i,j,i,j,i\rangle \ldots\}$. A formal definition of these operators is given in "Appendix A".

Each of the process tree operators can be translated to a sound block-structured workflow net, and to, for instance, a BPMN model [11]. Figure 3 provides some intuition by giving the translation of the process tree operators to (partial) Petri nets [42].

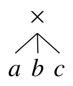
We refer to a process tree that is part of a larger process tree as a *subtree* of the larger tree.

4 A canonical normal form for process trees

Some process trees are structurally different but nevertheless express the same language. For instance, the process trees



and



have a different structure but the same language, consisting of the choice between a, b and c. Such trees are indistinguishable by their language, and thus also indistinguishable by any language abstraction. To take such structural differences out of the equation, we use a set of rules that, when applied, preserve the language of a tree while reducing the structural size and complexity of the tree. It does not matter which rules or in which order the rules are applied: applying the rules exhaustively always yields the same canonical normal form.

The set of rules consist of four types of rules: (1) a singularity rule, which removes operators with only a single child: $\oplus(M) \Rightarrow M$ for any process tree M and operator $\oplus$ (except $\circlearrowleft$); (2) associativity rules, which remove nested operators of the same type, such as $\circlearrowleft(\circlearrowleft(M, \ldots_1), \ldots_2) \Rightarrow \circlearrowleft(M, \ldots_1, \ldots_2)$; (3) τ-reduction rules, which remove superfluous τs, such as $\wedge(\ldots, M, \tau) \Rightarrow \wedge(\ldots, M)$; and (4) rules that establish the relation between concurrency and other operators, such as $\leftrightarrow(M_1, \ldots M_n) \Rightarrow \wedge(M_1, \ldots M_n)$ with $\forall_{1 \leqslant i \leqslant n, t \in \mathcal{L}(M_i)} |t| \leqslant 1$.

The set of rules is terminating, that is, to any process tree only a finite number of rules can be applied sequentially; the set is locally confluent, that is, if two rules are applicable to a single tree, yielding trees A and B, then it is possible to reduce A and B to a common tree C; and by extension, the set of rules is confluent and canonical. For more details, please refer to [40, p. 118].

As a result, the rules are canonical, so in the remainder of this paper, we only need to consider process trees in canonical normal form, to which we refer as reduced process trees; a *reduced process tree* is a process to which the reduction rules have been applied exhaustively. This allows us to reason based on the structure of reduced trees, rather than on the behaviour of arbitrarily structured trees.

5 Preservation and recovery with directly follows graphs

In order to study the information preserved in abstractions, in this section we study the first abstraction: directly follows graphs. This abstraction has been well studied in context of process discovery. For instance, in the context of the alpha-algorithm, the directly follows abstraction has been shown to preserve information about a certain class of Petri nets. In this section, we show that directly follows graphs preserve even more information than previously known.

We first formally introduce the abstraction, after which we introduce characteristics of the recursive process tree operators in the abstraction: *footprints*. Using these footprints, we show that every two reduced process trees that are structurally different have different abstractions. That is, we show that if two trees have the same directly follows abstraction, then there cannot be a structural difference between the two trees. For subsequent abstractions in this paper, which get more involved, we will use a similar proof strategy.

We first introduce the abstraction. Second, we introduce the footprints of the recursive process tree operators in this abstraction. Third, we introduce a class of recursively defined languages and show that this class is tight, that is, structurally different trees outside this class might yield the same abstraction. Finally, we show that different normal forms within the class have different abstractions (and hence languages), thereby using the research framework of Sect. 2.

5.1 Directly follows graphs

A directly follows graph abstracts from a language by only expressing which activities (the nodes of the graph) are followed directly by which activities in a trace in the language. Furthermore, a directly follows graph expresses the activities with which traces start or end in the language, and whether the language contains the empty trace.

Definition 5.1 (*directly follows graph*) Let Σ be an alphabet and let L be a language over Σ such that $\epsilon \notin \Sigma$. Then, the *directly follows graph* α_{dfg} of L consists of:

$$\begin{aligned}
\alpha_{\text{dfg}}(a, b) &\Leftrightarrow \exists_{t \in L}\ t = \langle \ldots a, b, \ldots \rangle \\
a \in \text{Start}(\alpha_{\text{dfg}}) &\Leftrightarrow \exists_{t \in L}\ t = \langle a, \ldots \rangle \\
a \in \text{End}(\alpha_{\text{dfg}}) &\Leftrightarrow \exists_{t \in L}\ t = \langle \ldots, a \rangle \\
\epsilon \in \alpha_{\text{dfg}} &\Leftrightarrow \exists_{t \in L}\ t = \epsilon
\end{aligned}$$

For a directly follows graph α_{dfg}, $\Sigma(\alpha_{\text{dfg}})$ denotes the alphabet of activities, i.e. the nodes of α_{dfg}.

For instance, let $L_4 = \{\langle a, b, c \rangle, \langle a, c, b \rangle, \epsilon\}$ be a language. Then, the directly follows graph of L_4 consists of the edges $\alpha_{\text{dfg}}(a, b), \alpha_{\text{dfg}}(b, c), \alpha_{\text{dfg}}(a, c), \alpha_{\text{dfg}}(c, b)$. The start activity is a, the end activities are b and c and the directly follows graph contains ϵ. A graphic representation of this directly follows graph is shown in Fig. 4.

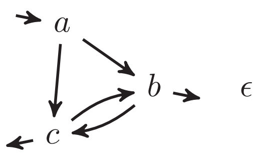


Fig. 4 An example of a directly follows graph: $\alpha_{\mathrm{dfg}}(L_4)$

5.2 Footprints

Each process tree operator has a particular characteristic in a directly follows graph, a *footprint*. We will use footprints in our uniqueness proofs to show differences between trees: if one tree contains a particular footprint and another one does not, the languages of these trees must be different. Furthermore, process discovery algorithms can use footprints to infer syntactic constructs that capture the abstracted behaviour.

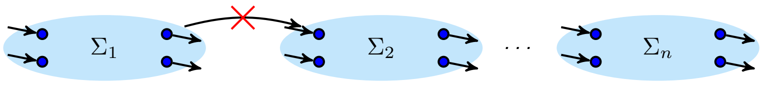
In this section, we introduce these characteristics and show that process tree operators exhibit them.

A *cut* is a tuple $(\oplus, \Sigma_1, \ldots \Sigma_m)$, such that $\oplus$ is a process tree operator and $\Sigma_1 \ldots \Sigma_m$ are sets of activities. A cut is *non-trivial* if $m > 1$ and no Σ_i is empty.

Definition 5.2 (*directly follows footprints*) Let α_{dfg} be a directly follows relation and let $c = (\oplus, \Sigma_1, \ldots \Sigma_n)$ be a cut, consisting of a process tree operator $\oplus \in \{\times, \rightarrow, \leftrightarrow, \wedge, \circlearrowleft\}$ and a partition of activities with parts $\Sigma_1 \ldots \Sigma_n$ such that $\Sigma(\alpha_{\mathrm{dfg}}) = \bigcup_{1 \leqslant i \leqslant n} \Sigma_i$ and $\forall_{1 \leqslant i < j \leqslant n}\ \Sigma_i \cap \Sigma_j = \emptyset$.

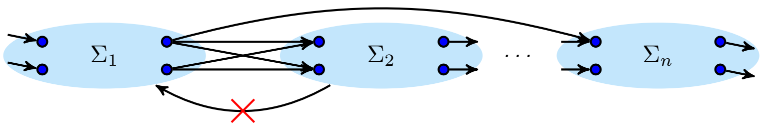
– Exclusive choice. c is an *exclusive choice cut* in α_{dfg} if $\oplus = \times$ and

x.1 No part is connected to any other part:



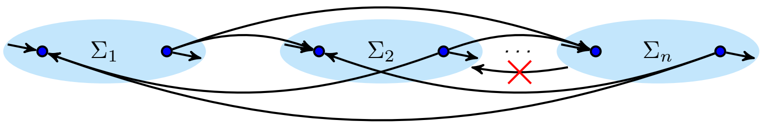
– Sequential. c is a *sequence cut* in α_{dfg} if $\oplus = \rightarrow$ and

s.1 Each node in a part is indirectly and only connected to all nodes in the parts "after" it:



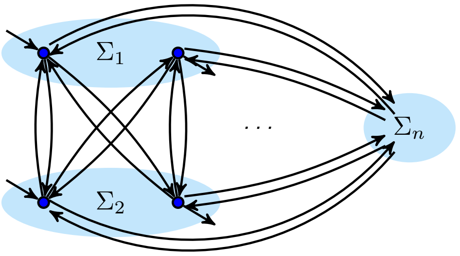
– Interleaved. c is an *interleaved cut* in α_{dfg} if $\oplus = \leftrightarrow$ and

i.1 Between parts, all and only connections exist from an end to a start activity:



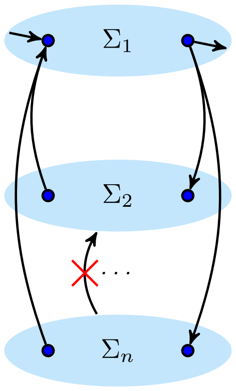
- Concurrent. c is a *concurrent cut* in α_{dfg} if $\oplus = \wedge$ and

  c.1 Each part contains a start and an end activity.
  c.2 All parts are fully interconnected.



- Loop. c is a *loop cut* in α_{dfg} if $\oplus = \circlearrowleft$ and

  l.1 All start and end activities are in the body (i.e. the first) part.
  l.2 Only start/end activities in the body part have connections from/to other parts.
  l.3 Redo parts have no connections to other redo parts.
  l.4 If an activity from a redo part has a connection to/from the body part, then it has connections to/from all start/end activities.



A formal definition is included in "Appendix B.1".

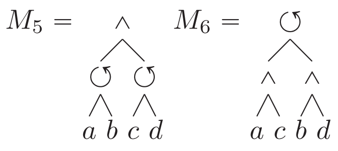
For instance, consider the directly follows graph of L_4 shown in Fig. 4: in this graph, the cut $(\rightarrow, \{a\}, \{b, c\})$ is a $\rightarrow$-cut.

Inspecting the semantics of the process tree operators (Definition A.1), it follows that in the directly follows graphs of process trees, these footprints are indeed present:

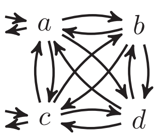
Lemma 5.1 (Directly follows footprints) *Let $M = \oplus(M_1, \ldots M_m)$ be a process tree without duplicate activities, with $\oplus \in \{\times, \rightarrow, \leftrightarrow, \wedge, \circlearrowleft\}$. Then, the cut $(\oplus, \Sigma(M_1), \ldots \Sigma(M_n))$ is a $\oplus$-cut in $\alpha_{dfg}(M)$ according to Definition 5.2.*

5.3 A class of trees: C_{dfg}

Not all process trees can be distinguished using the introduced footprints. In this section, we describe the class of trees that can be distinguished by directly follows graphs: C_{dfg}. To



(a) Language-different process trees.



(b) $\alpha_{\text{dfg}}(M_5) = \alpha_{\text{dfg}}(M_6)$

Fig. 5 An example for the necessity of Requirement C_{dfg}.l.1

illustrate that this class is tight, we give counterexamples of pairs of trees outside of C_{dfg} and show that these trees cannot be distinguished using their directly follows graphs.

Definition 5.3 (Class C_{dfg}) Let Σ be an alphabet of activities. Then, the following process trees are in C_{dfg}:

- a with $a \in \Sigma$ is in C_{dfg}
- Let $M_1 \dots M_n$ be reduced process trees in C_{dfg} without duplicate activities: $\forall_{i \in [1 \dots n], i \neq j \in [1 \dots n]}\ \Sigma(M_i) \cap \Sigma(M_j) = \emptyset$. Then,

  - $\oplus(M_1, \dots M_n)$ with $\oplus \in \{\times, \rightarrow, \wedge\}$ is in C_{dfg};
  - $\leftrightarrow (M_1, \dots M_n)$ is in C_{dfg} if all:

    i.1 At least one child has disjoint start and end activities: $\exists_{i \in [1 \dots n]}\ \text{Start}(M_i) \cap \text{End}(M_i) = \emptyset$;
    i.2 No child is interleaved itself: $\forall_{i \in [1 \dots n]}\ M_i \neq \leftrightarrow (\dots)$;
    i.3 Each concurrent child has at least one child with disjoint start and end activities: $\forall_{i \in [1 \dots n] \wedge M_i = \wedge(M_{i_1}, \dots M_{i_m})}\ \exists_{j \in [1 \dots m]}\ \text{Start}(M_{i_j}) \cap \text{End}(M_{i_j}) = \emptyset$

  - $\circlearrowleft (M_1, \dots M_n)$ is in C_{dfg} if:

    l.1 the first child has disjoint start and end activities: $\text{Start}(M_1) \cap \text{End}(M_1) = \emptyset$.

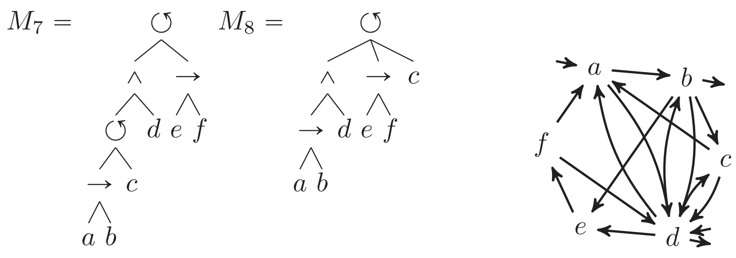
These requirements are tight as for each requirement there exists counterexamples of the following form: two process trees whose syntax violates the requirement and who have different languages but have identical directly follows graphs. Consequently, the two trees cannot be distinguished by the directly follows abstraction.

Figure 5 illustrates the counterexample for C_{dfg}.l.1. This requirement has some similarity with the so-called short loops of the α algorithm [1].

The counterexamples for C_{dfg}.i.1, C_{dfg}.i.2 and C_{dfg}.i.3 follow a similar reasoning and are given in "Appendix C.1".

Trees with τ and $\vee$ nodes are excluded from C_{dfg} entirely. The corresponding counterexamples for the directly follows abstraction and how to preserve information on τ and $\vee$ will both be discussed in Sect. 7.

A weaker requirement one could consider to replace C_{dfg}.l.1is that at least one child of the loop should have disjoint start and end activities: $M' = \circlearrowleft (M'_1, \dots M'_n) : \exists_{1 \leqslant i \leqslant n}\ \text{Start}(M'_i) \cap \text{End}(M'_i) = \emptyset$. This weaker requirement would allow both M_7 and M_8 (see Fig. 6), i.e. c can be in the redo of the inner or outer $\circlearrowleft$-nodes. However, the counterexample in Fig. 6 shows that this weaker constraint is not strong enough.


$M_7 =$
$M_8 =$
$\circlearrowleft$ $\wedge$ $\rightarrow$ d e f c a b

(a) Language-different process trees. **(b)** $\alpha_{\mathrm{dfg}}(M_7) = \alpha_{\mathrm{dfg}}(M_8)$

Fig. 6 A counterexample that a weaker constraint, which would state that at least one child of a $\circlearrowleft$ node should have disjoint start and end activities, would not suffice to replace Requirement $\mathrm{C}_{\mathrm{dfg}}$.l.1

5.4 Uniqueness

Now that all concepts for abstraction and recovery have been introduced, what is left is to prove for α_{dfg} which behaviour can be preserved. Here, we provide the lemmas and theorems according to the framework of Sect. 2 to establish the fundamental formal limits. This proof structure will be reused on the more complex abstractions later.

That is, in this section, we prove that two structurally different reduced process trees of $\mathrm{C}_{\mathrm{dfg}}$ always have different directly follows graphs. To prove this, we exploit the recursive structure of process trees: we first show that if the root operator of two process trees differs, then their directly follows graphs differ. Second, we show that if the activity partition (that is, the distribution of activities over the children of the root) of two trees differs, then their directly follows graphs differ. These two results yield that any structural difference over the entire tree will result in a different directly follows graph.

Lemma 5.2 (Operators are mutually exclusive) *Take two reduced process trees of* C_{dfg} $K = \oplus(K_1, \ldots K_n)$ *and* $M = \otimes(M_1, \ldots M_m)$ *such that* $\oplus \neq \otimes$*. Then,* $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$*.*

This lemma is proven by showing that for each pair of operators, there is always a difference in their directly follows graphs. For instance, the directly follows graph of a $\times$ node is not connected, while all other operators make the graph connected. For a detailed proof, please refer to "Appendix D.1".

Lemma 5.3 (Partitions are mutually exclusive) *Take two reduced process trees of* C_{dfg} $K = \oplus(K_1 \ldots K_n)$ *and* $M = \oplus(M_1 \ldots M_m)$ *such that their activity partition is different:* $\exists_{1 \leqslant w \leqslant min(n,m)}\ \Sigma(K_w) \neq \Sigma(M_w)$*. Then,* $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$*.*

For this proof, we assume a fixed order of children for the non-commutative operators $\rightarrow$ and $\circlearrowleft$. Then, we show, for each operator, that a difference in activity partitions leads to a difference in directly follows graphs. For a detailed proof, please refer to "Appendix D.2".

Lemma 5.4 (Abstraction uniqueness for $\mathrm{C}_{\mathrm{dfg}}$) *Take two reduced process trees of* C_{dfg} *K and M such that* $K \neq M$*. Then,* $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$*.*

Proof Towards contradiction, assume that there exist two reduced process trees K and M, both of $\mathrm{C}_{\mathrm{dfg}}$, such that $\alpha_{\mathrm{dfg}}(K) = \alpha_{\mathrm{dfg}}(M)$, but $K \neq M$. Then, there exist topmost subtrees K' in K and M' in M such that $\alpha_{\mathrm{dfg}}(K') = \alpha_{\mathrm{dfg}}(M')$ and such that K', M' are structurally different in their activity, operator or activity partition, i.e. either

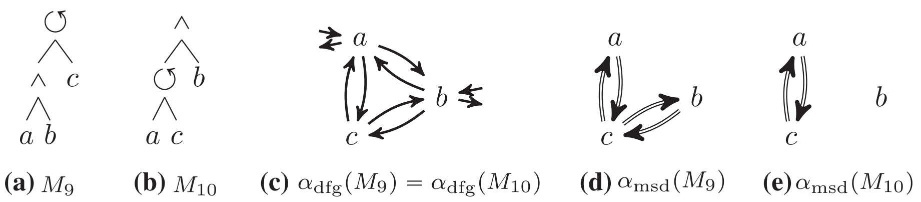

a
b
c
a
c
b
a
b
c
a
b
c
a
b
c
(a) M_9
(b) M_{10}
(c) $\alpha_{\mathrm{dfg}}(M_9) = \alpha_{\mathrm{dfg}}(M_{10})$
(d) $\alpha_{\mathrm{msd}}(M_9)$
(e) $\alpha_{\mathrm{msd}}(M_{10})$

Fig. 7 Example of two process trees outside of C_{dfg} that have the same directly follows graph. In this section, we introduce the new minimum self-distance abstraction (α_{msd}) and corresponding footprints that distinguish these trees

- K' or M' is a τ, while the other is not. Then, obviously $\alpha_{\mathrm{dfg}}(K') \neq \alpha_{\mathrm{dfg}}(M')$.
- K' or M' is a single activity while the other is not. Then, by the restrictions of C_{dfg}, $\alpha_{\mathrm{dfg}}(K') \neq \alpha_{\mathrm{dfg}}(M')$.
- $K' = \otimes(K'_1 \dots K'_n)$ and $M' = \oplus(M'_1 \dots M'_n)$ such that $\oplus \neq \otimes$. By Lemma 5.2, $\alpha_{\mathrm{dfg}}(K') \neq \alpha_{\mathrm{dfg}}(M')$.
- $K' = \oplus(K'_1 \dots K'_n)$ and $M' = \oplus(M'_1 \dots M'_n)$ such that the activity partition is different, i.e. there is an i such that $\Sigma(K'_i) \neq \Sigma(M'_i)$. By Lemma 5.3, $\alpha_{\mathrm{dfg}}(K') \neq \alpha_{\mathrm{dfg}}(M')$.

Hence, there cannot exist such K and M, and therefore, $\alpha_{\mathrm{dfg}}(K) \neq \alpha_{\mathrm{dfg}}(M)$. □

As a language abstraction is solely derived from a language, different directly follows graphs imply different languages (the reverse not necessarily holds), we immediately conclude uniqueness:

Corollary 5.1 (Language uniqueness for C_{dfg}) *There are no two different reduced process trees of C_{dfg} with equal languages. Hence, for trees of class C_{dfg}, the normal form of Sect. 4 is uniquely defined.*

6 Preservation and recovery with minimum self-distance

A structure that is difficult to distinguish in directly follows graphs is the nesting of $\circlearrowleft$ and $\wedge$ operators, resembling the so-called short loops of the α algorithm. For instance, Fig. 7 shows an example of two process trees with such a nesting, having the same directly follows graph. In C_{dfg}, such nestings were excluded.

In this section, we introduce a new abstraction, the *minimum self-distance* abstraction, that distinguishes some of such nestings by considering which activities must be in between two executions of the same activity. As illustrated in Fig. 7, the minimum self-distance abstraction captures more information of process trees, enabling discovery and conformance checking algorithms to distinguish the languages of these trees.

The remainder of this section follows a strategy similar to Sect. 5: first, we introduce the language abstraction of minimum self-distance graphs. Second, we introduce adapted footprints of the recursive process tree operators in minimum self-distance graphs. Third, we characterize the extended class of recursively defined languages and show where this class is tight (conditions cannot be dropped). Fourth, we show that different trees in normal form of this class have different combinations of directly follows graphs and minimum self-distance graphs (and hence languages), using the research framework of Sect. 2.

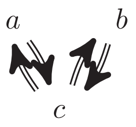


Fig. 8 Minimum self-distance graph of L_{11}

6.1 Minimum self-distance

The *minimum self-distance m* of an activity is the minimum number of events in between two executions of that activity:

Definition 6.1 (*Minimum self-distance*) Let L be a language, and let $a, b \in \Sigma(L)$. Then,

$$m(a) = \begin{cases} \min_{\langle \ldots, a, \ldots_1, a, \ldots\rangle \in L} |\ldots_1| & \text{if } \exists_{\langle \ldots, a, \ldots_1, a, \ldots\rangle \in L} \\ \infty & \text{otherwise} \end{cases}$$

A *minimum self-distance graph* α_{msd} is a directed graph whose nodes are the activities of the alphabet Σ. An edge $\alpha_{\text{msd}}(a, b)$ in a minimum self-distance graph denotes that b is a *witness* of the minimum self-distance of a, i.e. activity b can appear in between two minimum-distant executions of activity a:

Definition 6.2 (*Minimum self-distance graph*) Let L be a language, and let $a, b \in \Sigma(L)$. Then,

$$\alpha_{\text{msd}}(a, b) \equiv \exists_{\langle \ldots, a, \ldots_1, a, \ldots\rangle \in L}\ b \in \ldots_1 \wedge |\ldots_1| = m(a)$$

For a minimum self-distance graph α_{msd}, $\Sigma(\alpha_{\text{msd}})$ denotes the alphabet of activities, i.e. the nodes of α_{msd}.

For instance, consider the log $L_{11} = \{\langle a, b\rangle, \langle b, a, c, a, b\rangle, \langle a, b, c, a, b, c, b, a\rangle\}$. In this log, the minimum self-distances are $m(a) = 1$, $m(b) = 1$ and $m(c) = 2$, witnessed by the subtraces $\langle a, c, a\rangle$ (for a), $\langle b, c, b\rangle$ (for b) and $\langle c, a, b, c\rangle$ (for c). Thus, the minimum self-distance relations are $\alpha_{\text{msd}}(a, c)$, $\alpha_{\text{msd}}(b, c)$, $\alpha_{\text{msd}}(c, a)$ and $\alpha_{\text{msd}}(c, b)$. Figure 8 visualizes this minimum self-distance relation as a graph.

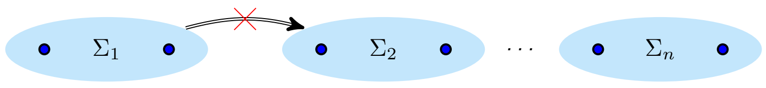
6.2 Footprints

In this section, we introduce the characteristics that process tree operators leave in minimum self-distance graphs.

Definition 6.3 (*minimum self-distance footprints*) Let α_{msd} be a minimum self-distance graph and let $c = (\oplus, \Sigma_1, \ldots \Sigma_n)$ be a cut, consisting of a process tree operator $\oplus \in \{\times, \rightarrow, \leftrightarrow, \wedge, \circlearrowleft\}$ and a partition of activities with parts $\Sigma_1 \ldots \Sigma_n$ such that $\Sigma(\alpha_{\text{msd}}) = \bigcup_{1 \leqslant i \leqslant n} \Sigma_i$ and $\forall_{1 \leqslant i < j \leqslant n}\ \Sigma_i \cap \Sigma_j = \emptyset$.

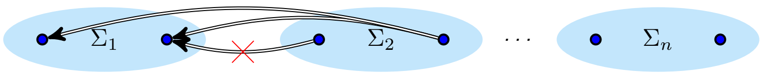
– Concurrent and interleaved. If $\oplus = \wedge$ or $\oplus = \leftrightarrow$, then in α_{msd}:

ci.1 There are no α_{msd} connections between parts:

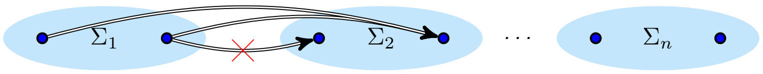


– Loop. If $\oplus = \circlearrowleft$ then in α_{msd}:

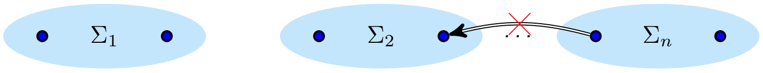
1.1 Each activity has an outgoing edge.
1.2 All redo activities that have a connection to a body activity have connections to the same body activities:



1.3 All body activities that have a connection to a redo activity have connections to the same redo activities:



1.4 No two activities from different redo children have an α_{msd}-connection:



A formal definition is included in "Appendix B.2".

For instance, consider the example shown in Fig. 7: the two process trees M_9 and M_{10} have the same directly follows graph (Fig. 7c). However, the minimum self-distance graphs of these two trees differ (Fig. 7d, e). To exploit this difference, the footprints in both graphs differ: in $\alpha_{\text{msd}}(M_9)$, the only footprint is $(\circlearrowleft, \{a, b\}, \{c\})$, while in $\alpha_{\text{msd}}(M_9)$ the only footprint is $(\wedge, \{a, c\}, \{b\})$ footprint.

From the semantics of process trees, it follows that these footprints are present in minimum self-distance graphs of process trees without duplicate activities:

Lemma 6.1 (Minimum self-distance footprints) *Let $M = \oplus(M_1, \dots M_m)$ be a process tree without duplicate activities, with $\oplus \in \{\times, \rightarrow, \leftrightarrow, \wedge, \circlearrowleft\}$. Then, the cut $(\oplus, \Sigma(M_1), \dots \Sigma(M_n))$ is a $\oplus$-cut in $\alpha_{msd}(M)$ according to Definition 6.3.*

6.3 A class of trees: C_{msd}

Using the minimum self-distance graph, we can lift a restriction of C_{dfg} partially. In this section, we introduce the extended class of process trees C_{msd}, in which a $\circlearrowleft$-node can be a direct child of a $\wedge$-node.

Definition 6.4 (*Class* C_{msd}) Let Σ be an alphabet of activities. Then, the following process trees are in C_{msd}:

– $M \in \mathrm{C}_{\text{dfg}}$ is in C_{msd};
– Let $M_1 \dots M_n$ be reduced process trees in C_{msd} without duplicate activities: $\forall_{i \in [1 \dots n], i \neq j \in [1 \dots n]}\ \Sigma(M_i) \cap \Sigma(M_j) = \emptyset$. Then, $\circlearrowleft(M_1, \dots M_n)$ is in C_{msd} if:

1.1 the body child is not concurrent:
$M_1 \neq \wedge(\dots)$

We neither have a proof nor a counterexample for the necessity of Requirement C_{msd}.1.1. Figure 9 illustrates this: the two trees have a different language, an equivalent α_{dfg}-graph

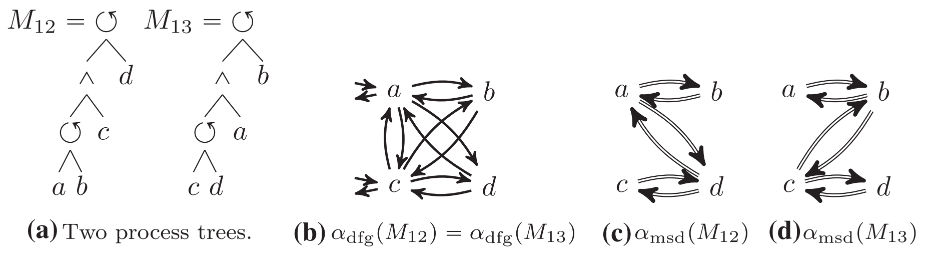

$M_{12} = \circlearrowleft$
$M_{13} = \circlearrowleft$
(a) Two process trees.
(b) $\alpha_{\mathrm{dfg}}(M_{12}) = \alpha_{\mathrm{dfg}}(M_{13})$
(c) $\alpha_{\mathrm{msd}}(M_{12})$
(d) $\alpha_{\mathrm{msd}}(M_{13})$

Fig. 9 A counterexample for uniqueness using α_{msd} footprints: M_{12} and M_{13} have a different language and α_{msd}-graph, but this does not become apparent in the α_{msd}-footprint

(shown in Fig. 9b) but a different α_{msd}-graph (shown in Fig 9c, d). Thus, they could be distinguished using their α_{msd}-graph. However, the footprint (Definition 6.3) cannot distinguish these trees: both cuts $(\circlearrowleft, \{a, b, c\}, \{d\})$ and $(\circlearrowleft, \{a, c, d\}, \{b\})$ are valid in both α_{msd}-graphs, where $(\circlearrowleft, \{a, b, c\}, \{d\})$ corresponds to M_{12} and $(\circlearrowleft, \{a, c, d\}, \{b\})$ corresponds to M_{13}. This implies that a discovery algorithm that uses only the footprint cannot distinguish these two trees. In "Appendix E.1", we elaborate on this.

6.4 Uniqueness

To prove that the combination of directly follows and minimum self-distance graphs distinguishes all reduced process trees of $\mathrm{C}_{\mathrm{msd}}$, we follow a strategy that is similar to the one used in Sect. 5.4: we first show that, given two process trees, a difference in root operators implies a difference in either the α_{dfg} or α_{msd}-abstractions of the trees. Second, we show that, given two process trees, a difference in root activity partitions implies a difference in the abstractions. Third, using the first two results, we show that any structural difference between two process trees implies that the trees have different abstractions. For the full proofs, please refer to "Appendix E". Consequently:

Corollary 6.1 (Language uniqueness for $\mathrm{C}_{\mathrm{msd}}$) *There are no two different reduced process trees of* C_{msd} *with equal languages. Hence, for trees of class* C_{msd}*, the normal form of Sect. 4 is uniquely defined.*

7 Preserving and recovering optionality and inclusive choice

The abstractions discussed in previous sections preserve information only if all behaviour is always fully observable. The α_{dfg} and α_{msd} abstractions cannot preserve information about *unobservable behaviour*, or *skips*, caused by the inclusive choice operator $\vee$ or by an alternative silent τ step making behaviour *optional*.

The model M_{15} in Fig. 10b exhibits *optionality* of the sequence b, c due to a silent activity τ under a choice $\times$. Although its α_{dfg}-graph in Fig. 10c differs from the α_{dfg}-graph of M_{14} in Fig. 10a without optionality, the optionality of b, c cannot be *recovered* as both models have the same $\rightarrow$-footprint of Sect. 5.2. Model M_{17} in Fig. 11a exhibits skips due to the inclusive choice operator $\vee$ and $\times(\tau, .)$ under $\wedge$. Yet, its α_{dfg} and α_{msd} abstractions are identical to the α_{dfg} and α_{msd} abstractions (Fig. 11c and d) of the model of Fig. 11b having no skips, making their behaviour indistinguishable under these abstractions.

In this section, we introduce techniques to preserve and recover information about these unobservable types of behaviour. We first discuss the influence of optionality on α_{dfg}-

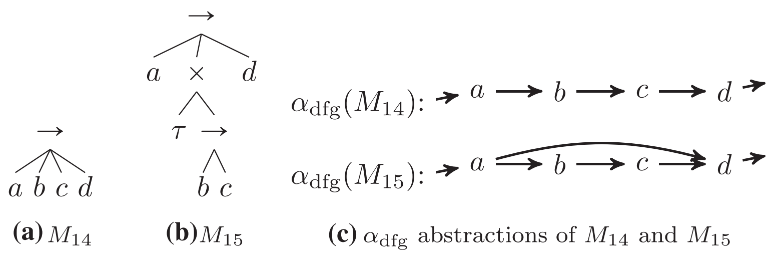

$\alpha_{\mathrm{dfg}}(M_{14})$:
$\alpha_{\mathrm{dfg}}(M_{15})$:
(a) M_{14}
(b) M_{15}
(c) α_{dfg} abstractions of M_{14} and M_{15}

Fig. 10 The process tree M_{15} is outside of C_{msd}. Although they have different abstractions $\alpha_{\mathrm{dfg}}(M_{14}) \neq \alpha_{\mathrm{dfg}}(M_{15})$, their $\rightarrow$-footprints are indistinguishable: $(\rightarrow, \{a\}, \{b\}, \{c\}, \{d\})$

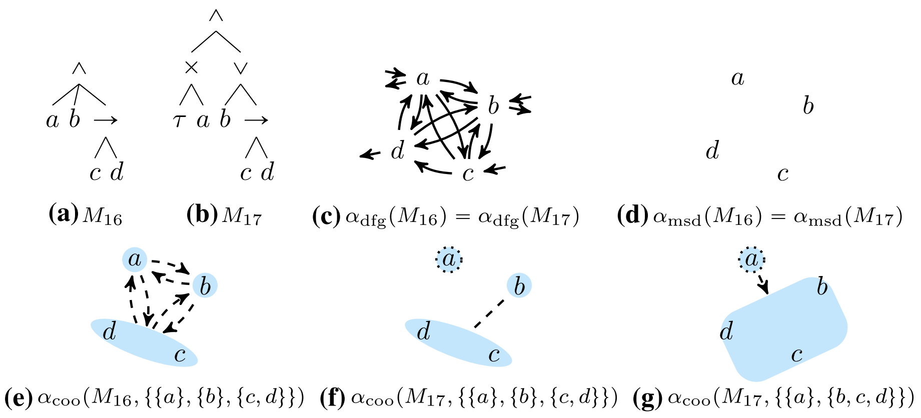

(a) M_{16}
(b) M_{17}
(c) $\alpha_{\mathrm{dfg}}(M_{16}) = \alpha_{\mathrm{dfg}}(M_{17})$
(d) $\alpha_{\mathrm{msd}}(M_{16}) = \alpha_{\mathrm{msd}}(M_{17})$
(e) $\alpha_{\mathrm{coo}}(M_{16}, \{\{a\}, \{b\}, \{c, d\}\})$
(f) $\alpha_{\mathrm{coo}}(M_{17}, \{\{a\}, \{b\}, \{c, d\}\})$
(g) $\alpha_{\mathrm{coo}}(M_{17}, \{\{a\}, \{b, c, d\}\})$

Fig. 11 The two models $M_{16} \in C_{\mathrm{msd}}$ and $M_{17} \notin C_{\mathrm{msd}}$ have different behaviour but the same α_{dfg}-graph (**c**) and the same α_{msd}-graph (**d**). Information about τ and $\vee$ in M_{17} is preserved in the α_{coo}-graphs in (**f**) and (**g**)

footprints and exclude some cases, in particular some types of loops, from further discussion in this paper (Sect. 7.1). We then follow our framework and provide abstractions and footprints to preserve information about optionality and inclusive choices. We show that *partial cuts* (that consider only a subset of the activities in a language) allow to localize skips.

For example, to recover the optionality in $\alpha_{\mathrm{dfg}}(M_{15})$ of Fig. 10c, we find the *partial* $\rightarrow$-*cut footprint* $(\rightarrow, \{b\}, \{c\})$ as a subgraph that adheres to the $\rightarrow$-cut of Sect. 5.2 *and* can be *tightly skipped* by $\alpha_{\mathrm{dfg}}(M_{15})(a, d)$. We formalize this notion and show that it correctly recovers skips in a sequence from α_{dfg} in Sect. 7.2.

Preserving information about $\vee$ requires an entirely new abstraction as the α_{dfg}-graph for $\vee$ and $\wedge$ is identical. We introduce the *concurrent-optional-or (coo,* α_{coo}*) abstraction* which preserves information on a family of graphs "*bottom-up*". For instance, the traces of M_{17} such as $\langle b\rangle$, $\langle a, b\rangle$, $\langle c, d, b\rangle$, $\langle c, b, d, a\rangle$, $\langle c, d\rangle$ and $\langle c, d, a\rangle$ show that if $\{a\}$ occurs there is a similar trace in which $\{a\}$ does not occur ($\{a\}$ is *optional*), and that if any of the sets $\{b\}$ and $\{c, d\}$ occur, then similar traces with any combination of $\{b\}$ and $\{c, d\}$ occur as well (are *interchangeable*), which we encode in the *coo-graph* of Fig. 11f. A pattern for also relating $\{a\}$ emerges by grouping b, c and d: the presence of $\{a\}$ in a trace *implies the occurrence* of $\{b, c, d\}$ in that trace (but not vice versa), which we encode in the coo-graph of Fig. 11g. Both coo-graphs together are the α_{coo}-abstraction of M_{17}. Similarly, Fig. 11e is the α_{coo}-abstraction of 11a. In Sect. 7.3, we further explain and define α_{coo} for languages and we show that $\vee$, $\wedge$, and $\times(\tau, .)$ can be correctly recovered from α_{coo} through corresponding partial cut footprints.

In Sect. 7.4, we characterize the class C_{coo} of behaviour (in terms of process trees) and prove that α_{dfg}, α_{msd}, and α_{coo} abstractions and footprints together uniquely preserve behavioural information in C_{coo}, thereby using the research framework outlined in Sect. 2.

7.1 Optionality

Unobservable skips can be described syntactically by an $\times(\tau, .)$ construct and a $\circlearrowleft(\tau, .)$ construct, making one or more subtrees *optional*. We discuss the influence of optional behaviour on abstractions for these two cases.

The effect of a $\times(\tau, .)$ construct is that its subtree is *optional*, that is, it can be skipped. We call a process tree optional ($\overline{?}$) if its language contains the empty trace:

Definition 7.1 (*optionality*)

$$\overline{?}(M) \equiv \epsilon \in \mathcal{L}(M)$$

Optionality has surprisingly little direct influence on the directly follows graph: in a directly follows graph, optionality shows up as an empty trace (by Definition 5.1).

Lemma 7.1 ($\times(\tau, .)$ footprint) *Let M be a process tree such that $\overline{?}(M)$. Then, $\epsilon \in \alpha_{dfg}(M)$.*

However, optionality may influence the footprints in the α_{dfg}-graph of other behaviour it is embedded in, that is, when an optional subtree is below another operator. The case of $\times(\tau, .)$ under $\times$ can be eliminated through syntactical normal forms; $\times(\tau, .)$ under $\leftrightarrow$ does not influence the footprints in the α_{dfg}-graph [40, p. 150]. The case of $\times(\tau, .)$ under $\rightarrow$ changes the α_{dfg}-graph and requires a new $\rightarrow$-footprint which we introduce in Sect. 7.2.

The cases of $\times(\tau, .)$ under $\circlearrowleft$ and of $\circlearrowleft(\tau, .)$ also change the α_{dfg}-graph but cannot be recovered by existing $\circlearrowleft$-footprints. However, in contrast to $\rightarrow$, we may find two syntactically different models with $\times(\tau, .)$ under $\circlearrowleft$ and $\circlearrowleft(\tau, .)$ which have identical languages, which renders our proof strategy inapplicable as it relies on the assumption that each behaviour (language) has a unique syntactic description, see [40, p. 150] for details. We exclude these cases in the following.

Finally, the α_{dfg}-graph cannot distinguish whether $\times(\tau, .)$ is present under $\wedge$ or not. Likewise, $\vee$ and $\wedge$ yield the same α_{dfg}-graph. Preserving information requires a new abstraction which we introduce in Sect. 7.3.

In [20], five types of silent Petri net transitions were identified: Initialize and Finalize, which start or end a concurrency, are captured by the $\wedge$-operator and considered in this work. Skips, which bypass one or more transitions, are the transitions under study in this section, that is, $\times(\tau, .)$-constructs and $\vee$-constructs, see Fig. 3. Redo, which allows the model to go back and redo transitions, are not studied in this paper, that is, $\circlearrowleft(., \tau)$-constructs. Switch-transitions, which allow jumping between exclusive branches, have no equivalent in block-structured models and are typically mimicked by duplicating activities.

7.2 Sequence

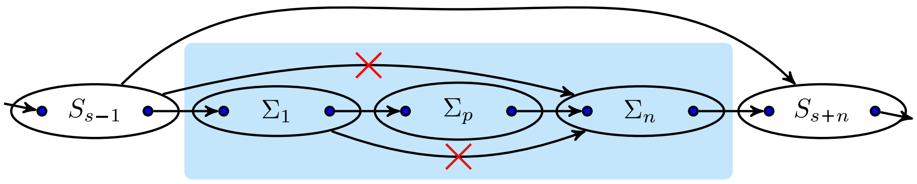
The example of Fig. 10 showed that optionality cannot be detected from α_{dfg} when it is contained in a sequence. The reason is, as shown in the proofs for α_{dfg}, that the footprints of Sect. 5.2 recover behaviour for *all* activities together, i.e. the $\rightarrow$-cut at the root level, from the α_{dfg} abstraction. As shown, in the presence of $\times(\tau, .)$-constructs, a new directly follows footprint is necessary for $\rightarrow$.

To detect optionality for only a subset of activities, we use cuts that partition an alphabet only partially, called *partial cuts*. Our proof strategy will be to show that if two trees differ in the α_{dfg}-abstraction, then they differ in at least one such partial cut showing a particular $\to$ relation that can be recovered from α_{dfg}.

Definition 7.2 (*partial cut*) Let Σ be an alphabet of activities and let $\oplus$ be a process tree operator. Then, $(\oplus, \Sigma_1, \dots \Sigma_n)$ is a *partial cut* of Σ if $\Sigma_1, \dots \Sigma_n \subseteq \Sigma$ and $\forall_{1 \leqslant i < j \leqslant n}\ \Sigma_i \cap \Sigma_j = \emptyset$. A partial cut expresses an $\oplus$-relation between its parts $\Sigma_1 \dots \Sigma_n$.

For the partial $\to$-cut footprint, one set Σ_p of activities has a special role which we call a *pivot*. Intuitively, occurrence of $a \in \Sigma_p$ signals the occurrence of the optional behaviour: if one of the activities in the partial cut appears in a trace, then the pivot must appear in the trace as well.

Definition 7.3 (*Partial$\to$-cut footprint*) Let Σ be an alphabet of activities, let α_{dfg} be a directly follows graph over Σ and let $C = (\to, S_1, \dots S_m)$ be a $\to$-cut of α_{dfg} according to Definition 5.2. Then, a partial cut $(\to, \Sigma_1, \dots \Sigma_n)$ is a *partial $\to$-cut* if there is a *pivot* Σ_p such that in α_{dfg}:



s.1 The partial cut is a consecutive part of C.
s.2 There are no end activities before the pivot in the partial cut.
s.3 There are no start activities after the pivot in the partial cut.
s.4 There are no directly follows edges bypassing the pivot in the partial cut.
s.5 The partial cut can be tightly avoided:

"Appendix B.3" provides the complete formal definition and proves information preservation according to our framework: in a process tree each $\to$-node has a pivot and adheres to the footprint. The proof that the pivot, and hence the $\to$-node, can be uniquely rediscovered uses that by the reduction rules, at least one of the children of any $\to$-node is not optional. This child is the pivot, and by the reduction rules, a pivot cannot be a sequential node itself.

For instance, consider the process trees M_{14} and M_{15} shown in Fig. 10. Their directly follows graphs shown in Fig. 10c are obviously different, and this becomes apparent in the new directly follows footprint: in $\alpha_{\text{dfg}}(M_{14})$, there is no pivot present, and therefore, there is partial cut that satisfies Definition 7.3 (this allows the conclusion that there is no nested $\to$-behaviour). However, in $\alpha_{\text{dfg}}(M_{15})$, both b and c are pivots, yielding the partial $\to$-cut $(\to, \{b\}, \{c\})$. Intuitively, considering the structure of the directly follows graph, the combination of b and c forms a partial cut because executing either implies executing the other, but both together can be avoided.

7.3 Inclusive choice

7.3.1 Idea

Preserving and recovering inclusive choice require remembering *more* behaviour than what is stored in α_{dfg} as illustrated by Fig. 11. We show that this additional information can be stored and recovered in a new abstraction, called α_{coo}. The different nature of the α_{coo}-abstraction requires a slightly different proof and reasoning strategy as we illustrate next on the example tree M_{17} of Fig. 11 before providing the definitions.

While the directly follows graph does not suffice to distinguish M_{17} from other process trees, it yields the concurrent cut $C = (\wedge, \{a\}, \{b\}, \{c, d\})$. From this concurrent cut, we conclude that c and d are not of interest for distinguishing $\vee$- from $\wedge$-behaviour; thus, we group them together.

From this concurrent cut, we work our way upwards. That is, we consider the α_{coo}-graph for our partition $\{\{a\}, \{b\}, \{c, d\}\}$, shown in Fig. 11f. In this α_{coo}-graph, we identify a footprint that involves two or more *sets of activities* of the partition. In our example, we identify that the sets $\{b\}$ and $\{c, d\}$ are related using $\vee$. Then, we merge b, c and d in our partition, which becomes $\{\{a\}, \{b, c, d\}\}$ and repeat the procedure.

Figure 11g shows the α_{coo}-graph belonging to $\{\{a\}, \{b, c, d\}\}$. In this graph, a footprint is present linking $\{a\}$ to $\{b, c, d\}$ using $\wedge$. Thus, we have shown that the α_{coo}-abstraction, that is, the family of graphs $\alpha_{\mathrm{coo}}(M_{17})$, can only belong to a process tree with root $\wedge$ and activity partition $\{a\}, \{b, c, d\}$.

The proof strategy of this section formalizes each of these steps and proves that at each step, footprints can only hold in the relevant α_{coo}-graph if and only if they correspond to nodes in the tree.

7.3.2 Coo-abstraction

A coo-abstraction α_{coo} is a *family* of *coo-graphs*, that is, a coo-abstraction contains a coo-graph for each partition of the alphabet of the language. Intuitively, the coo-abstraction indicates dependencies between the sets of activities that can occur in the language described by the α_{coo}-abstraction. Therefore, we first introduce a helper function that expresses which sets of activities occur together in a language. Second, we introduce coo-graphs and third, the coo-abstraction.

Definition 7.4 (*occurrence function f_o*) Let $S = \{\Sigma_1, \ldots \Sigma_n\}$ be a partition, let t be a trace and let L be a language. Then, the *occurrence function* of t under S yields the sets of S that occur in t or L:

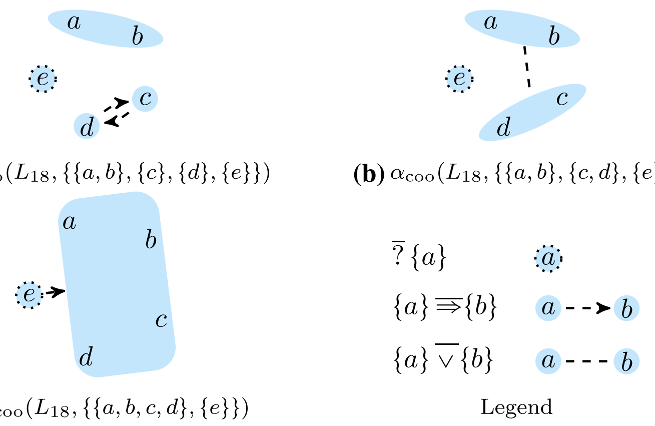
$$f_o(t, S) = \{\Sigma_i \mid \Sigma_i \in S \wedge \Sigma_i \cap \Sigma(t) \neq \emptyset\}$$
$$f_o(L, S) = \{f_o(t, S) \mid t \in L\}$$

For instance:

$$\begin{aligned}
L_{18} &= \{\langle a, b\rangle, && \langle d, c\rangle, && \langle d, a, c, b\rangle, \\
&\quad\ \langle a, b, e\rangle, && \langle e, d, c\rangle, && \langle e, a, d, b, c\rangle\} \\
S_{18} &= \{\{a, b\}, \{c\}, \{d\}, \{e\}\} \\
f_o(L_{18}, S_{18}) &= \{\{\{a, b\}\}, && \{\{c\}, \{d\}\}, && \{\{a, b\}, \{c\}, \{d\}\}, \\
&\quad\ \{\{a, b\}, \{e\}\}, && \{\{c\}, \{d\}, \{e\}\}, && \{\{a, b\}, \{c\}, \{d\}, \{e\}\}\}
\end{aligned}$$

Using this occurrence function, we can define coo-graphs. A coo-graph expresses three types of properties of and between sets:

- The unary *optionality* ($\overline{?}$) expresses that if the set occurs in a trace, then there is a trace without the set as well;



(a) $\alpha_{\text{coo}}(L_{18}, \{\{a,b\},\{c\},\{d\},\{e\}\})$ **(b)** $\alpha_{\text{coo}}(L_{18}, \{\{a,b\},\{c,d\},\{e\}\})$

(c) $\alpha_{\text{coo}}(L_{18}, \{\{a,b,c,d\},\{e\}\})$ Legend

Fig. 12 Several examples of coo-graphs for the language L_{18} and several partitions. Groups of activities are denoted by blue regions

- The binary *implication* ($\Rrightarrow$) expresses that if one set occurs, then the other set occurs as well;
- The binary *interchangeability* ($\overline{\vee}$) expresses that if one of the two sets occur, then either or both can occur.

Notice that $\Rrightarrow$-edges are directed, while $\overline{\vee}$-edges are undirected. Formally:

Definition 7.5 (*coo-graph* α_{coo}) Let L be a language, let S be a partition of $\Sigma(L)$ and let $A, B \in S$. Then, a *coo-graph* $\alpha_{\text{coo}}(L, S)$ is a graph in which the nodes are the activities of $\Sigma(L)$ and the edges connect sets of activities of S:

$$\begin{aligned}
\overline{?}(A) &\equiv \forall_{t \in f_o(L,S)}\ A \in t \Rightarrow t \setminus \{A\} \in f_o(L, S) \\
A \Rrightarrow B &\equiv \forall_{t \in f_o(L,S)}\ A \in t \Rightarrow B \in t \\
A\,\overline{\vee}\,B &\equiv \forall_{t \in f_o(L,S)}\ (A \in t \vee B \in t) \Rightarrow (t \cup \{A, B\} \in f_o(L, S) \wedge \\
&\qquad (t \cup \{A\}) \setminus \{B\} \in f_o(L, S) \wedge \\
&\qquad (t \cup \{B\}) \setminus \{A\} \in f_o(L, S))
\end{aligned}$$

For instance, Fig. 12 shows several α_{coo}-graphs of our example log L_{18}. Coo-graphs bear some similarities with directed hypergraphs [43], however use two types of edges ($\Rrightarrow$ and $\overline{\vee}$) and a node annotation ($\overline{?}$).

Finally, we combine all possible coo-graphs for a language into the coo-abstraction:

Definition 7.6 (*coo-abstraction* α_{coo}) Let L be a language. Then, the *coo-abstraction* $\alpha_{\text{coo}}(L)$ is the family of coo-graphs consisting of one coo-graph $\alpha_{\text{coo}}(L, S)$ for each possible partition S of $\Sigma(L)$.

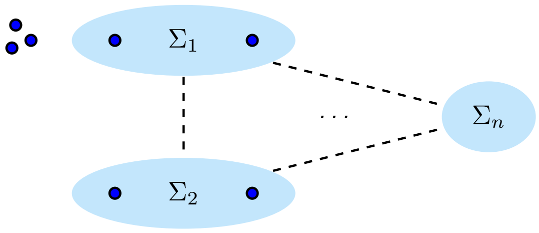
In the remainder of this section, we might omit L: α_{coo} denotes a particular coo-abstraction and $\alpha_{\text{coo}}(S)$ denotes one of its coo-graphs for a particular partition S. Next, we introduce footprints that link the coo-relations to process tree operators.

7.3.3 Footprints

For inclusive choice and concurrency, we can now introduce the α_{coo}-abstraction footprints. We first give the footprints, after which we illustrate them using an example.

Definition 7.7 (*partial* $\vee$*-cut*) Let Σ be an alphabet of activities, S a partition of Σ, let $\alpha_{\text{coo}}(S)$ be a coo-graph, let α_{dfg} be a directly follows graph, and let $C = (\vee, \Sigma_1, \ldots \Sigma_n)$ be a partial cut such that $\forall_{1 \leqslant i \leqslant n}\ \Sigma_i \in S$. Then, C is a partial $\vee$-cut if in $\alpha_{\text{coo}}(S)$ and α_{dfg}:

o.1 C is a part of a α_{dfg}-concurrent cut (Definition 5.2).
o.2 All parts are interchangeable ($\overline{\vee}$) in $\alpha_{\text{coo}}(S)$::



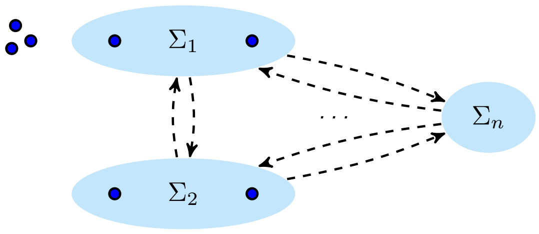
A formal definition is included in "Appendix B.4".

Definition 7.8 (*partial* $\wedge$*-cut*) Let Σ be an alphabet of activities, S a partition of Σ, let $\alpha_{\text{coo}}(S)$ be a coo-graph, let α_{dfg} be a directly follows graph, and let $C = (\wedge, \Sigma_1, \ldots \Sigma_n)$ be a partial cut such that $\forall_{1 \leqslant i \leqslant n}\ \Sigma_i \in S$. Then, C is a partial $\wedge$-cut if in $\alpha_{\text{coo}}(S)$ and α_{dfg}:

c.1 C is a part of an α_{dfg}-concurrent cut (Definition 5.2).

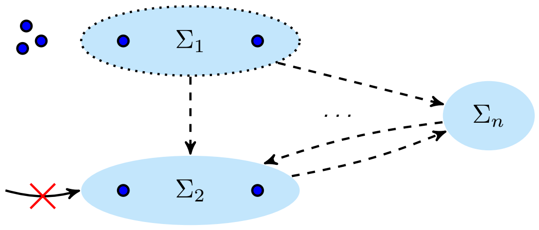
Furthermore, in $\alpha_{\text{coo}}(S)$:
– EITHER –

c.2.1 All parts bi-imply ($\overline{\rightrightarrows}$) one another::



– OR –

c.3.1 The first part is optional ($\overline{?}$).
c.3.2 The first part implies all other parts.
c.3.3 All non-first parts bi-imply one another.
c.3.4 No non-first part Σ_i is implied by any part not in C::



A formal definition and a proof that these footprints apply in process trees without duplicate activities are included in "Appendix B.5".

For instance, consider a process tree M_{19} without duplicate activities, and one of its subtrees $M_{20} = \vee(M_1, M_2)$, in which M_1 and M_2 are arbitrary subtrees. Furthermore, consider a partition S of the activities of M_{20}. In the language of M_{20}, whenever in a trace M_1 is executed, there must be a similar trace in which both M_1 and M_2 are executed, and there must be a trace in which M_2 is executed but M_1 is not. Then in $\alpha_{\text{coo}}(\mathcal{L}(M_{19}), S)$, the coo-relation $\Sigma(M_1) \overline{\vee}\, \Sigma(M_2)$ holds, and hence, $(\vee, \Sigma(M_1), \Sigma(M_2))$ is a partial $\vee$-cut of $\alpha_{\text{coo}}(\mathcal{L}(M_{19}), S)$.

7.4 A class of trees: C_{coo}

In this section, we introduce the class of trees that we consider for the handling of $\vee$- and $\times(\tau, .)$-constructs: C_{coo}. This class of trees extends C_{dfg} by including $\vee$-nodes and some τ-leaves.

Definition 7.9 (*Class* C_{coo}) Let Σ be an alphabet of activities, then the following process trees are in C_{coo}:

- τ is in C_{coo};
- a with $a \in \Sigma$ is in C_{coo};
- Let $M_1 \ldots M_n$ be reduced process trees in C_{coo} without duplicate activities: $\forall_{i \in [1 \ldots n], i \neq j \in [1 \ldots n]}\ \Sigma(M_i) \cap \Sigma(M_j) = \emptyset$. Then,

  - A node $\oplus(M_1, \ldots M_n)$ with $\oplus \in \{\times, \rightarrow, \wedge, \vee\}$ is in C_{coo}
  - An interleaved node $\leftrightarrow (M_1, \ldots, M_n)$ is in C_{coo} if all:

  i.1 At least one child has disjoint start and end activities: $\exists_{i \in [1 \ldots n]}$ $\text{Start}(M_i) \cap \text{End}(M_i) = \emptyset$
  i.2 no child is interleaved itself: $\forall_{i \in [1 \ldots n]}\ M_i \neq \leftrightarrow (\ldots)$
  i.3 no child is optionally interleaved: $\forall_{i \in [1 \ldots n]}\ M_i \neq \times(\tau, \leftrightarrow (\ldots))$
  i.4 each concurrent or inclusive choice child has at least one child with disjoint start and end activities: $\forall_{i \in [1 \ldots n]}\ M_i = \oplus(M'_1, \ldots M'_m) \Rightarrow \exists_{j \in [1 \ldots m]}\ \text{Start}(M'_j) \cap \text{End}(M'_j) = \emptyset$ with $\oplus \in \{\wedge, \vee\}$

  - A loop node $\circlearrowleft (M_1, \ldots M_n)$ is in C_{coo} if all:

  l.1 the body child is not concurrent: $M_1 \neq \wedge(\ldots)$
  l.2 no redo child can produce the empty trace: $\forall_{i \in [2 \ldots n]}\ \epsilon \notin \mathcal{L}(M_i)$

Notice that $\mathrm{C}_{\text{dfg}} \subseteq \mathrm{C}_{\text{msd}} \subseteq \mathrm{C}_{\text{coo}}$. We illustrate the necessity of the newly added or relaxed requirements:

- Requirement C_{coo}.i.3: the child of an interleaved node cannot be an optional interleaved node. Nested interleaved nodes cannot be distinguished using directly follows graphs, as was shown in Sect. 5.3. As an optional nested interleaved has the same directly follows graph as a nested interleaving, a similar argument applies here.
- Requirement C_{coo}.l.2: redo children of loops cannot be optional. As shown in Sect. 7.1, the reduction rules of Sect. 4 are not strong enough to distinguish trees with $\circlearrowleft (., \tau)$ constructs. Furthermore, as shown in Sect. 7.1, optional children under a $\circlearrowleft$ might invalidate directly follows footprints.

From our discussion in Sect. 7.1, it follows that $\times(\tau, .)$ constructs preserve the footprints of the process tree operators $\times$, $\rightarrow$, $\leftrightarrow$, $\wedge$ and $\circlearrowleft$ for trees in $\mathrm{C}_{\mathrm{coo}}$.

Corollary 7.1 (optionality preserves cuts) *Take two reduced process trees $M \in \mathrm{C}_{dfg}$, and $M' \in \mathrm{C}_{coo}$, such that $M = \oplus(M_1, \ldots M_m)$, $M' = \oplus(M'_1, \ldots M'_n)$ and each M'_i is equal to either M_i or $\times(\tau, M_i)$. Then, $\alpha_{dfg}(M')$ contains a cut $(\oplus, \Sigma(M_1) \ldots \Sigma(M_m))$, i.e. a footprint according to Definition 5.2.*

7.5 Uniqueness

In this section, we show that the coo-abstraction in combination with the directly follows abstraction provides enough information to distinguish all languages of $\mathrm{C}_{\mathrm{coo}}$. To prove this, we follow a strategy similar to the proofs for directly follows graphs and minimum self-distances: we first show that if the root operators of two process trees are different, then their abstractions must be different.

First, we show that the introduced α_{dfg}-footprints for nested $\rightarrow$- and $\times(\tau, .)$-structures are correct (“Appendix F.1”). That is, that each two different process trees of $\mathrm{C}_{\mathrm{coo}}$ with top parts consisting of such constructs have different directly follows graphs. Second, we show that the introduced α_{coo}-footprints and the reasoning procedure illustrated in Sect. 7.3.1 are correct. That is, that each two different process trees of $\mathrm{C}_{\mathrm{coo}}$ with top parts consisting of nested $\vee$-, $\wedge$- or $\times(\tau, .)$-constructs have different α_{coo}-graphs (“Appendix F.2”). Third, we show that if the partition of activities over the direct children of the root is different, then their abstractions must be different, and that hence the abstractions are uniquely defined (“Appendix F.3”).

Consequently, uniqueness holds:

Corollary 7.2 (Uniqueness for $\mathrm{C}_{\mathrm{coo}}$) *There are no two different reduced process trees of C_{coo} with equal languages: for all reduced $K \neq M$ of C_{coo}, $\mathcal{L}(K) \neq \mathcal{L}(M)$.*

8 Application and evaluation on real-life logs

In the previous sections, we showed under which circumstances behavioural information captured in α_{dfg}, α_{msd} and α_{coo} can be *exactly* recovered. In this section, we evaluate the practical applicability, relevance and robustness of these abstractions and their footprints for process discovery on 13 real-life event logs:[1] BPIC$\underline{11}$, BPIC$\underline{12}$, BPIC$\underline{13}$ ($\underline{\text{c}}$losed $\underline{\text{p}}$roblems, $\underline{\text{in}}$cident), BPIC$\underline{14}$, BPIC$\underline{15(1-5)}$, BPIC$\underline{17}$, $\underline{\text{R}}$oad Traffic $\underline{\text{F}}$ine Management and $\underline{\text{S}}$epsis.

We wanted to investigate (Q1) how much α_{dfg} alone suffices for process discovery on real-life event logs, and (Q2) whether and how much α_{msd}, optionality footprints, and α_{coo} contribute to discovering processes on real-life event logs.

We used the Inductive Miner framework [40] to compare the effect of the different abstractions in algorithms that all follow the same principle. All these algorithms discover a process model from an event log in a recursive way: they abstract the log and search for footprints. If a footprint is found, then the log is split accordingly, the footprint is recorded as a process tree node and recursion continues on the sublogs. If no footprint is found in a sublog, then a generalization (fall through) is applied. In some cases, such a fall through allows the recursion

[1] Available from https://data.4tu.nl/repository/collection:event_logs_real.

to continue. However, as a last resort, a model representing all possible behaviour (a flower model) is returned by that step of the recursion. In that case, the model will give no insights and be highly imprecise for the subset of activities in the sublog. Thus, the more footprints are identified by these algorithms, the more behavioural relations are captured by the syntax of the discovered model giving more insights into the process. In this experiment, we used the following four algorithms to compare the ability of the abstractions to identify footprints, i.e. meaningful behavioural relations, for as many activities as possible:

– Inductive Miner* (IM*), which is a version of Inductive Miner [40], tailor-made for this experiment, that only uses the directly follows abstraction (α_{dfg}) and the footprints discussed in Sect. 5.
– Inductive Miner-all operators (IMa) [40] uses the α_{dfg}, α_{msd} and α_{coo} abstractions and all footprints mentioned in this paper.
– Inductive Miner-infrequent (IMf) [40] uses the α_{dfg} and α_{msd} abstractions and footprints, and discovers τ constructs (Definition 7.1) and nested $\rightarrow$-constructs (Definition 7.3). Furthermore, IMf filters the α_{dfg} abstraction if necessary to handle infrequent behaviour (please refer to [40] for more details).
– Inductive Miner-infrequent and all operators (IMfa) [40] combines IMa and IMf: it uses all abstractions and footprints, and filters the α_{dfg} abstraction to handle infrequent behaviour if necessary.

We applied the miners to the 13 original logs, as well as to seven versions of the logs provided by a recent process discovery benchmark [10] from which events were removed for infrequent behaviour using the technique of [44].[2]

To answer Q1 and Q2, we measured (0) the number of activities that were included in the model; (1) the number of footprints that were identified indicating how many meaningful behavioural relations between activities could be found, (2) the number of activities for which a footprint could be found, (3) the types of footprints that were identified. Furthermore, we measured (4) the quality of the resulting model as precision and the recall of the model with respect to the log. We used the Projected Conformance Checking (PCC) [37] framework to measure precision and recall, which projects the behaviour of event log and model onto all subsets of activities (in our experiments, of size $k = 2$) and computes precision and recall using automata. The PCC framework with $k = 2$ was the only measure and parameter setting for which values could be obtained for all models and logs. We report normalized precision $p_n = (p - p_f)/(1 - p_f)$ against the precision p_f of the flower model.[3] The experiment's code is available [45].

The results are shown in Table 1 (all intermediate models are available from [46]). Comparing IM* and IMa for unfiltered logs, group (1) shows that α_{dfg} alone leads to significantly less discovered footprints (9 on average) than when α_{msd} and α_{coo} are also used (300 on average). Group (2) shows that using α_{dfg} alone only discovers meaningful behavioural relations for 2% of the activities, whereas also using α_{msd} and α_{coo} leads to meaningful behavioural relations for 71% of the activities on average (93% on filtered logs). Group (3) shows that in *each* log either optionality or α_{coo} footprints (or both) were discovered. Group (4) shows that the discovery of more footprints (through α_{msd} and α_{coo}) increases precision for all logs (except IMa on the 13.in-log). For the RF-log, the use of α_{msd} and α_{coo} alone (IMa) leads to a highly precise and fully fitting model. Models with optionality footprints introduce invisible skips which limits the gain in precision.

[2] The filtered logs are available from http://doi.org/10.4121/uuid:adc42403-9a38-48dc-9f0a-a0a49bfb6371.

[3] NB: if not all activities are present in a process model, PCC might report a negative p_n.

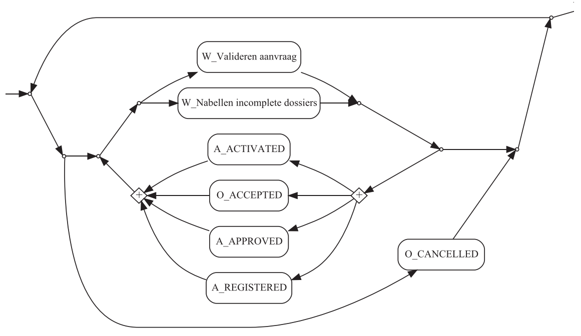
Table 1 the number of footprints identified by IM*, IMa, IMf and IMfa on filtered and non-filtered real-life event logs (colour figure online)

logs	0) act			1) footprints				2) % act in footpr.				3) optionality			seq.opt			or		4) precisison				rec.
	Log	IMf	IMfa	IM*	IMa	IMf	IMfa	IM*	IMa	IMf	IMfa	IMa	IMf	IMfa	IMa	IMf	IMfa	IMa	IMfa	IM*	IMa	IMf	IMfa	IMf
orig.	**206**	**59%**	**58%**	**9**	**300**	**139**	**133**	**2%**	**71%**	**94%**	**95%**	**122**	**78**	**72**	**2**	**3**	**4**	**2**	**5**	**0.03**	**0.2**	**0.53**	**0.52**	**0.93**
13.cp	4	100%	100%	3	8	3	3	0%	50%	100%	100%	1	1	1	0	0	0	0	0	0	0.19	0.62	0.62	0.88
13.in	4	75%	75%	3	6	5	5	0%	100%	67%	67%	1	0	0	0	0	0	1	0	0	-0.01	0.54	0.54	0.63
RF	11	100%	100%	4	17	13	13	9%	100%	100%	100%	8	8	7	0	1	1	1	1	0.16	0.78	0.87	0.86	0.99
Sep	16	88%	88%	3	23	10	9	0%	69%	100%	100%	6	5	4	0	0	0	0	0	0	0.36	0.76	0.76	0.86
12	24	96%	96%	4	39	30	29	8%	71%	87%	91%	8	4	4	1	0	0	2	1	0.14	0.26	0.36	0.36	0.97
17	26	92%	92%	6	43	25	26	4%	69%	83%	83%	6	8	8	0	0	0	2	1	0.1	0.28	0.35	0.19	0.84
14	39	100%	100%	3	24	25	24	0%	15%	100%	100%	2	11	10	0	1	1	0	1	0	0.02	0.05	0.05	1
15.4	356	65%	65%	19	528	268	257	0%	77%	100%	100%	233	163	148	7	5	7	3	8	0	0.08	0.51	0.51	0.99
15.3	383	75%	75%	18	579	354	339	0%	77%	100%	100%	264	209	187	4	10	13	3	22	0	0.13	0.39	0.39	0.99
15.5	389	57%	57%	3	592	235	227	0%	74%	100%	100%	244	134	125	3	8	9	3	11	0	0.13	0.59	0.59	0.99
15.1	398	61%	61%	17	541	263	252	0%	84%	100%	99%	285	149	138	7	8	9	5	8	0	0.14	0.59	0.58	0.99
15.2	410	54%	54%	25	560	242	239	1%	84%	100%	100%	287	131	126	9	3	4	4	5	0	0.17	0.45	0.45	0.99
11	624	41%	39%	3	937	333	311	0%	57%	91%	91%	235	186	175	0	9	10	2	2	0	0.04	0.84	0.84	0.98
filtered	**54**	**99%**	**99%**	**26**	**79**	**60**	**59**	**12%**	**93%**	**95%**	**100%**	**40**	**32**	**30**	**4**	**4**	**4**	**1**	**1**	**0.01**	**0.14**	**0.42**	**0.39**	**0.97**
14	9	100%	100%	19	13	11	11	11%	89%	67%	100%	2	2	2	0	0	0	1	2	0	0.02	0.05	0.05	1
17	18	100%	100%	18	21	18	18	50%	100%	100%	100%	6	6	6	2	2	2	0	0	0.1	0.28	0.35	0.19	0.84
15.3	62	100%	100%	4	103	70	63	5%	84%	100%	100%	43	38	31	1	0	0	3	4	0	0.13	0.39	0.39	0.99
15.4	65	100%	100%	21	102	79	77	5%	92%	100%	100%	52	38	37	6	6	6	2	1	0	0.08	0.51	0.51	0.99
15.1	70	100%	100%	77	88	71	73	7%	99%	100%	100%	52	40	39	7	6	7	0	0	0	0.14	0.59	0.58	0.99
15.5	74	96%	96%	23	100	75	77	4%	99%	100%	100%	62	43	42	5	7	7	1	1	0	0.13	0.59	0.59	0.99
15.2	82	98%	98%	17	125	95	91	4%	88%	100%	100%	63	55	51	6	5	5	2	2	0	0.17	0.45	0.45	0.99
scale		0%	33%	0	250	500	750	0%	50%	75%	100%	0	60	120		0	10	20	30	0	0.3	0.6	0.9	
		66%	100%									180	240	300										

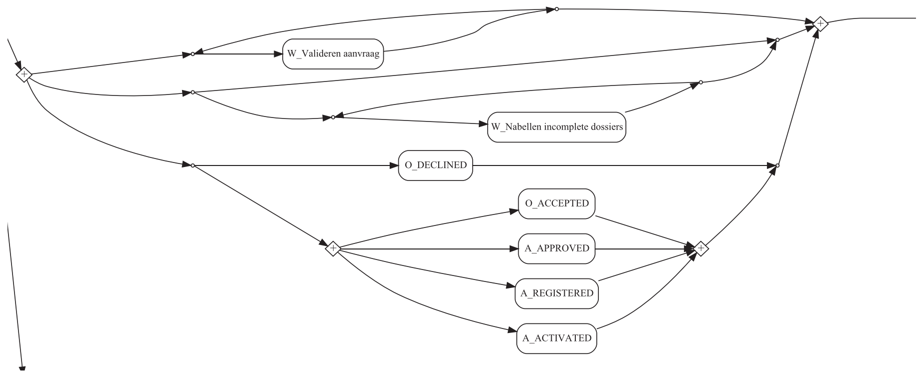
The algorithms that filter the α_{dfg} abstraction (IMf, IMfa) omit some activities from the abstraction and ultimately from the model (group (0)). For large unfiltered logs, 39–75% of the activities remain in abstractions, for smaller or filtered logs 100% remain. By group (2), using only α_{dfg}, α_{msd}, and the optionality footprints (IMf) leads to 94% of the remaining activities being in a meaningful behavioural relation; also using α_{coo} (IMfa) increases this share further to 95% for the unfiltered and 100% for the filtered logs. Group (3) shows that filtering α_{dfg} reduces the number of optionality footprints (IMf, IMfa vs IMa) and may increase the α_{coo} footprint. Group (4) confirms again that filtering α_{dfg} significantly increases precision while reducing recall (the last column reports recall for IMf, the value for IMfa differed by at most 0.01). In most cases, using α_{coo} and α_{dfg}-filtering together has no impact on precision, but in some cases precision may decrease. The lower number of optionality footprints for the filtered logs can be explained by the filtering procedure of [44], which seems to remove events that could be skipped in other traces and thus have a lower frequency.

Altogether, we found that α_{dfg} alone is insufficient to discover process models from real-life event logs, answering Q1. Furthermore, α_{msd}, optionality footprints and α_{coo} lead to significantly more behavioural relations discovered in all real-life logs. Yet, not all logs benefit from α_{coo}, and thus, non-behaviour preserving steps such as filtering α_{dfg} remain necessary to obtain models of high precision and recall, which answers Q2.

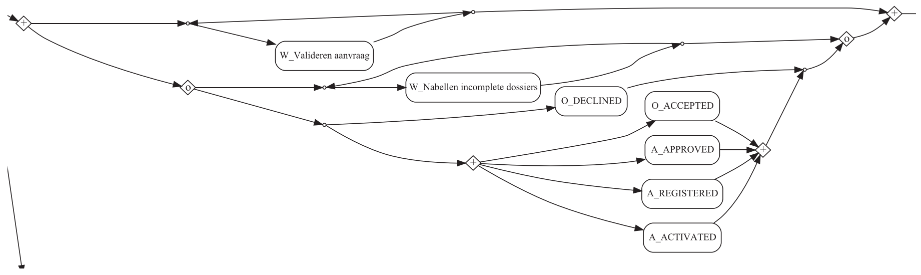
The following examples illustrate the influence of the different abstractions on process discovery when applying IM* (Fig. 13a), IMf (Fig. 13b) and IMfa (Fig. 13c) on BPIC12 [47] filtered to only contain start and completion events, which was recorded from a loan application process in a Dutch financial institution. All three models express that `A_ACTIVATED`, `O_ACCEPTED`, `A_APPROVED` and `A_REGISTERED` are executed concurrently, which holds in $> 99\%$ of the traces. However, only in Fig. 13b, c, execution of any activity shown



(a) Inductive Miner IM*using only α_{dfg}-footprints.



(b) Inductive Miner - infrequent [17, IMf].



(c) Inductive Miner using α_{dfg}- and α_{coo}-footprints [17, IMfa].

Fig. 13 Excerpts from models discovered by IM*, IMf and IMfa on the BPI Challenge 2012 log

implies the execution of `W_Valideren aanvraag`, which also holds in $> 99\%$ of the traces. The model in Fig. 13b describes that `W_Nabellen incomplete dossiers` is followed by either `O_DECLINED` or all four activities below it, but this dependency is violated in 34% of the traces. In contrast, this dependency is absent in the model in Fig. 13c due to the inclusive choice gateway. These examples together illustrate that the use of more footprints and constructs might reveal information that would otherwise remain undiscovered and can increase the reliability of constraints that are recovered from event logs.

9 Conclusion

In this paper, we studied capabilities of abstractions in process discovery for preserving and inferring information from event logs. In particular, we formally studied the inherent limitations of three concrete abstractions for encoding and recovering infinite behaviour from a finite abstraction. As information cannot be fully preserved on finite abstractions even for regular languages [8], we focused on block-structured process models. We defined a *formal research framework* which we instantiated and applied for three different abstractions. We tightly characterized the behaviour which can be preserved and recovered from the frequently used directly follows abstraction α_{dfg}. To overcome the limits of α_{dfg}, we developed the minimum self-distance α_{msd} and the concurrent-optional-or α_{coo} abstractions. We thereby established for the first time precise boundaries for information preservation and recovery also for interleaving, unobservable behaviour due to skipping and inclusive choices, demonstrating the versatility of the research framework.

The principles and hard information-preservation limits identified in this paper have implications on process mining applications. First, algorithms using any of the discussed abstractions $\alpha_{\text{dfg}}, \alpha_{\text{msd}}, \alpha_{\text{coo}}$ are limited to correctly recovering only models from the classes these abstractions actually can distinguish; models outside this call *will* inevitably be misidentified. Second, our results guarantee that on any finite log $L' \subseteq \mathcal{L}(M)$ of some unknown process M where $\alpha(L') = \alpha(L)$, an algorithm using α has sufficient information to *rediscover* $\gamma(\alpha(L')) = M = \gamma(\alpha(L))$. Algorithms such as Inductive Miner [5] use this principle to obtain information about M from the abstraction $\alpha(L')$, but also leverage more information from L' to handle deficiencies in smaller event logs. Third, the formal results and our evaluation on 13 real-life logs show that α_{dfg} alone is too limited to reliably preserve and recover information of real-life processes; any discovery algorithm aiming to reliably recover *all* behaviour a real-life process needs to use more information than α_{dfg}, such as provided by α_{msd} or α_{coo}. Fourth, when combining α_{dfg} with α_{msd} and α_{coo}, significantly more behaviour, in particular inclusive choices and *unobservable behaviour* due to skips, not considered previously, can be *correctly* recovered from these abstractions also in the presence of infrequent and noisy behaviour. Particularly, α_{msd} and α_{coo} are central to feasible outcomes of process discovery on real-life logs. Ultimately, as our results are defined on the abstractions and independent of any specific discovery algorithm, they can be used in any process mining technique using abstractions. Information-preservation properties of α_{dfg} are already exploited in discovering process models from event streams [12] and in maintaining database indexes for efficient discovery from large event data [13,14]. The results of this work show that to fully preserve and recover behavioural information in these settings, further abstractions and indices must be developed. The results of this work may provide templates for the required data structures.

Limitations of our study are that our characterization for behaviour preserved under abstraction states tight sufficient conditions but lacks necessary conditions. Further, although the evaluation shows the applicability on real-life logs, we only prove information preservation and recovery results for block-structured processes with unique activities, leaving other types of models as future work. The abstraction α determines when a log is *complete* to allow discovering the correct model, that is, when $\alpha(L') = \alpha(\mathcal{L}(M))$ for $L' \subset \mathcal{L}(M)$. As we have shown, for most processes more than one abstraction have to be considered to preserve all information. The question when a log L' is large enough (also in presence of noise) to be complete in *all relevant abstractions* (without knowing $\mathcal{L}(M)$) is an open question to which this paper may serve as input.

Finally, the results of this paper could inform further research in process mining. The results obtained in this paper could support the development of better *generalization* measures to quantify the degree with which a model captures future, unseen behaviour [2]. The information preservation by abstractions is also relevant for the management and analysis of event data in distributed settings, where not all information can or shall be shared explicitly between two different sources, but instead information-preserving abstractions or projections of the data may be shared guaranteeing or preventing the reconstruction of certain information from the source data.

Acknowledgements We thank Wil van der Aalst for his contribution to earlier versions of the directly follows and minimum self-distance abstractions [5]. Furthermore, we thank Alifah Syamsiyah for her corrections in the examples of the minimum self-distance abstraction.

A Semantics of process trees

Definition A.1 (*process tree semantics*) Let Σ be an alphabet of activities and let $\oplus_{\mathcal{L}}$ be a language-combination function, then

$$\begin{aligned}\mathcal{L}(a) &= \{\langle a\rangle\} \text{ for } a \in \Sigma\\ \mathcal{L}(\tau) &= \{\epsilon\}\\ \mathcal{L}(\oplus(M_1, \dots M_n)) &= \oplus_{\mathcal{L}}(\mathcal{L}(M_1), \dots \mathcal{L}(M_n))\end{aligned}$$

Then, the semantics of process tree operators can be described by a specific language-combination function $\oplus_{\mathcal{L}}$, depending on the operator $\oplus$:

Definition A.2 (*process tree operator semantics*) Let $\sqcup\!\sqcup$ be a language shuffle function shuffling the events in traces $t_1 \dots t_n$, and languages $L_1 \dots L_n$:

$$\begin{aligned}t \in t_1 \sqcup\!\sqcup \dots \sqcup\!\sqcup t_n \Leftrightarrow\ & \exists_{\text{function } f}\\ & \forall_{1 \leqslant i_1 < i_2 \leqslant |t| \wedge f(i_1)=(t,k_1) \wedge f(i_2)=(t_j,k_2)}\ k_1 < k_2 \wedge\\ & \forall_{1 \leqslant i \leqslant n \wedge f(i)=(t,k)}\ t(i) = t_j(k)\\ L_1 \sqcup\!\sqcup L_2 \sqcup\!\sqcup \dots L_n = \{t \mid\ & \forall_{1 \leqslant i \leqslant n}\ t_i \in L_i \wedge t = t_1 \sqcup\!\sqcup t_2 \sqcup\!\sqcup \dots t_n\}\end{aligned}$$

Furthermore, let $p(n)$ denote the set of all permutations of the numbers $1 \dots n$ and let $q(n)$ denote the set of all subsets of the numbers $1 \dots n$. Then,

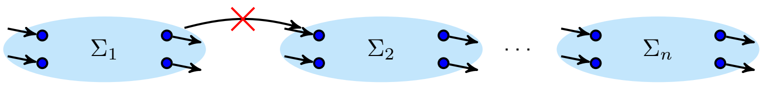
$$\begin{aligned}\times_{\mathcal{L}}(L_1, \dots, L_n) &= L_1 \cup L_2 \cup \dots \cup L_n\\ \rightarrow_{\mathcal{L}}(L_1, \dots, L_n) &= L_1 \cdot L_2 \cdots L_n\\ \leftrightarrow_{\mathcal{L}}(L_1, \dots, L_n) &= \bigcup_{(i_1 \dots i_n) \in p(n)} \rightarrow_{\mathcal{L}}(L_{i_1}, \dots, L_{i_n})\\ \wedge_{\mathcal{L}}(L_1, \dots, L_n) &= L_1 \sqcup\!\sqcup L_2 \sqcup\!\sqcup \dots L_n\\ \vee_{\mathcal{L}}(L_1, \dots, L_n) &= \bigcup_{\{i_1 \dots i_m\} \in q(n)} \wedge_{\mathcal{L}}(L_{i_1}, \dots, L_{i_m})\\ \circlearrowleft_{\mathcal{L}}(L_1, \dots, L_n) &= L_1 \cdot (\times_{\mathcal{L}} \cdot (L_2, \dots, L_n) \cdot L_1)^* \qquad \text{for } \circlearrowleft, \quad n \geqslant 2\end{aligned}$$

B Footprints

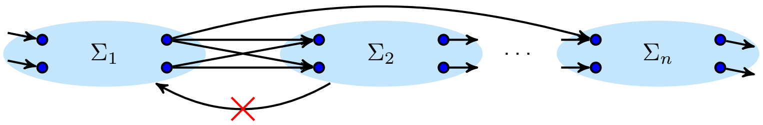
B.1 Directly follows (Definition 5.2)

Let α_{dfg} be a directly follows relation and let $c = (\oplus, \Sigma_1, \ldots \Sigma_n)$ be a cut, consisting of a process tree operator $\oplus \in \{\times, \rightarrow, \leftrightarrow, \wedge, \circlearrowleft\}$ and a partition of activities with parts $\Sigma_1 \ldots \Sigma_n$ such that $\Sigma(\alpha_{\text{dfg}}) = \bigcup_{1 \leqslant i \leqslant n} \Sigma_i$ and $\forall_{1 \leqslant i < j \leqslant n}\ \Sigma_i \cap \Sigma_j = \emptyset$.

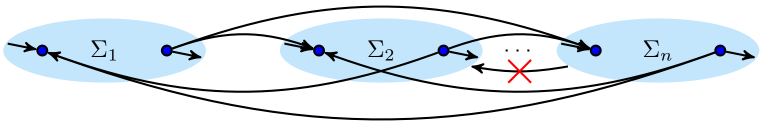
– Exclusive choice. c is an *exclusive choice cut* in α_{dfg} if $\oplus = \times$ and
  - x.1 No part is connected to any other part:
    $\forall_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j}\ \forall_{a \in \Sigma_i, b \in \Sigma_j}\ \neg\alpha_{\text{dfg}}(a, b) \wedge \neg\alpha_{\text{dfg}}(b, a)$:



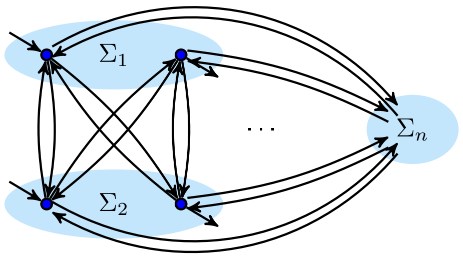
– Sequential. c is a *sequence cut* in α_{dfg} if $\oplus = \rightarrow$ and
  - s.1 Each node in a part is indirectly and only connected to all nodes in the parts “after” it:
    $\forall_{1 \leqslant i < j \leqslant n}\ \forall_{a \in \Sigma_i, b \in \Sigma_j}\ \alpha_{\text{dfg}}^{+}(a, b) \wedge \neg\alpha_{\text{dfg}}^{+}(b, a)$:



– Interleaved. c is an *interleaved cut* in α_{dfg} if $\oplus = \leftrightarrow$ and
  - i.1 Between parts, all and only connections exist from an end to a start activity:
    $\forall_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j}\ \forall_{a \in \Sigma_i, b \in \Sigma_j}\ \alpha_{\text{dfg}}(a, b) \Leftrightarrow (a \in \text{End} \wedge b \in \text{Start}))$:

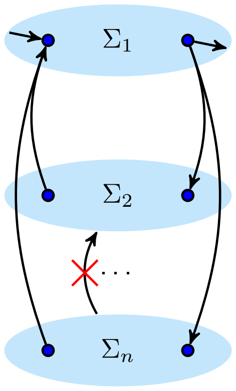


– Concurrent. c is a *concurrent cut* in α_{dfg} if $\oplus = \wedge$ and
  - c.1 Each part contains a start and an end activity:
    $\forall_{1 \leqslant i \leqslant n}\ \text{Start}(\alpha_{\text{dfg}}) \cap \Sigma_i \neq \emptyset \wedge \text{End}(\alpha_{\text{dfg}}) \cap \Sigma_i \neq \emptyset$
  - c.2 All parts are fully interconnected:
    $\forall_{1 \leqslant i < n, 1 \leqslant j \leqslant n, i \neq j}\ \forall_{a \in \Sigma_i, b \in \Sigma_j}\ \alpha_{\text{dfg}}(a, b) \wedge \alpha_{\text{dfg}}(b, a))$:



– Loop. c is a *loop cut* in α_{dfg} if $\oplus = \circlearrowleft$ and

1.1 All start and end activities are in the body (i.e. the first) part: $\text{Start}(\alpha_{\text{dfg}}) \cup \text{End}(\alpha_{\text{dfg}}) \subseteq \Sigma_1$
1.2 Only start/end activities in the body part have connections from/to other parts: $\forall_{2\leqslant j\leqslant n} \forall_{a\in\Sigma_1,b\in\Sigma_j}\ \alpha_{\text{dfg}}(a,b) \Rightarrow a \in \text{End}(\alpha_{\text{dfg}})$
$\forall_{2\leqslant j\leqslant n} \forall_{a\in\Sigma_1,b\in\Sigma_j}\ \alpha_{\text{dfg}}(b,a) \Rightarrow a \in \text{Start}(\alpha_{\text{dfg}})$
1.3 Redo parts have no connections to other redo parts: $\forall_{2\leqslant i\leqslant n,2\leqslant j\leqslant n,i\neq j} \forall_{a\in\Sigma_i,b\in\Sigma_j}\ \neg\alpha_{\text{dfg}}(a,b) \wedge \neg\alpha_{\text{dfg}}(b,a)$
1.4 If an activity from a redo part has a connection to/from the body part, then it has connections to/from all start/end activities:
$\forall_{2\leqslant i\leqslant n} \forall_{a\in\text{Start},b\in\Sigma_i}\ \alpha_{\text{dfg}}(b,a) \Leftrightarrow \forall_{c\in\text{Start}(\alpha_{\text{dfg}})}\ \alpha_{\text{dfg}}(b,c)$
$\forall_{2\leqslant i\leqslant n} \forall_{a\in\text{End},b\in\Sigma_i}\ \alpha_{\text{dfg}}(a,b) \Leftrightarrow \forall_{c\in\text{End}(\alpha_{\text{dfg}})}\ \alpha_{\text{dfg}}(c,b)$):

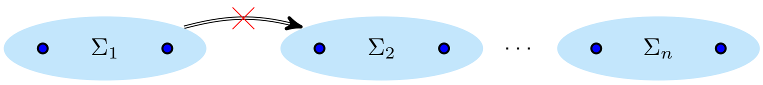


B.2 Minimum self-distance (Definition 6.3)

Let α_{msd} be a minimum self-distance graph and let $c = (\oplus, \Sigma_1, \ldots \Sigma_n)$ be a cut, consisting of a process tree operator $\oplus \in \{\times, \rightarrow, \leftrightarrow, \wedge, \circlearrowleft\}$ and a partition of activities with parts $\Sigma_1 \ldots \Sigma_n$ such that $\Sigma(\alpha_{\text{msd}}) = \bigcup_{1\leqslant i\leqslant n} \Sigma_i$ and $\forall_{1\leqslant i<j\leqslant n}\ \Sigma_i \cap \Sigma_j = \emptyset$.

– Concurrent and interleaved. If $\oplus = \wedge$ or $\oplus = \leftrightarrow$, then in α_{msd}:

ci.1 There are no α_{msd} connections between parts: $\forall_{i\in[1\ldots n],i\neq j\in[1\ldots n]} \forall_{a\in\Sigma_i,b\in\Sigma_j}\ \neg\alpha_{\text{msd}}(a,b)$):

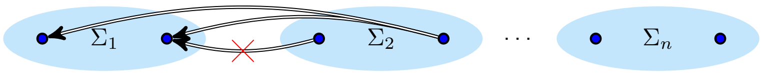


– Loop. If $\oplus = \circlearrowleft$ then in α_{msd}:

1.1 Each activity has an outgoing edge: $\forall_{a\in\Sigma(\alpha_{\text{msd}})} \exists_{b\in\Sigma(\alpha_{\text{msd}}),b\neq a}\ \alpha_{\text{msd}}(a,b)$

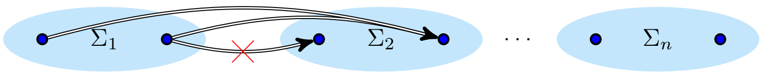
l.2 All redo activities that have a connection to a body activity have connections to the same body activities:

$$\forall_{2\leqslant i\leqslant n, 2\leqslant j\leqslant n}\ \forall_{a\in\Sigma_i, b\in\Sigma_j}\ (\{c \mid \alpha_{\text{msd}}(a,c)\}\cap\Sigma_1 \neq \emptyset \wedge \{c \mid \alpha_{\text{msd}}(b,c)\}\cap\Sigma_1 \neq \emptyset) \Rightarrow \{c \mid \alpha_{\text{msd}}(a,c)\}\cap\Sigma_1 = \{c \mid \alpha_{\text{msd}}(b,c)\}\cap\Sigma_1$$

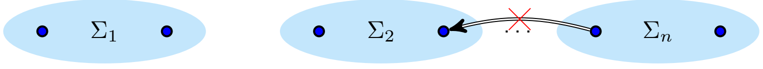


l.3 All body activities that have a connection to a redo activity have connections to the same redo activities:

$$\forall_{a,b\in\Sigma_1}\ (\{c \mid \alpha_{\text{msd}}(a,c)\}\cap\Sigma(\alpha_{\text{msd}})\setminus\Sigma_1 \neq \emptyset \wedge \{c \mid \alpha_{\text{msd}}(b,c)\}\cap\Sigma(\alpha_{\text{msd}})\setminus\Sigma_1 \neq \emptyset) \Rightarrow \{c \mid \alpha_{\text{msd}}(a,c)\}\cap\Sigma(\alpha_{\text{msd}})\setminus\Sigma_1 = \{c \mid \alpha_{\text{msd}}(b,c)\}\cap\Sigma(\alpha_{\text{msd}})\setminus\Sigma_1$$



l.4 No two activities from different redo children have an α_{msd}-connection: $\forall_{2\leqslant i<j\leqslant n}\ \forall_{a\in\Sigma_i, b\in\Sigma_j}\ \neg\alpha_{\text{msd}}(a,b) \wedge \neg\alpha_{\text{msd}}(b,a)$



B.3 Sequence with $\times(\tau, .)$-constructs: (Definition 7.3)

Let Σ be an alphabet of activities, let α_{dfg} be a directly follows graph over Σ and let $C = (\rightarrow, S_1, \ldots S_m)$ be a $\rightarrow$-cut of α_{dfg} according to Definition 5.2. Then, a partial cut $(\rightarrow, \Sigma_1, \ldots \Sigma_n)$ is a *partial* $\rightarrow$*-cut* if there is a *pivot* Σ_p such that:

s.1 The partial cut is a consecutive part of C: $\exists_{1\leqslant s\leqslant m}\ \forall_{1\leqslant j\leqslant n}\ S_{s+j-1} = \Sigma_j$

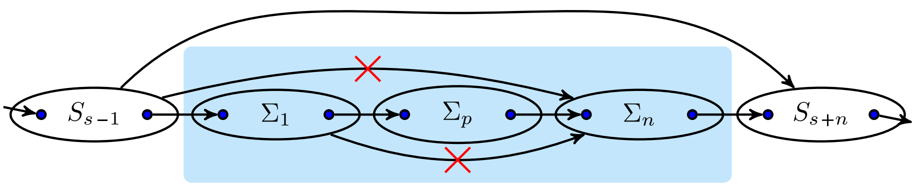
s.2 There are no end activities before the pivot in the partial cut: $\forall_{x\in[1\ldots p-1]}\ \Sigma_x \cap \text{End}(\alpha_{\text{dfg}}) = \emptyset$

s.3 There are no start activities after the pivot in the partial cut: $\forall_{x\in[p+1\ldots n]}\ \Sigma_x \cap \text{Start}(\alpha_{\text{dfg}}) = \emptyset$

s.4 There are no directly follows edges bypassing the pivot in the partial cut: $\forall_{x\in[1\ldots p-1]}\ \forall_{a\in\Sigma_x}\ \forall_{\alpha_{\text{dfg}}(a,b)}\ \exists_{y\in[1\ldots p]}\ b\in\Sigma_y$
$\forall_{y\in[p+1\ldots n]}\ \forall_{b\in\Sigma_y}\ \forall_{\alpha_{\text{dfg}}(a,b)}\ \exists_{x\in[p\ldots n]}\ a\in\Sigma_x$

s.5 The partial cut can be tightly avoided:

$$\begin{aligned} &\exists_{1\leqslant x<s, s+n\leqslant y\leqslant m}\ \exists_{a\in S_x, b\in S_y}\ \alpha_{\text{dfg}}(a,b) \\ \vee\ &\exists_{1\leqslant x<s}\ \exists_{a\in S_x}\ a\in\text{End}(\alpha_{\text{dfg}}) \\ \vee\ &\exists_{s+n\leqslant y\leqslant m}\ \exists_{b\in S_y}\ b\in\text{Start}(\alpha_{\text{dfg}}) \\ \vee\ &\epsilon\in\alpha_{\text{dfg}} \end{aligned}$$



end of definition

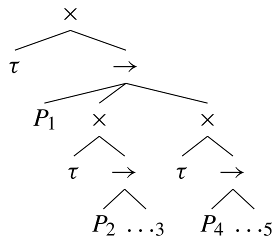
Partial cuts are considered only when close enough to the root of the process tree. We formalize this in *sequence-optional stems* (so-stems). If we consider a process tree as a graph, then the so-stem is the connected subgraph that starts at the root of the tree and that consists of $\rightarrow$- and $\times$-nodes. In addition, $\times$-nodes are only included if such nodes have two children of which one is a τ:

Definition B.1 (*sequence-optional tree and stem*) A reduced process tree $M = \oplus(M_1, \ldots M_n)$ is a *so-tree* if and only if $\oplus = \rightarrow$, or if $\oplus = \times$, $n = 2$ and $M_i = \tau$ for some $i \in [1 \ldots n]$. The *so-stem* of a reduced process tree M is the smallest set so-stem(M) with:

– if M is a so-tree, then $M \in$ so-stem(M);
– for each $\oplus(M_1, \ldots M_n) \in$ so-stem(M) and each $M_i, i \in [1 \ldots n]$ holds: if M_i is a so-tree, then $M_i \in$ so-stem(M)

(For this set, we assume that subtrees can be distinguished.)

For instance, the following tree has an so-stem, and P_1, P_2 and P_4 are non-optional non-sequential subtrees:



All subtrees shown here are sequential. Some subtrees can be skipped, however dependencies exist: if a subtree is executed, then each "P-sibling" at any higher level is executed as well. For instance, if $\ldots_3$ is executed, then P_2 is executed, as well as P_1. The challenge in preserving information in the abstraction is to *not* combine P_1 with P_2 in a partial cut without $\ldots_3$ and not to combine $\ldots_3$ and P_4 without P_2.

Lemma B.1 (Partial $\rightarrow$-cut for process trees) *Let M be a reduced process tree without duplicate activities and with an* so-stem, *and let $M' = \times(\tau, \rightarrow (M'_1, \ldots M'_n)) \in$* so-stem$(M)$. *Let S be a partition of $\Sigma(M)$ such that $\forall_{i \in [1 \ldots n]}\ \Sigma(M'_i) \in S$. Then, $(\rightarrow, \Sigma(M'_1), \ldots \Sigma(M'_n))$ is a partial $\rightarrow$-cut of $\alpha_{dfg}(M)$ over S.*

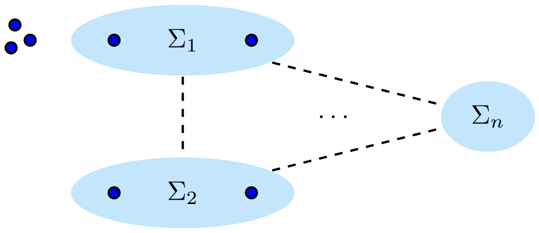
By the reduction rules, at least one of the children is not optional: $\exists_{i \in [1 \ldots n]}\ \neg\overline{?}(M_i)$. This child is the pivot. By the reduction rules, a pivot cannot be a sequential node itself. Then, this lemma follows from inspection of semantics of process trees.

B.4 Concurrent-optional-Or: ∨ (Definition 7.7)

Let Σ be an alphabet of activities, S a partition of Σ, let $\alpha_{\text{coo}}(S)$ be a coo-graph, let α_{dfg} be a directly follows graph, and let $C = (\vee, \Sigma_1, \ldots \Sigma_n)$ be a partial cut such that $\forall_{1 \leqslant i \leqslant n}\ \Sigma_i \in S$. Then, C is a partial $\vee$-cut if in $\alpha_{\text{coo}}(S)$ and α_{dfg}:

o.1 C is a part of a concurrent cut $(\wedge, X_1, \ldots X_m)$ (Definition 5.2):
$\forall_{1 \leqslant i \leqslant n}\ \exists_{1 \leqslant j \leqslant m}\ \Sigma_i = X_j$

o.2 all parts are interchangeable in $\alpha_{\text{coo}}(S)$:
$\forall_{1 \leqslant i < j \leqslant n}\ \Sigma_i\ \overline{\vee}\ \Sigma_j$



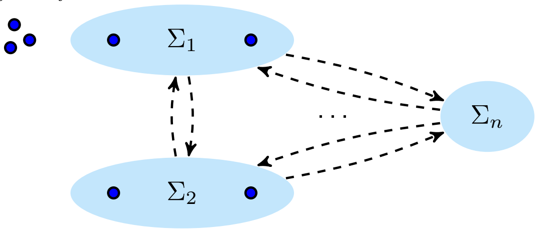
B.5 Concurrent-optional-Or: ∧ (Definition 7.8)

Let Σ be an alphabet of activities, S a partition of Σ, let $\alpha_{\text{coo}}(S)$ be a coo-graph, let α_{dfg} be a directly follows graph, and let $C = (\wedge, \Sigma_1, \ldots \Sigma_n)$ be a partial cut such that $\forall_{1 \leqslant i \leqslant n}\ \Sigma_i \in S$. Then, C is a partial $\wedge$-cut if in $\alpha_{\text{coo}}(S)$ and α_{dfg}:

c.1 C is a part of a concurrent cut $(\wedge, X_1, \ldots X_m)$ (Definition 5.2):
$\forall_{1 \leqslant i \leqslant n}\ \exists_{1 \leqslant j \leqslant m}\ \Sigma_i = X_j$

Furthermore, in $\alpha_{\text{coo}}(S)$:
– EITHER –

c.2.1 all parts bi-imply one another:
$\forall_{1 \leqslant i < j \leqslant n}\ \Sigma_i \overline{\Rightarrow} \Sigma_j \wedge \Sigma_j \overline{\Rightarrow} \Sigma_i$



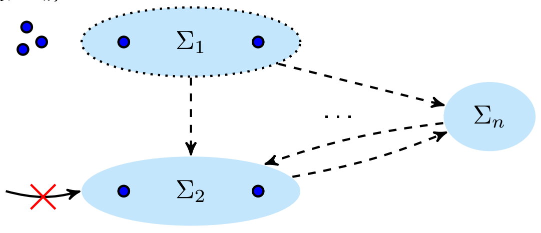
– OR –

c.3.1 the first part is optional:
$\overline{?}\Sigma_1$

c.3.2 The first part implies the other parts:
$\forall_{i \in [2 \ldots n]}\ \Sigma_1 \overline{\Rightarrow} \Sigma_i$

c.3.3 All non-first parts bi-imply one another:
$\forall_{i,j \in [2 \ldots n] \wedge i \neq j}\ \Sigma_i \overline{\Rightarrow} \Sigma_j$

c.3.4 No non-first part Σ_i is implied by any part not in C:
$\neg\exists_{i\in[2\ldots n]\wedge\Sigma'\in S\setminus\{\Sigma_1,\ldots\Sigma_n\}}\ (\Sigma_i \not\Rrightarrow \Sigma' \wedge \Sigma' \Rrightarrow \Sigma_i)$



end of definition

Similar to so-stems, when considering a process tree M as a graph, the *coo-stem* of M is the connected subgraph starting at the root of M consisting of $\wedge$-, $\vee$-, and $\times$ nodes (the latter ones only if they have two children of which one is a τ).

Definition B.2 (*concurrent-optional-or tree and stem*) A reduced process tree $M = \oplus(M_1,\ldots M_n)$ is a *coo-tree* if and only if $\oplus \in \{\wedge,\vee\}$, or if $\oplus = \times$, $n = 2$ and $M_i = \tau$ for some $i \in [1\ldots n]$. The *coo-stem* of a reduced process tree M is the smallest set coo-stem(M) with:

- if M is a coo-tree, then $M \in$ coo-stem(M);
- for each $\oplus(M_1,\ldots M_n) \in$ coo-stem(M) and each $M_i, i \in [1\ldots n]$ holds: if M_i is a coo-tree, then $M_i \in$ coo-stem(M)

(For this set, we assume that subtrees can be distinguished.)

Lemma B.2 (partial $\wedge$- and $\vee$-cuts for process trees) *Let M be a reduced process tree without duplicate activities and with an* coo-stem, *and let $M' = \oplus(M'_1,\ldots M'_n) \in$* coo-stem, *with $\oplus \in \{\wedge,\vee\}$. Let S be a partition of $\Sigma(M)$ such that $\forall_{i\in[1\ldots n]}\ \Sigma(M'_i) \in S$. Then, $(\oplus, \Sigma(M'_1),\ldots \Sigma(M'_n))$ is a partial $\oplus$-cut of the coo-graph $\alpha_{coo}(\mathcal{L}(M), S)$.*

This lemma follows from inspection of the semantics of process trees.

C Class counterexamples

C.1 $\mathrm{C}_{\mathbf{dfg}}$

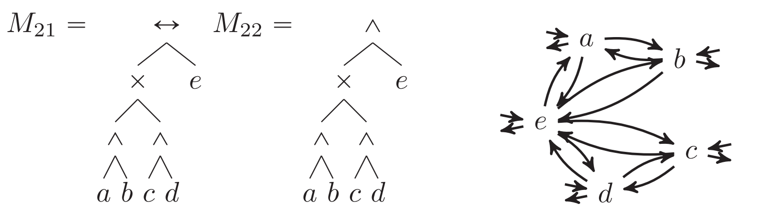
See Figs. 14, 15 and 16.

D Uniqueness for $\alpha_{\mathbf{dfg}}$

D.1 Lemma 5.2

Lemma: Take two reduced process trees of $\mathrm{C}_{\mathrm{dfg}}$ $K = \oplus(K_1,\ldots K_n)$ and $M = \otimes(M_1,\ldots M_m)$ such that $\oplus \neq \otimes$. Then, $\alpha_{\mathrm{dfg}}(K) \neq \alpha_{\mathrm{dfg}}(M)$.

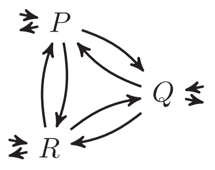
Proof Towards contradiction, assume that $\alpha_{\mathrm{dfg}}(K) = \alpha_{\mathrm{dfg}}(M)$. By the reduction rules of Sect. 4, $n \geqslant 2$ and $m \geqslant 2$. Perform case distinction on $\oplus$ to prove that $\alpha_{\mathrm{dfg}}(K) \neq \alpha_{\mathrm{dfg}}(M)$.



(a) Language-different process trees. **(b)** $\alpha_{\text{dfg}}(M_{21}) = \alpha_{\text{dfg}}(M_{22})$

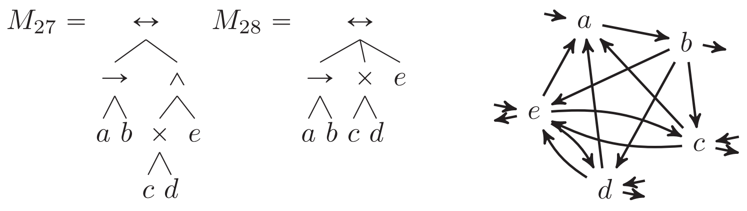
Fig. 14 An example for the necessity of Requirement C_{dfg}.i.1: the trees have different languages but equivalent directly follows graphs. These trees differ in their root operator: activity e can be executed before and after all other activities, making the difference between interleaved and concurrent invisible

$$M_{23} = \leftrightarrow(a, \leftrightarrow(b, c))$$
$$M_{24} = \leftrightarrow(b, \leftrightarrow(c, a))$$
$$M_{25} = \leftrightarrow(c, \leftrightarrow(b, a))$$
$$M_{26} = \leftrightarrow(a, b, c)$$



(a) Language-different process trees. **(b)** $\alpha_{\text{dfg}}(M_{23}) = \alpha_{\text{dfg}}(M_{24}) = \alpha_{\text{dfg}}(M_{25}) = \alpha_{\text{dfg}}(M_{26})$

Fig. 15 An example for the necessity of Requirement C_{dfg}.i.2: four trees having different languages but equivalent directly follows graphs. The difference between these trees is "semi-long-dependencies", e.g. in M_{23}, a cannot be executed between b and c, and such dependencies cannot be captured by a directly follows relation



(a) Language-different process trees. **(b)** $\alpha_{\text{dfg}}(M_{27}) = \alpha_{\text{dfg}}(M_{28})$

Fig. 16 An example for the necessity of Requirement C_{dfg}.i.3. Activity e witnesses ambiguity: e can be concurrent to $\times(c, d)$ (M_{27}) or interleaved to all other activities (M_{28})

$\oplus = \times$ By semantics of the $\times$ operator and the reduction rules, there exist at least n unconnected parts in $\alpha_{\text{dfg}}(K)$ (see Lemma 5.1). As $\otimes \neq \times$ and by the semantics of the other operators, $\alpha_{\text{dfg}}(M)$ is connected, so $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\oplus = \rightarrow$ By semantics of the $\rightarrow$ operator, $\alpha_{\text{dfg}}(K)$ is a chain of at least n clusters (see Lemma 5.1). As $\otimes \neq \rightarrow$ and by the semantics of the other operators, $\alpha_{\text{dfg}}(M)$ is not a chain (for $\times$, the graph is not connected while for $\leftrightarrow$, $\wedge$ and $\circlearrowleft$, the graph is a clique), so $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\oplus = \wedge$ By semantics of the $\wedge$ operator, $\alpha_{\text{dfg}}(K)$ consists of at least n fully interconnected clusters (see Lemma 5.1). Perform case distinction on the (due to symmetry) remaining cases of $\otimes$:

$\otimes = \circlearrowleft$ We try to construct a concurrent cut $(\wedge, \Sigma_1, \ldots \Sigma_p)$ for M. Take an activity $a \in \text{Start}(M_1)$. By Requirement C_{dfg}.1.1, $a \notin \text{End}(M_1)$. Take an activity $b \in \Sigma(M) \setminus \Sigma(M_1)$. Then, by semantics of $\circlearrowleft$, $\neg\alpha_{\text{dfg}}(a, b)$ and by Requirement c.2, a and b are part of the same Σ in the cut we are constructing, e.g. Σ_1. This holds for all a and b, thus $\Sigma(M) = \Sigma_1$. Hence, there is no non-trivial concurrent cut, and $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.
$\otimes = \leftrightarrow$ By C_{dfg}, $\exists_{1 \leqslant i \leqslant n}\ \exists_{a \in \Sigma(M_i)}\ a \notin \text{Start}(M_i) \vee a \notin \text{End}(M_i)$. Take such an M_i and a. As either $a \notin \text{Start}(M_i)$ or $a \notin \text{End}(M_i)$, there is no connection to/from a to any other subtree, i.e. $\forall_{1 \leqslant j \leqslant n, j \neq i}\ \forall_{b \in \Sigma(M_j)}\ \neg\alpha_{\text{dfg}}(b, a) \vee \neg\alpha_{\text{dfg}}(a, b)$. If we would construct a concurrent cut $(\wedge, \Sigma_1 \ldots \Sigma_p)$, then both a and all such b's would be in the same Σ, e.g. $\{a\} \cup (\Sigma(K) \setminus \Sigma(M_j)\}) \subseteq \Sigma_1$. This holds for all activities of $\text{Start}(M_i)$ and $\text{End}(M_i)$. Hence, if we would construct a concurrent cut, all $\text{Start}(K)$ and $\text{End}(K)$ activities would be part of the same Σ. Therefore, there cannot be a non-trivial concurrent cut for K, and hence, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\oplus = \circlearrowleft$ Perform case distinction on the remaining case of $\otimes$:

$\otimes = \leftrightarrow$ We try to construct a loop cut $(\circlearrowleft, \Sigma_1, \ldots \Sigma_n)$. Consider a child M_i, and an activity s from the start activities of another child. Moreover, consider a path $\alpha_{\text{dfg}}(a_1, a_2) \ldots \alpha_{\text{dfg}}(a_{p-1}, a_p)$ such that all activities on the path are in $\Sigma(M_i)$, and $a_1 \in \text{Start}(M_i)$ and $a_p \in \text{End}(M_i)$. By Lemma 5.1, $a_1 \in \Sigma_1 \wedge a_p \in \Sigma_1$. Consider activity a_2. If $a_2 \in \text{Start}(M_i)$, then $a_2 \in \Sigma_1$. If $a_2 \in \text{End}(M_i)$, then $a_2 \in \Sigma_1$. If $a_2 \notin \text{Start}(M_i) \wedge a_2 \notin \text{End}(M_i)$, then by the semantics of $\leftrightarrow$, $\neg\alpha_{\text{dfg}}(s, a_2)$. If a_2 would be in Σ_2, as it has a connection $\alpha_{\text{dfg}}(a_1, a_2)$, by the semantics of $\circlearrowleft$ there should be a connection $\alpha_{\text{dfg}}(s, a_2)$. Thus, $a_2 \in \Sigma_1$. This argument holds for the entire path, and by construction of $\alpha_{\text{dfg}}(M)$ each activity is on such a path; thus, $\Sigma(M_i) \subseteq \Sigma_1$. This holds for all children M_i, so there cannot be a non-trivial loop cut. Hence, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

As these arguments are symmetric in $\oplus$ and $\otimes$, we conclude that $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$. □

D.2 Lemma 5.3

Lemma: Take two reduced process trees of C_{dfg} $K = \oplus(K_1 \ldots K_n)$ and $M = \oplus(M_1 \ldots M_m)$ such that their activity partition is different: $\exists_{1 \leqslant w \leqslant \min(n,m)}\ \Sigma(K_w) \neq \Sigma(M_w)$. Then, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

Proof Without loss of generality, we assume that children of the non-commutative operators $(\rightarrow, \circlearrowleft)$ have a fixed order. Towards contradiction, assume that $\alpha_{\text{dfg}}(K) = \alpha_{\text{dfg}}(M)$. Perform case distinction on $\oplus$ (the case for K and M swapped is symmetric):

$\oplus = \times$ Take a pair of activities a, b such that $a \in \Sigma(K_x)$, $a \in \Sigma(M_y)$, $b \in \Sigma(K_x)$ and $b \notin \Sigma(M_y)$ (choose x and y as desired). Obviously, if the activity partitions of K and M are different such a pair exists. By the reduction rules, no child $K_1 \ldots K_n$ is an exclusive-choice subtree itself, and by semantics of the other operators there is an undirected path in $\alpha_{\text{dfg}}(K)$, i.e. $a \leftrightsquigarrow b$ in $\alpha_{\text{dfg}}(K)$. However, as $a \in \Sigma(M_y) \wedge b \notin \Sigma(M_y)$, $a \not\leftrightsquigarrow b$ in $\alpha_{\text{dfg}}(M)$. Hence, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.
$\oplus = \rightarrow$ Take $a \in \Sigma(K_i)$ and $b \in \Sigma(K_j)$ such that $i < j$. Then by the $\rightarrow$-cut, $\alpha_{\text{dfg}}^{+}(a, b) \wedge \neg\alpha_{\text{dfg}}^{+}(b, a)$. By the reduction rules, all children of K and M are not $\rightarrow$-nodes themselves, thus, by the semantics of the other operators ($\times$ is unconnected, $\wedge$ and $\circlearrowleft$ are strongly connected), either $\neg\alpha_{\text{dfg}}(a, b)$ or $\alpha_{\text{dfg}}^{+}(b, a)$. Then, $a \in \Sigma(M_x) \wedge b \in \Sigma(M_y)$

with $x < y$. This holds for all such a and b, hence $\forall_{1 \leqslant i \leqslant n=m}\ \Sigma(K_i) = \Sigma(M_i)$, which contradicts the initial assumption.

$\oplus = \wedge$ To prove the equality of the activity partitions, we consider two symmetrical directions: a) if two activities are in the same Σ_i in K, then they are in the same Σ_i in M. b) if two activities are in the same Σ_i in M, then they are in the same Σ_i in K. Consider a child M_x. Perform case distinction on the structure of M_x:

$M_x = a$ A single activity cannot be split. Thus, $\Sigma(K_x) \subseteq \Sigma(M_x)$.
$M_x = \times(M_{x_1}, \ldots M_{x_p})$ Take two activities $a \in \Sigma(M_{x_1})$ and $b \in \Sigma(M_{x_2})$. By semantics of $\times$, $\neg\alpha_{\text{dfg}}(a, b)$. Thus, in a concurrent cut, a and b should be part of the same Σ. This holds for all such activities of all children of M_x; thus, $\Sigma(K_x) \subseteq \Sigma(M_x)$.
$M_x = \rightarrow(M_{x_1}, \ldots M_{x_p})$ Similar, using that either $\neg\alpha_{\text{dfg}}(a, b)$ or $\neg\alpha_{\text{dfg}}(b, a)$.
$M_x = \wedge(M_{x_1}, \ldots M_{x_p})$ Excluded by the reduction rules.
$M_x = \circlearrowleft(M_{x_1}, \ldots M_{x_p})$ By C_{dfg}, there is at least one child M_{x_i} such that $\mathrm{Start}(M_{x_i}) \cap \mathrm{End}(M_{x_i}) = \emptyset$. Take such a M_{x_i} and an a from $\Sigma(M_{x_i})$. Furthermore, take b from any other child. There are three cases for a: $a \notin \mathrm{Start}(M_{x_i})$, $a \notin \mathrm{End}(M_{x_i})$ or both. For all these three cases, $\neg\alpha_{\text{dfg}}(a, b) \vee \neg\alpha_{\text{dfg}}(b, a)$. Thus, by argumentation similar to the $\times$ case, $\Sigma(K_x) \subseteq \Sigma(M_x)$.
$M_x = \leftrightarrow(M_{x_1}, \ldots M_{x_p})$ Similar to the $\circlearrowleft$ case.

Hence, $\Sigma(K_x) \subseteq \Sigma(M_x)$. This holds for all $\Sigma(M_x)$ and by symmetry for all $\Sigma(K_x)$. Hence, $\forall_{1 \leqslant i \leqslant n}\ \Sigma(K_i) = \Sigma(M_i)$, which contradicts the initial assumption.

$\oplus = \circlearrowleft$ Consider $\Sigma(K_i)$ for some $2 \leqslant i \leqslant n$. By the reduction rules, K_i is of the form $\times(\ldots)$. By semantics of the other operators, for all $a, b \in \Sigma(K_i)$, there exists an undirected path $a \leftrightsquigarrow b$ in $\alpha_{\text{dfg}}(K)$, such that all activities on this undirected path are in K_i. Between all the activities on this path, there exists a connection in $\alpha_{\text{dfg}}(K_i)$, and none of the activities on this path is in $\mathrm{Start}(K)$ or $\mathrm{End}(K)$. By Lemma 5.1, in a non-trivial loop cut, (without loss of generality) $\Sigma(K_i) \subseteq \Sigma(M_i)$.
Let $K_1 = \otimes(K_{1_1}, \ldots K_{1_p})$. Perform case distinction on $\otimes$:
$\otimes = \times$ Take a child K_{1_i}. By the reduction rules, this child is not an $\times$. For all activities $a \in \mathrm{Start}(K_{1_i})$, $b \in \mathrm{End}(K_{1_i})$, there exist a directed path $\alpha_{\text{dfg}}^{+}(a, b)$, such that this path is completely in $\Sigma(K_{1_i})$. Furthermore, take an activity $c \in \mathrm{End}(K_{1_{j \neq i}})$. By semantics of $\times$, c has no directly follows connection to any node on the path. Towards contradiction, assume there is a first node d on the path $\notin \Sigma(M_1)$. Then, by semantics of $\circlearrowleft$, there should be a connection $\alpha_{\text{dfg}}(c, d)$. This holds for all activities d and children i, so $\Sigma(K_1) \subseteq \Sigma(M_1)$.
$\otimes = \rightarrow$ Similar to the $\times$-case.
$\otimes = \wedge$ $\mathrm{Start}(K) \cup \mathrm{End}(K) \subseteq \Sigma(M_1)$, thus we only need to consider non-start non-end activities. Take such an activity a in child K_{1_i}, and take an activity $b \in \mathrm{End}(K_{1_{j \neq i}})$. By semantics of $\wedge$, $\alpha_{\text{dfg}}(a, b)$; by C_{dfg}, $b \notin \mathrm{Start}(K_1)$; thus by Lemma 5.1, $a \in \Sigma(M_1)$. This holds for all a, so $\Sigma(K_1) \subseteq \Sigma(M_1)$.
$\otimes = \circlearrowleft$ Excluded by the reduction rules.
$\otimes = \leftrightarrow$ Similar to the $\times$-case.

$\oplus = \leftrightarrow$ Take a w such that $\Sigma(K_w) \neq \Sigma(M_w)$ and let $K_w = \otimes(K_{w_1} \ldots K_{w_p})$. Perform case distinction on $\otimes$:

$\otimes = \times$ By semantics of $\times$, no end activity of K_{w_1} has a connection to any start activity of any other K_{w_j}. Thus, as M contains an interleaved activity partition, $\Sigma(K_w) \subseteq \Sigma(M_w)$.
$\otimes = \rightarrow$ Similar to the $\times$ case.
$\otimes = \wedge$ By C_{dfg}, at least one child of K_w has disjoint start and end activities. Take such a child K_{w_y}, and consider two activities: $a \notin \text{Start}(K_{w_y})$ and $b \in \Sigma(K_w) \setminus K_{w_y}$. By semantics of $\wedge$, $\alpha_{\text{dfg}}(b, a)$. Then, by Lemma 5.1, $a \in \Sigma(M_w)$ and $b \in \Sigma(M_w)$. This holds for all b and by symmetry for $\text{Start}(K_{w_y}) \cup \text{End}(K_{w_y})$. By semantics of $\leftrightarrow$, non-start non-end activities only have connections with start/end activities of K_w. Therefore, $\Sigma(K_w) \setminus (\text{Start}(K_w) \cup \text{End}(K_w)) \subseteq \Sigma(M_w)$. Hence, $\Sigma(K_w) \subseteq \Sigma(M_w)$.
$\otimes = \circlearrowleft$ By semantics of $\leftrightarrow$, non-start non-end activities only have connections with start/end activities of K_w. Therefore, $\Sigma(K_w) \setminus (\text{Start}(K_w) \cup \text{End}(K_w)) \subseteq \Sigma(M_w)$. All activities $\in \text{Start}(K_w) \cup \text{End}(K_w)$ have connections from/to $\text{End}(K_{w_2}) \cup \text{Start}(K_{w_2})$, thus $\text{Start}(K_w) \cup \text{End}(K_w) \subseteq \Sigma(M_w)$. Hence, $\Sigma(K_w) \subseteq \Sigma(M_w)$.
$\otimes = \leftrightarrow$ Excluded by C_{dfg}.

By contradiction, we conclude $\alpha_{\text{dfg}} K \neq \alpha_{\text{dfg}} M$. □

E Uniqueness for C_{msd}

Lemma E.1 (Operators are mutually exclusive) *Take two reduced process trees of* C_{msd} $K = \oplus(K_1, \dots K_n)$ *and* $M = \otimes(M_1, \dots M_m)$ *such that* $\oplus \neq \otimes$. *Then,* $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$ *or* $\alpha_{msd}(K) \neq \alpha_{msd}(M)$.

This proof of this lemma is similar to the proof of Lemma 5.2: for each combination of $\oplus$ and $\otimes$, a difference in α_{msd}-graphs is shown. For a detailed proof, please refer to “Appendix E.2”.

Lemma E.2 (Partitions are mutually exclusive) *Take two reduced process trees of* C_{dfg} $K = \oplus(K_1 \dots K_n)$ *and* $M = \oplus(M_1 \dots M_m)$ *such that their activity partition is different:* $\exists_{1 \leqslant w \leqslant min(n,m)}\ \Sigma(K_w) \neq \Sigma(M_w)$. *Then,* $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$ *or* $\alpha_{msd}(K) \neq \alpha_{msd}(M)$.

This proof of this lemma is similar to the proof of Lemma 5.3: for each $\oplus$, it is shown that a difference in partitions must lead to a difference in α_{msd}-graphs. For a detailed proof, please refer to “Appendix E.3”.

Lemma E.3 (Abstraction uniqueness for C_{msd}) *Take two reduced process trees of class* C_{msd}: $K = \oplus(K_1, \dots K_n)$ *and* $M = \otimes(M_1, \dots M_m)$. *Then,* $K = M$ *if and only if* $\alpha_{dfg}(K) = \alpha_{dfg}(M)$ *and* $\alpha_{msd}(K) = \alpha_{msd}(M)$.

The proof of this lemma is similar to the proof of Lemma 5.4, using Lemmas E.1 and E.2.

Corollary E.1 (Language uniqueness for C_{msd}) *There are no two different reduced process trees of* C_{msd} *with equal languages. Hence, for trees of class* C_{msd}, *the normal form of Sect. 4 is uniquely defined.*

E.1 LC-property

The minimum self-distance graph possesses more expressive power than the footprints of Definition 6.3 utilize. That is, there exist pairs of process trees that have different normal forms, languages and α_{msd}-graphs, but that the footprints do not distinguish.

For instance, consider the trees

$$\begin{tikzcd}[column sep=tiny, row sep=small]
M_{12} = & & & \circlearrowleft \arrow[dl, dash] \arrow[dr, dash] & \\
& & \wedge \arrow[dl, dash] \arrow[dr, dash] & & d \\
& \circlearrowleft \arrow[dl, dash] \arrow[dr, dash] & & c & \\
a & & b & &
\end{tikzcd}$$

and

$$\begin{tikzcd}[column sep=tiny, row sep=small]
M_{13} = & & & \circlearrowleft \arrow[dl, dash] \arrow[dr, dash] & & . \\
& & \wedge \arrow[dl, dash] \arrow[dr, dash] & & b & \\
& \circlearrowleft \arrow[dl, dash] \arrow[dr, dash] & & a & & \\
c & & d & & &
\end{tikzcd}$$

These trees have a different language, an equivalent α_{dfg}-graph (shown in Fig. 9b) but a different α_{msd}-graph (shown in Fig. 9c, d). Thus, they could be distinguished using their α_{msd}-graph.

However, the footprint (Definition 6.3) cannot distinguish these trees: both cuts $(\circlearrowleft, \{a, b, c\}, \{d\})$ and $(\circlearrowleft, \{a, c, d\}, \{b\})$ are valid in both α_{msd}-graphs, where $(\circlearrowleft, \{a, b, c\}, \{d\})$ corresponds to M_{12} and $(\circlearrowleft, \{a, c, d\}, \{b\})$ corresponds to M_{13}. This implies that a discovery algorithm that uses only the footprint cannot distinguish these two trees.

This problem occurs in certain nestings of loops and concurrent operators, as indicated in the proof of Lemma E.2. We formalize this remaining problem as a *loop-concurrency property* (LC-property). An LC-property could distinguish the specific nesting using only the α_{msd}-graph.

Definition E.1 (*LC-property*) Let $K, M \in \mathrm{C}_{\text{msd}}$ be process trees in normal form such that

$$\begin{tikzcd}[column sep=small, row sep=small]
K = & & \circlearrowleft \arrow[dl, dash] \arrow[d, dash] \arrow[drr, dash] & & & , \\
& K_1 = \wedge \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & K_2 & \dots & K_n & \\
K_{1,1} & \dots & K_{1,p} & & &
\end{tikzcd}$$

$$\begin{tikzcd}[column sep=small, row sep=small]
M = & & \circlearrowleft \arrow[dl, dash] \arrow[d, dash] \arrow[drr, dash] & & & , \\
& M_1 = \wedge \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & M_2 & \dots & M_n & \\
M_{1,1} & \dots & M_{1,q} & & &
\end{tikzcd}$$

and $\alpha_{\text{dfg}}(K) = \alpha_{\text{dfg}}(M)$. Then, an *LC-property LC* is a function that distinguishes the cuts of K and M in their minimum self-distance graphs, i.e. $LC(\alpha_{\text{msd}}(K)) \wedge LC(\alpha_{\text{msd}}(M))$ if and only if the cut $(\circlearrowleft, \Sigma(K_1), \dots \Sigma(K_n))$ conforms to both K and M.

Consider $\mathrm{C}_{\text{msd}}{}'$ to be the class of trees C_{msd} where arbitrary nestings of $\circlearrowleft$ and $\wedge$ are allowed, that is, Requirement C_{msd}.1.1 is dropped. Then, if an LC-property exists, Lemma E.2 applies for $\mathrm{C}_{\text{msd}}{}'$.

We did not find an LC-property, but we also did not prove that it cannot exist. A proof that an LC-property cannot exist would, for instance, be the existence of an example of two process trees of $\mathrm{C_{msd}}'$ in normal form with equivalent α_{msd}-graphs. We did not find such examples in an extensive search, so we conjecture that an LC-property exists:

Conjecture E.1 (LC-property) *There exists an LC-property (Definition E.1).*

E.2 Lemma E.1

Lemma: Take two reduced process trees of $\mathrm{C_{msd}}$ $K = \oplus(K_1, \ldots K_n)$ and $M = \otimes(M_1, \ldots M_m)$ such that $\oplus \neq \otimes$. Then, $\alpha_{\mathrm{dfg}}(K) \neq \alpha_{\mathrm{dfg}}(M)$ or $\alpha_{\mathrm{msd}}(K) \neq \alpha_{\mathrm{msd}}(M)$.

Proof Towards contradiction, assume $\alpha_{\mathrm{dfg}}(K) = \alpha_{\mathrm{dfg}}(M)$ and $\alpha_{\mathrm{msd}}(K) = \alpha_{\mathrm{msd}}(M)$. We only consider the cases that were not covered in the proof of Lemma 5.2.

$\oplus = \wedge$ and $\otimes = \circlearrowleft$. We try to construct a concurrent cut $\Sigma_1 \ldots \Sigma_q$ for M. By Requirement c.1, every such Σ_i must have a start and an end activity. Thus, we only need to prove that $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1) \subseteq \Sigma_1$. Perform case distinction on M_1:

$M_1 = \times(M_{1_1}, \ldots M_{1_p})$ Each $a \in \Sigma(M_{1_i})$ has no α_{dfg}-connection to any activity in $\Sigma(M_{j \neq i})$. Therefore, $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1) \subseteq \Sigma_1$.
$M_1 = \rightarrow(M_{1_1}, \ldots M_{1_p})$ Each $a \in \Sigma(M_{1_i})$ has no α_{dfg}-connection to any activity in $\Sigma(M_{j<i})$. Therefore, $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1) \subseteq \Sigma_1$.
$M_1 = \wedge(\ldots)$ Consider three cases:

- If any of the $M_{2 \leqslant i \leqslant p}$ contains a $\circlearrowleft$, consider an activity a in the redo of that $\circlearrowleft$. By semantics of $\circlearrowleft$, there is no α_{dfg}-connection between a and any activity in $\Sigma(M_1)$. Therefore, $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1) \subseteq \Sigma_1$.
- If none of the $M_{2 \leqslant i \leqslant p}$ contains a $\circlearrowleft$ and M_1 does not contain a $\circlearrowleft$, then the α_{msd}-graph is connected, and therefore, by Requirement ci.1, $\Sigma(M) \subseteq \Sigma_1$.
- If none of the $M_{2 \leqslant i \leqslant p}$ contains a $\circlearrowleft$ and M_1 contains a $\circlearrowleft$, then consider an activity a under a redo of any such $\circlearrowleft$, and any activity $b \in \Sigma(M_{2 \leqslant i \leqslant m})$. By semantics of $\circlearrowleft$, $\neg\alpha_{\mathrm{dfg}}(a, b)$ and $\neg\alpha_{\mathrm{dfg}}(b, a)$, thus a and b must be in the same Σ_1. All activities $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1)$ have at least an α_{msd}-connection with at least some activity in the redo of a $\circlearrowleft$. Thus, by Requirement ci.1, $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1) \subseteq \Sigma_1$.

$M_1 = \circlearrowleft(\ldots)$ Excluded by $\mathrm{C_{msd}}$.
$M_1 = \leftrightarrow(\ldots)$ By $\mathrm{C_{msd}}$, there exists a child M_{1_i} such that $\mathrm{Start}(M_{1_i}) \cap \mathrm{End}(M_{1_i}) = \emptyset$. Thus, all activities in $\mathrm{End}(M_{1_{j \neq i}})$ have no α_{dfg}-connection to $\mathrm{End}(M_{1_i})$, and similarly for the activities of $\mathrm{Start}(M_{1_j})$. Therefore, $\mathrm{Start}(M_1) \cup \mathrm{End}(M_1) \subseteq \Sigma_1$.

Hence, there is no concurrent cut in M, and therefore, $\alpha_{\mathrm{dfg}}(K) \neq \alpha_{\mathrm{dfg}}(M)$.
$\oplus = \circlearrowleft$ and $\otimes = \leftrightarrow$. No change is necessary. □

E.3 Lemma E.2

Lemma: Take two reduced process trees of $\mathrm{C_{dfg}}$ $K = \oplus(K_1 \ldots K_n)$ and $M = \oplus(M_1 \ldots M_m)$ such that their activity partition is different: $\exists_{1 \leqslant w \leqslant \min(n,m)}\ \Sigma(K_w) \neq \Sigma(M_w)$. Then, $\alpha_{\mathrm{dfg}}(K) \neq \alpha_{\mathrm{dfg}}(M)$ or $\alpha_{\mathrm{msd}}(K) \neq \alpha_{\mathrm{msd}}(M)$.

Proof Towards contradiction, assume that $\oplus = \otimes$, $\alpha_{\mathrm{dfg}}(K) = \alpha_{\mathrm{dfg}}(M)$, $\alpha_{\mathrm{msd}}(K) = \alpha_{\mathrm{msd}}(M)$ and that there is a w such that $\Sigma(K_w) \neq \Sigma(M_w)$. We only consider the cases that were not covered in the proof of Lemma 5.2.

$\oplus = \wedge$ and $M_x = \circlearrowleft (M_{x_1}, \ldots M_{x_p})$. Try to construct a $\wedge$-cut and prove that $\Sigma(M_x) \subseteq \Sigma_x$. Consider three cases:

- If any of the $M_{x_{2 \leqslant i \leqslant p}}$ contains a $\circlearrowleft$, consider an activity a in the redo of that $\circlearrowleft$. By semantics of $\circlearrowleft$, there is no α_{dfg}-connection between a and any activity in $\Sigma(M_{x_1})$. Therefore, $\Sigma(M_{x_1}) \subseteq \Sigma_x$. This holds for all such a, thus all such redo activities are in Σ_x. Consider all remaining activities, i.e. $b \in \Sigma(M_{x_{j \neq i}})$ such that b is in no other $\circlearrowleft$-redo than M_x. For each of these activities b, there is a α_{msd}-relation with an activity in Σ_{x_1} or an activity such as a. Thus, $\Sigma(M_x) \subseteq \Sigma_x$.
- If none of the $M_{x_{2 \leqslant i \leqslant p}}$ contains a $\circlearrowleft$ and M_{x_1} does not contain a $\circlearrowleft$, then the α_{msd}-graph is connected, and therefore, $\Sigma(M_x) \subseteq \Sigma_x$.
- If none of the $M_{x_{2 \leqslant i \leqslant p}}$ contains a $\circlearrowleft$ and M_{x_1} contains a $\circlearrowleft$, then consider an activity a under a redo of any such $\circlearrowleft$, and any activity $b \in \Sigma(M_{x_{2 \leqslant i \leqslant m}})$. By semantics of $\circlearrowleft$, $\neg\alpha_{\text{dfg}}(a, b)$ and $\neg\alpha_{\text{dfg}}(b, a)$, thus a and b must be in the same Σ_x. All activities in $\Sigma(M_{x_1})$ have at least an α_{msd}-connection with at least some activity in the redo of a $\circlearrowleft$, Thus, $\Sigma(M_x) \subseteq \Sigma_x$.

$\oplus = \circlearrowleft$ and $K_1 = \wedge(K_{1,1}, \ldots K_{1,p})$. Try to construct a $\circlearrowleft$-cut and prove that $\Sigma(K_1) \subseteq \Sigma_1$. By semantics of $\circlearrowleft$, $\text{Start}(K_1) \cup \text{End}(K_1) \subseteq \Sigma_1$. Take an activity $a \in \Sigma(K_1)$, such that $a \notin \text{Start}(K_1) \cup \text{End}(K_1)$, and take another $b \in \bigcup_{1 \leqslant i \leqslant n} \Sigma(K_i)$ such that $b \in \text{Start}(K_1) \cup \text{End}(K_1)$. Then, $b \in \Sigma_1$. Perform case distinction on b:

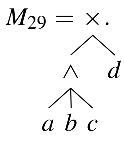
$b \notin \text{End}(K_1)$ Then, $\alpha_{\text{dfg}}(b, a)$ and thus $a \in \Sigma_1$.
$b \notin \text{Start}(K_1)$ Then, $\alpha_{\text{dfg}}(a, b)$ and thus $a \in \Sigma_1$.
$b \in \text{Start}(K_1) \cap \text{End}(K_1)$ Excluded by C_{msd}. (Here, the LC-property (Conjecture E.1) would detect and guarantee $a \in \Sigma_1$). □

F Uniqueness for α_{coo}

Given a particular language, several partial cuts might apply. In this section, we prove that each of these partial cuts is "correct", that is, corresponds to the process tree from which the language was derived. We first formalize this concept, after which we use it to prove uniqueness for so- and coo-stems.

Definition F.1 (*partition correspondence*, $\mathbb{M}^{\Sigma}$) Let M be a reduced process tree without duplicate activities. Then, $\mathbb{M}^{\Sigma}(M)$ denotes the set of all partitions S that *correspond* to M. That is, let $S = \{S_1, \ldots S_n\}$ be a partition of $\Sigma(M)$, then $S \in \mathbb{M}^{\Sigma}(M)$ if and only if there exists an M' such that M' reduces to M using only the associativity rules ($\text{A}_{\times}$, $\text{A}_{\rightarrow}$, $\text{A}_{\wedge}$, $\text{A}_{\vee}$, $\text{A}_{\circlearrowleft\text{B}}$, $\text{A}_{\circlearrowleft\text{R}}$); and for every $S_i \in S$, there is a subtree M'' of M' such that $S_i = \Sigma(M'')$.

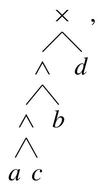
Intuitively, a partition corresponds to a process tree if each set of activities in the partition can be mapped to a node in the process tree (up to the associativity rules). For instance, let



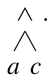
Then,

$$\begin{aligned}\mathbb{M}^{\Sigma}(M_{29}) = \{&\{\{a\},\{b\},\{c\},\{d\}\}, && \{\{a,b\},\{c\},\{d\}\}, && \{\{a\},\{b,c\},\{d\}\},\\ &\{\{a,c\},\{b\},\{d\}\}, && \{\{a,b,c\},\{d\}\}, && \{\{a,b,c,d\}\}\}\end{aligned}$$

Each of these partitions corresponds to M_{29}. For instance, consider the partition $\{\{a,c\},\{b\},\{d\}\}$. Each of the sets in this partition can be mapped on a node in the tree



which reduces to M_{29} using the associativity rules. In this mapping, $\{d\}$ is mapped on the leaf d, $\{b\}$ is mapped on the leaf b, and $\{a,c\}$ is mapped on the node



In contrast, the partition $\{\{a,b\},\{c,d\}\}$ does not correspond to M_{29}, as c and d cannot be mapped together without a and b.

F.1 SO-stems

In Lemma B.1, we showed that a tree with an $\times(\tau, \rightarrow(\ldots))$ structure has a partial $\rightarrow$-cut. Now, we prove the opposite: there can only be a partial $\rightarrow$-cut if the tree has such a structure.

Lemma F.1 (Partitions are mutually exclusive for so-stems) *Let $M = \rightarrow(M_1, \ldots M_m)$, $K = \rightarrow(K_1, \ldots K_n)$ be reduced trees of C_{coo} such that their activity partition is different, i.e. there is a $w \in [1 \ldots n]$ such that $\Sigma(K_w) \neq \Sigma(M_w)$. Then, $\alpha_{dfg}(M) \neq \alpha_{dfg}(K)$.*

Proof Towards contradiction, assume that $\alpha_{\mathrm{dfg}}(K) = \alpha_{\mathrm{dfg}}(M)$, and perform case distinction.

In case no child K_i has the $\times(\tau, \rightarrow(\ldots))$ structure, $\alpha_{\mathrm{dfg}}(K)$ is a chain of strongly connected or unconnected clusters, which correspond to $\Sigma(K_i)$'s. Notice that α_{dfg}-edges can skip clusters, hence $\alpha_{\mathrm{dfg}}(K)$ contains a maximal $\rightarrow$ cut. The same holds for $\alpha_{\mathrm{dfg}}(M)$, and this holds for all such $\Sigma(K_i)$, so $\alpha_{\mathrm{dfg}}(M) \neq \alpha_{\mathrm{dfg}}(K)$.

In case at least one child K_i has a structure $\times(\tau, \rightarrow(K_{i_1}, \ldots K_{i_k}))$, the corresponding cluster $\Sigma(K_i)$ is a chain itself. By Rule $\mathrm{T}_{\times}$, at least one child of K_i (say K_{i_p}) is a pivot (Definition 7.3). By Lemma B.1, $(\rightarrow, \Sigma(K_{i_1}), \Sigma(K_{i_k}))$ is a partial $\rightarrow$-cut. Due to Requirement s.5, for every pivot, there is *one* partial $\rightarrow$-cut. The same holds for $\alpha_{\mathrm{dfg}}(M)$, and this holds for all such $\Sigma(K_i)$, so $\alpha_{\mathrm{dfg}}(M) \neq \alpha_{\mathrm{dfg}}(K)$.

Hence, $\alpha_{\mathrm{dfg}}(M) \neq \alpha_{\mathrm{dfg}}(K)$. □

F.2 Coo-stems

In Lemma B.2, we showed that a tree with a coo-stem has partial $\wedge$- and $\vee$-cuts. In this section, we prove the opposite: there can only be a partial $\wedge$- and $\vee$-cuts if the tree has such a corresponding coo-stem.

The main Lemmas, F.5 and F.6, consider partitions in a repetitive way: starting from a particular partition, a partial cut is considered, after which the sets of activities of the partial are merged in the partition, and a new, smaller, partition is obtained. For instance, the partition $\{\{a\}, \{b\}, \{c\}\}$ combined with the partial $\wedge$-cut $(\wedge, \{a\}, \{b\})$ becomes $\{\{a, b\}, \{c\}\}$. This reasoning procedure traverses the coo-stem of the process tree.

Invariant in the repetition is that every obtained partition corresponds to the process tree $(\mathbb{M}^\Sigma)$. To keep the invariant, we prove that for any partition in $\mathbb{M}^\Sigma$, partial $\wedge$- and $\vee$-cuts are always "correct" (Lemmas F.2 and F.3). Second, we prove that every obtained partition using such a partial cut is in $\mathbb{M}^\Sigma$ (Lemma F.4).

The repetition starts with the partition of the concurrent cut, which we formalize in Definition F.2. The repetition ends when the partial cut partitions the entire alphabet, and a contradiction is derived.

Definition F.2 (*activity sets of non-coo-subtrees*) Let M be a reduced process tree without duplicate activities. Then, $\Sigma^{\mathcal{C}}(M)$ denotes the activity sets of the non-coo-subtrees of M:

$$\begin{aligned}
\Sigma^{\mathcal{C}}(a) &= \{\{a\}\} \\
\Sigma^{\mathcal{C}}(\times(\tau, \ldots)) &= \Sigma^{\mathcal{C}}(\times(\ldots)) \\
\Sigma^{\mathcal{C}}(\times(M_1, \ldots M_m)) &= \{\Sigma(\oplus(M_1, \ldots M_m))\} && \text{with } \forall_i M_i \neq \tau \\
\Sigma^{\mathcal{C}}(\oplus(M_1, \ldots M_m)) &= \{\Sigma(\oplus(M_1, \ldots M_m))\} && \text{with } \oplus \in \{\rightarrow, \leftrightarrow, \circlearrowleft\} \\
\Sigma^{\mathcal{C}}(\oplus(M_1, \ldots M_m)) &= \bigcup_{1 \leqslant i \leqslant m} \Sigma^{\mathcal{C}}(M_i) && \text{with } \oplus \in \{\vee, \wedge\}
\end{aligned}$$

Notice that $\Sigma^{\mathcal{C}}(M)$ corresponds to a concurrent cut of the directly follows graph (Definition 5.2).

Lemma F.2 (partial $\vee$-cut corresponds to $\vee$) *Let $M \in \mathrm{C}_{coo}$ be a reduced process tree with a coo-stem, let $M' = \vee(M'_1, \ldots M'_m)$, $M' \in$ coo-stem(M) be a coo-stem node, let M'_i be a child of M', let $S \in \mathbb{M}^\Sigma(M)$ be a partition such that $\Sigma(M'_i) \in S$, and let $\alpha_{coo}(S)$ be a coo-graph. Take any $A \in S$ such that $A \neq \Sigma(M'_i)$. Then, $(\vee, \Sigma(M'_i), A)$ is a partial $\vee$-cut of $\alpha_{coo}(S)$ if and only if $\exists_{1 \leqslant j \leqslant m}\ A = \Sigma(M'_j)$.*

The proof of this lemma considers both directions of the bi-implication separately. Towards contradiction, it is assumed that such an A exists, after which semantics of sub-structures of M are used to derive a contradiction. "Appendix F.4" shows the detailed proof.

Lemma F.3 (partial $\wedge$-cut corresponds to $\wedge$) *Let $M \in \mathrm{C}_{coo}$ be a reduced process tree with a coo-stem, let $M' = \wedge(M'_1, \ldots M'_m)$, $M' \in$ coo-stem(M) be a coo-stem node, let M'_i be a child of M', let $S \in \mathbb{M}^\Sigma(M)$ a partition such that $\Sigma(M'_i) \in S$, and let $\alpha_{coo}(\mathcal{L}(M), S)$ be a coo-graph. Take any $A \in S$ such that $A \neq \Sigma(M'_i)$. Then, $(\wedge, \Sigma(M'_i), A)$ or $(\wedge, A, \Sigma(M'_i))$ is a partial $\wedge$-cut of $\alpha_{coo}(\mathcal{L}(M), S)$ if and only if $\exists_{1 \leqslant j \leqslant m}\ A = \Sigma(M'_j)$.*

The proof of this lemma is similar to the proof of Lemma F.2: the two directions are proven separately and semantics of sub-structures of M are used to derive a contradiction. For a detailed proof, see "Appendix F.5".

Lemma F.4 (merge sigmaset preservation) *Let M be a reduced process tree of C_{coo} with a coo-stem. Let $S \in \mathbb{M}^\Sigma(M)$ and let $C = (\oplus, S_1, \ldots S_n)$ be a partial $\oplus$-cut of $\alpha_{coo}(\mathcal{L}(M), S)$. Let $S' = S \setminus \{S_1, \ldots S_n\} \cup \{\cup_{i \in [1 \ldots n]} S_i\}$ be S collapsed with respect to C. Then, $S' \in \mathbb{M}^\Sigma(M)$.*

Proof As $S \in \mathbb{M}^{\Sigma}(M)$, there must be a tree M' to which S can be structurally mapped (Definition F.1). By Lemmas F.2 and F.3, for each S_i there is a node M'_i in M' such that $S_i = \Sigma(M'_i)$. By the associativity rules, M' can be transformed into M'' such that M'' contains a node $\oplus(M'_1, \dots M'_n)$. Then, S' can be structurally mapped to M''. Hence, $S' \in \mathbb{M}^{\Sigma}(M)$. □

Lemma F.5 (Operators are mutually exclusive for coo-stems) *Let $M = \wedge(M_1, \dots M_m)$, $K = \vee(K_1, \dots K_n)$ be reduced trees of C_{coo}. Then, $\alpha_{dfg}(M) \neq \alpha_{dfg}(K)$ or $\alpha_{coo}(M) \neq \alpha_{coo}(K)$.*

Proof Towards contradiction, assume that $\alpha_{\mathrm{dfg}}(M) = \alpha_{\mathrm{dfg}}(K)$ and $\alpha_{\mathrm{coo}}(M) = \alpha_{\mathrm{coo}}(K)$. Let $S = \Sigma^{\mathcal{C}}(M)$ be a partition of $\Sigma(M)$. As $\alpha_{\mathrm{dfg}}(M) = \alpha_{\mathrm{dfg}}(K)$, $S \in \Sigma^{\mathcal{C}}(K)$. Then, $S \in \mathbb{M}^{\Sigma}(M)$ and $S \in \mathbb{M}^{\Sigma}(K)$. (repeat from here) Take a partial cut C in $\alpha_{\mathrm{coo}}(\mathcal{L}(M), S)$. As $\alpha_{\mathrm{coo}}(M) = \alpha_{\mathrm{coo}}(K)$, C is a partial cut in $\alpha_{\mathrm{coo}}(\mathcal{L}(K), S)$ as well. Update S by collapsing it using C. Then, by Lemma F.4, still $S \in \mathbb{M}^{\Sigma}(M)$ and $S \in \mathbb{M}^{\Sigma}(K)$. Repeat such that $S = \{\Sigma(M_1), \dots \Sigma(M_m)\}$. As $S \in \mathbb{M}^{\Sigma}(K)$, $\forall_{i \in [1 \dots n]}\ \Sigma(K_i) \in S$. By Lemmas F.3 and F.2, there is a partial cut $(\wedge, \Sigma(M_1), \dots \Sigma(M_m))$ and a partial cut $(\vee, \Sigma(M_1), \dots \Sigma(M_m))$, which cannot happen by Definitions 7.7 and 7.8. Hence, $\alpha_{\mathrm{dfg}}(M) \neq \alpha_{\mathrm{dfg}}(K)$ or $\alpha_{\mathrm{coo}}(M) \neq \alpha_{\mathrm{coo}}(K)$. □

Lemma F.6 (Partitions are mutually exclusive for coo-stems) *Let $M = \oplus(M_1, \dots M_m)$, $K = \oplus(K_1, \dots K_n)$ be reduced trees of C_{coo} such that their activity partition is different, i.e. there is a $w \in [1 \dots n]$ such that $\Sigma(K_w) \neq \Sigma(M_w)$. Then, $\alpha_{dfg}(M) \neq \alpha_{dfg}(K)$ or $\alpha_{coo}(M) \neq \alpha_{coo}(K)$.*

Proof Towards contradiction, assume that there exists such a w. Similar to the proof of Lemma F.5, obtain a partition $S = \{\Sigma(M_1), \dots \Sigma(M_m)\}$ such that $S \in \mathbb{M}^{\Sigma}(M)$ and $S \in \mathbb{M}^{\Sigma}(K)$. By Lemmas F.3 and F.2, there is a node K_x in K that corresponds to $\Sigma(M_w)$. As we assumed $\Sigma(K_w) \neq \Sigma(M_w)$, this cannot happen, so $\alpha_{\mathrm{dfg}}(M) \neq \alpha_{\mathrm{dfg}}(K)$ or $\alpha_{\mathrm{coo}}(M) \neq \alpha_{\mathrm{coo}}(K)$. □

F.3 Uniqueness

Lemma F.7 (Operators are mutually exclusive for $\mathrm{C}_{\mathrm{coo}}$) *Take two reduced process trees of C_{coo} $K = \oplus(K_1, \dots K_n)$ and $M = \otimes(M_1, \dots M_m)$ such that $\oplus \neq \otimes$. Then, $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$ or $\alpha_{coo}(K) \neq \alpha_{coo}(M)$.*

We prove this lemma by showing a difference in abstraction for each pair of process tree operators. For a detailed proof, please refer to "Appendix F.6".

Lemma F.8 (Partitions are mutually exclusive for $\mathrm{C}_{\mathrm{coo}}$) *Take two reduced process trees of C_{coo} $K = \oplus(K_1 \dots K_n)$ and $M = \oplus(M_1 \dots M_m)$ such that their activity partition is different, i.e. there is a $1 \leqslant w \leqslant n$ such that $\Sigma(K_w) \neq \Sigma(M_w)$. Then, $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$ or $\alpha_{coo}(K) \neq \alpha_{coo}(M)$.*

This lemma is proven by case distinction on the process tree operators, while for each operator showing a difference in abstraction if the root partition differs. For a detailed proof, see "Appendix F.7".

Lemma F.9 (Directly follows graph and coo-relation uniqueness for $\mathrm{C}_{\mathrm{coo}}$) *For trees of class C_{coo}, the abstractions of normal forms of Sect. 4 are uniquely defined: for any two reduced process trees $K \neq M$ of C_{coo}, it holds that $\alpha_{dfg}(K) \neq \alpha_{dfg}(M)$ or $\alpha_{coo}(K) \neq \alpha_{coo}(M)$.*

The proof for this lemma is similar to the proof of Lemma 5.4, using Lemmas F.7 and F.8.

F.4 Lemma F.2

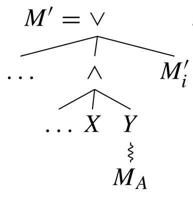
Lemma: Let $M \in \mathrm{C}_{\mathrm{coo}}$ be a reduced process tree with a coo-stem, let $M' = \vee(M'_1, \dots M'_m)$, $M' \in \text{coo-stem}(M)$ be a coo-stem node, let M'_i be a child of M', let $S \in \mathbb{M}^{\Sigma}(M)$ a partition such that $\Sigma(M'_i) \in S$, and let $\alpha_{\mathrm{coo}}(S)$ be a coo-graph. Take any $A \in S$ such that $A \neq \Sigma(M'_i)$. Then, $(\vee, \Sigma(M'_i), A)$ is a partial $\vee$-cut of $\alpha_{\mathrm{coo}}(S)$ if and only if $\exists_{1 \leqslant j \leqslant m}\ A = \Sigma(M'_j)$.

Proof Prove both directions separately:

$\Leftarrow$ Take such an M'_j. By Lemma B.2, $(\vee, \Sigma(M'_i), A)$ is a partial $\vee$-cut of $\alpha_{\mathrm{coo}}(S)$.

$\Rightarrow$ Towards contradiction, assume that there exists a set of activities A such that $(\vee, \Sigma(M'_i), A)$ is a partial $\vee$-cut of $\alpha_{\mathrm{coo}}(S)$ and $\forall_{1 \leqslant j \leqslant m}\ A \neq \Sigma(M'_j)$. By Definition F.1, A corresponds to a node in M. Let M_A be this node. Perform case distinction on whether the lowest common parent of M_A and M' is either M' itself or a parent of M':

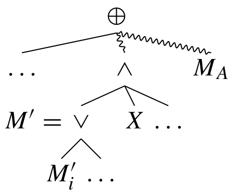
– The lowest common parent is M'. By the assumptions made, M_A is not a direct child of M', so there is a $\wedge$-node between M' and M_A. Without loss of generality, assume that this $\wedge$-node is a direct child of M'. Then, M contains the following structure, for certain nodes X and Y (wiggled edges denote possibly indirect children; M_A may be equal to Y):

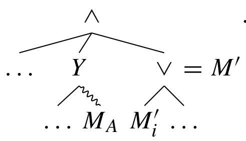


.

By semantics of $\vee$, execution of M'_i does not imply execution of either X or M_A. If X is not optional, then execution of M_A implies execution of X, and therefore, M_A and M'_i are not interchangeable ($\alpha_{\mathrm{coo}}(S) \not\models A \,\overline{\vee}\, \Sigma(M'_i)$).
If X is optional, then Y cannot be optional (reduction rule $\mathrm{C}_{\vee}$) and execution of X implies execution of Y. That is, a coo-graph $\alpha_{\mathrm{coo}}(S)'$ that contains $\Sigma(Y)$ and $\Sigma(X)$ would contain the traces $\langle \Sigma(X), \Sigma(Y) \rangle$ and $\langle \Sigma(X), \Sigma(M'_i), \Sigma(Y) \rangle$ but not $\langle \Sigma(X), \Sigma(M'_i) \rangle$. Therefore, $\alpha_{\mathrm{coo}}(S)' \not\models \Sigma(Y) \,\overline{\vee}\, \Sigma(M'_i)$. By definition of the occurrence function ("contains any activity of"), $\alpha_{\mathrm{coo}}(S) \not\models A \,\overline{\vee}\, \Sigma(M'_i)$.

– The lowest common parent is a parent of M'. Then, M contains either of the following structures, for certain nodes X and Y (wiggled edges denote possibly indirect children; M_A may be equivalent to Y):



or


.

In the first case, X and M' cannot be both optional (reduction rule $\mathrm{C}_{\vee}$). Then, by semantics of $\wedge$, execution of X implies execution of M' (if M' is not optional) and/or execution of M' implies execution of X (if X is not optional). By the reduction rules, there must be an $\vee$-node

or an $\times(\tau, \ldots)$ construct between $\oplus$ and M'. Then, execution of neither M' nor X is implied by execution of M_A, so $\alpha_{\text{coo}}(S) \not\models A \overline{\vee} \Sigma(M'_i)$.
In the second case, a similar argument holds for Y and M'.

Then, by Definition 7.7, $(\vee, A, \Sigma(M'_i))$ is not a partial $\vee$-cut of $\alpha_{\text{coo}}(S)$.
Hence, $(\vee, \Sigma(M'_i), A)$ is a partial $\vee$-cut of $\alpha_{\text{coo}}(S)$ if and only if $\exists_{1 \leqslant j \leqslant m}\ A = \Sigma(M'_j)$. □

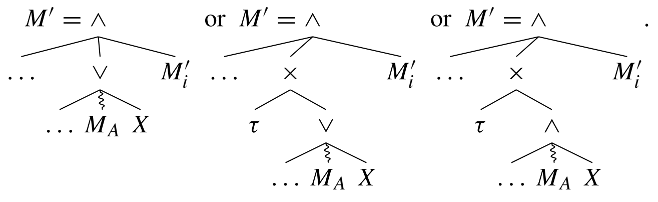
F.5 Lemma F.3

Lemma: Let $M \in \mathrm{C}_{\text{coo}}$ be a reduced process tree with a coo-stem, let $M' = \wedge(M'_1, \ldots M'_m)$, $M' \in \text{coo-stem}(M)$ be a coo-stem node, let M'_i be a child of M', let $S \in \mathbb{M}^{\Sigma}(M)$ a partition such that $\Sigma(M'_i) \in S$, and let $\alpha_{\text{coo}}(\mathcal{L}(M), S)$ be a coo-graph. Take any $A \in S$ such that $A \neq \Sigma(M'_i)$. Then, $(\wedge, \Sigma(M'_i), A)$ or $(\wedge, A, \Sigma(M'_i))$ is a partial $\wedge$-cut of $\alpha_{\text{coo}}(\mathcal{L}(M), S)$ if and only if $\exists_{1 \leqslant j \leqslant m}\ A = \Sigma(M'_j)$.

Proof Prove both directions separately:

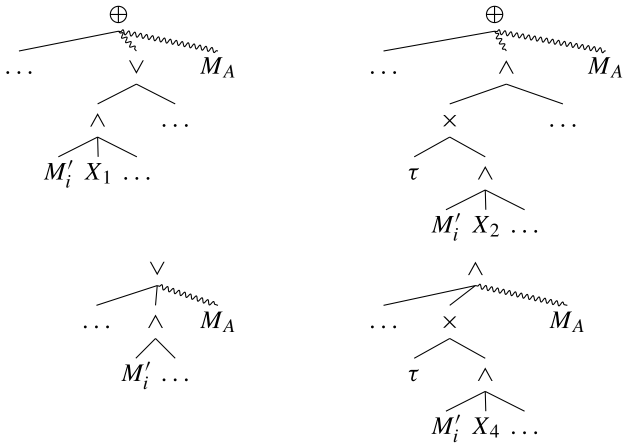
$\Leftarrow$ Take such an M'_j. By Lemma B.2, $(\wedge, \Sigma(M'_i), A)$ is a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$.
$\Rightarrow$ Towards contradiction, assume that there exists a set of activities A such that $(\wedge, \Sigma(M'_i), A)$ is a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$ and $\forall_{1 \leqslant j \leqslant m}\ A \neq \Sigma(M'_j)$. By Definition F.1, A corresponds to a node in M. Let M_A be this node. Perform case distinction on whether the lowest common parent of M_A and M' is either M' itself or a parent of M':

– The lowest common parent is M'. By the assumptions made, M_A is not a direct child of M'. Then, M contains either of the following structures, for a certain node X (wiggled edges denote possibly indirect children):



In all cases, $\alpha_{\text{coo}}(S) \not\models \Sigma(M'_i) \Rrightarrow A$, so $(\wedge, \Sigma(M'_i), A)$ is not a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$ and the partial cut $(\wedge, A, \Sigma(M'_i))$ must adhere to the second option of Definition 7.8. In the first case, if M'_i is not optional, then execution of X implies execution of M'_i. Then, for every $Y \in \mathbb{M}^{\Sigma}(X)$, $\alpha_{\text{coo}}(S) \models Y \Rrightarrow \Sigma(M'_i)$, and at least one such Y is in S, which violates Requirement c.3.4. If M'_i is optional, then $\alpha_{\text{coo}}(S) \not\models A \Rrightarrow \Sigma(M'_i)$, which violates Requirement c.3.2. In the second and third cases, by reduction rule $\mathrm{C}_{\vee}$, M'_i cannot be optional, and an argument similar to the first case applies. Hence, $(\wedge, A, \Sigma(M'_i))$ is not a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$.

– The lowest common parent is a parent of M'. Then, M contains either of the following structures, for certain nodes X_1, X_2, X_4 (wiggled edges denote possibly indirect children):



In all these cases, $\alpha_{\text{coo}}(S) \not\models A \Rrightarrow \Sigma(M'_i)$, so $(\wedge, A, \Sigma(M'_i))$ is not a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$ and the partial cut $(\wedge, \Sigma(M'_i), A)$ must adhere to the second option of Definition 7.8.

- In the first case, if $\alpha_{\text{coo}}(S) \models \Sigma(M'_i) \Rrightarrow A$ then for every $Y \in \mathbb{M}^{\Sigma}(X_1)$ it holds that $\alpha_{\text{coo}}(S) \models Y \Rrightarrow A$, and at least one such Y is in S, which violates Requirement c.3.4.
- The second case is similar to the first.
- In the third case, $\alpha_{\text{coo}}(S) \not\models \Sigma(M'_i) \Rrightarrow A$, which violates Requirement c.3.2.
- In the fourth case, for every $\alpha_{\text{coo}}(S) \models Y \in \mathbb{M}^{\Sigma}(X_4)$, $Y \Rrightarrow A$, and one such Y is in S, which violates Requirement c.3.4.

Then, neither $(\wedge, A, \Sigma(M'_i))$ nor $(\wedge, \Sigma(M'_i), A)$ is a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$.
Hence, $(\wedge, \Sigma(M'_i), A)$ or $(\wedge, A, \Sigma(M'_i))$ is a partial $\wedge$-cut of $\alpha_{\text{coo}}(S)$ if and only if $\exists_{1 \leqslant j \leqslant m}\ A = \Sigma(M'_j)$. □

F.6 Lemma F.7

Lemma: Take two reduced process trees of C_{coo} $K = \oplus(K_1, \dots K_n)$ and $M = \otimes(M_1, \dots M_m)$ such that $\oplus \neq \otimes$. Then, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$ or $\alpha_{\text{coo}}(K) \neq \alpha_{\text{coo}}(M)$.

Proof Towards contradiction, assume that $\alpha_{\text{dfg}}(K) = \alpha_{\text{dfg}}(M)$ and $\alpha_{\text{coo}}(K) = \alpha_{\text{coo}}(M)$. Perform case distinction on $\oplus$:

$\oplus = \times$ and one child K_i is a τ. As described before, the footprint of $\times(\tau, \dots)$ applies whenever the root is optional. Thus, we need to consider the case in which M is optional, but does not have the $\times(\tau, .)$ construct as root. Let $K = \times(\tau, (\oplus'(K'_1, \dots K'_k))$ (or $K = \times(K'_1, \dots, K'_k)$ if $\oplus = \times$) and perform case distinction on $\otimes$:

$\otimes = \times$ By semantics of $\times$, $\alpha_{\text{dfg}}(M)$ consists of unconnected clusters. As $\alpha_{\text{dfg}}(M) = \alpha_{\text{dfg}}(K)$, and by semantics of the operators, $\oplus' = \times$. At least one child (say M_j) is optional, but does not have the $\times(\tau, .)$ construct as root. Let K'_i be the corresponding child in K. Then, $\mathcal{L}(K'_i) \cup \{\epsilon\} = M_j$. M_j cannot be a single activity (cannot be optional without the $\times(\tau, .)$ construct), or $\times$ (by Rule $A_{\times}$). For the other operators, see the other cases (termination of the argument guaranteed as K_i and M_j are strictly smaller than M).

$\otimes = \rightarrow$ By semantics of $\rightarrow$, $\alpha_{\text{dfg}}(M)$ consists of a chain of clusters. As $\alpha_{\text{dfg}}(M) = \alpha_{\text{dfg}}(K)$, and by semantics of the operators, $\oplus' = \rightarrow$. By semantics of $\rightarrow$, all children M_j are optional. By Rule $T_{\times}$, at least one child (say K_i) is not optional. Therefore, there is a non-empty trace in $\mathcal{L}(K)$ in which no activity of $\Sigma(K_i)$ occurs. There is no such M_j, thus $\mathcal{L}(K) \neq \mathcal{L}(M)$.

$\otimes = \wedge$ By semantics of $\wedge$, all children M_j must be optional. However, by Rule $C_{\vee}$, this situation cannot occur.

$\otimes = \vee$ By semantics of $\vee$, at least one child M_j is optional. Consider the options for M_j exhaustively: $\times(\tau, \ldots)$ (would be reduced by Rule $T_{\vee,\times}$), $\vee(\ldots)$ (would be reduced by Rule $A_{\vee}$), $\wedge(\ldots)$ with all children optional (would be reduced by Rule $C_{\vee}$), a (cannot be optional without $\times(\tau, .)$ construct), or, hence, an optional non-coo-subtree without $\times(\tau, .)$ as root. For the other operators, see the other cases (termination of the argument guaranteed as K_i and M_j are strictly smaller than M).

$\otimes = \leftrightarrow$ By semantics of the process tree operators, $\oplus' = \leftrightarrow$. By reduction rule $T_{\times}$, at least one child K_i' is not optional. By Definition 7.9, all children M_i must be optional.

Take a child $K'_{j \neq i}$. Then, execution of some activity in K_j' implies execution of some activity in K_i', while there can be no child $M_{j \neq i}$ with such a dependency can exist in M_i, as $\leftrightarrow$ cannot be nested by Definition 7.9. Hence, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\otimes = ⟲$ In this case, $⟲$ is optional and this is excluded by Requirement C_{coo}.1.2.

Hence, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\oplus = \times$ and no child is a τ. The graph $\alpha_{\text{dfg}}(M)$ consists of several unconnected components, while as $\otimes$ is either ($\rightarrow, \wedge, \vee, \leftrightarrow, ⟲$), $\alpha_{\text{dfg}}(M)$ is connected. Thus, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\oplus = \rightarrow$ The graph $\alpha_{\text{dfg}}(M)$ is a chain, while as $\otimes$ is either ($\times, \wedge, \vee, \leftrightarrow, ⟲$), $\alpha_{\text{dfg}}(M)$ is either unconnected or strongly connected. Thus, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$.

$\oplus = \wedge$ We consider the remaining cases of $\otimes$:

$\otimes = \vee$ By Lemma F.5, $\alpha_{\text{dfg}}(M) \neq \alpha_{\text{dfg}}(K)$ or $\alpha_{\text{coo}}(M) \neq \alpha_{\text{coo}}(K)$.

$\otimes = \leftrightarrow$ As shown in Sect. 7.1, optionality does not influence the footprint of $\wedge$ or $\leftrightarrow$. Therefore, Lemma 5.2 applies. Hence, $\alpha_{\text{dfg}}(M) \neq \alpha_{\text{dfg}}(K)$.

$\otimes = ⟲$ By Definition 7.9, children of $⟲$ are not allowed to be optional. Therefore, Lemma 5.2 applies. Hence, $\alpha_{\text{dfg}}(M) \neq \alpha_{\text{dfg}}(K)$.

$\oplus = \vee$ $\vee$ has the same directly follows footprint as $\wedge$. Therefore, the arguments given at $\oplus = \wedge$, $\otimes = \leftrightarrow$ and $\otimes = ⟲$ apply.

$\oplus = \leftrightarrow$ We consider the remaining case of $\otimes$, being $\otimes = ⟲$.

By Definition 7.9, children of $⟲$ are not allowed to be optional. Therefore, Lemma 5.2 applies. Hence, $\alpha_{\text{dfg}}(M) \neq \alpha_{\text{dfg}}(K)$.

We conclude that $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$ or $\alpha_{\text{coo}}(K) \neq \alpha_{\text{coo}}(M)$. □

F.7 Lemma F.8

Lemma: take two reduced process trees of C_{coo} $K = \oplus(K_1 \dots K_n)$ and $M = \oplus(M_1 \dots M_m)$ such that their activity partition is different, i.e. there is a $1 \leqslant w \leqslant n$ such that $\Sigma(K_w) \neq \Sigma(M_w)$. Then, $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$ or $\alpha_{\text{coo}}(K) \neq \alpha_{\text{coo}}(M)$.

Proof Without loss of generality, we assume a fixed order of subtrees for all operators. Towards contradiction, assume that $\alpha_{\text{dfg}}(K) = \alpha_{\text{dfg}}(M)$ and $\alpha_{\text{coo}}(K) = \alpha_{\text{coo}}(M)$. Perform case distinction on $\oplus$ (the case for K and M swapped is symmetric).

$\oplus = \times$ If a child K_i is τ, see the proof of Lemma F.7.
As K is reduced, $\alpha_{\text{dfg}}(K)$ contains n unconnected clusters, corresponding to $\Sigma(K_i)$'s. These clusters themselves are connected (by Rule $A_\times$ and semantics of the other operators); hence, $\alpha_{\text{dfg}}(K)$ contains a maximal $\times$ cut. The same holds for $\alpha_{\text{dfg}}(M)$, hence $\Sigma(K_w) = \Sigma(M_w)$.
$\oplus = \rightarrow$ By Lemma F.1, $\Sigma(K_w) = \Sigma(M_w)$.
$\oplus = \wedge$ By Lemma F.6, $\Sigma(K_w) = \Sigma(M_w)$.
$\oplus = \leftrightarrow$ Let $K_w = \otimes(K_{w_1} \dots K_{w_p})$. Perform case distinction on $\otimes$:

$\otimes = \times$ and a child M_i is τ. The $\leftrightarrow$ operator has a distinct directly follows graph footprint, on which $\times(\tau, .)$ has no influence. Therefore, refer to the other cases as if $\otimes$ is the child of $\times(\tau, .)$, using the requirements of C_{coo}.
$\otimes = \times$ and no child M_i is τ. By semantics of $\times$, no end activity of K_{w_1} has a connection to any start activity of any other K_{w_j}. Thus, as M contains an interleaved activity partition, $\Sigma(K_w) \subseteq \Sigma(M_w)$.
$\otimes = \rightarrow$ Similar to the $\times$ case.
$\otimes = \wedge$ and $\otimes = \vee$. By Definition 7.9, at least one child of K_w has disjoint start and end activities. Take such a child K_{w_y}, and consider two activities: $a \notin \text{Start}(K_{w_y})$ and $b \in \Sigma(K_w) \backslash K_{w_y}$. By semantics of $\wedge$ and $\vee$, $\alpha_{\text{dfg}}(b, a)$. Then, by Lemma 5.1, $a \in \Sigma(M_w)$ and $b \in \Sigma(M_w)$. This holds for all b and by symmetry for $\text{Start}(K_{w_y}) \cup \text{End}(K_{w_y})$. By semantics of $\leftrightarrow$, non-start non-end activities only have connections with start/end activities of K_w. Therefore, $\Sigma(K_w) \backslash (\text{Start}(K_w) \cup \text{End}(K_w)) \subseteq \Sigma(M_w)$. Hence, $\Sigma(K_w) \subseteq \Sigma(M_w)$.
$\otimes = \leftrightarrow$ Excluded by Definition 7.9.
$\otimes = \circlearrowleft$ By semantics of $\leftrightarrow$, non-start non-end activities only have connections with start/end activities of K_w. Therefore, $\Sigma(K_w) \backslash (\text{Start}(K_w) \cup \text{End}(K_w)) \subseteq \Sigma(M_w)$. All activities $\in \text{Start}(K_w) \cup \text{End}(K_w)$ have connections from/to $\text{End}(K_{w_2}) \cup \text{Start}(K_{w_2})$, thus $\text{Start}(K_w) \cup \text{End}(K_w) \subseteq \Sigma(M_w)$. Hence, $\Sigma(K_w) \subseteq \Sigma(M_w)$.

By symmetry, $\Sigma(K_w) = \Sigma(M_w)$.
$\oplus = \vee$ In K, $\Sigma(K_w) \overline{\vee} \Sigma(K_{v \neq w})$. By Lemma F.2 and as $\alpha_{\text{dfg}}(K) = \alpha_{\text{dfg}}(M)$ and $\alpha_{\text{coo}}(K) = \alpha_{\text{coo}}(M)$, it holds that $\Sigma(M_w) \overline{\vee} \Sigma(M_{v \neq w})$. Hence, $\Sigma(K_w) = \Sigma(M_w)$.
$\oplus = \circlearrowleft$ By Definition 7.9, children of $\circlearrowleft$ are not allowed to be optional. Therefore, Lemma 5.3 applies.

By contradiction, we conclude that $\alpha_{\text{dfg}}(K) \neq \alpha_{\text{dfg}}(M)$ or $\alpha_{\text{coo}}(K) \neq \alpha_{\text{coo}}(M)$. □

References

1. van der Aalst WMP (2016) Process mining–data science in action, 2nd edn. Springer, Berlin. https://doi.org/10.1007/978-3-662-49851-4

2. Buijs JCAM, van Dongen BF, van der Aalst WMP (2014) Quality dimensions in process discovery: the importance of fitness, precision, generalization and simplicity. Int J Cooperative Inf Syst. https://doi.org/10.1142/S0218843014400012
3. van der Aalst WMP, Weijters AJMM, Maruster L (2004) Workflow mining: discovering process models from event logs. IEEE Trans Knowl Data Eng 16:1128–1142
4. vanden Broucke SKLM, Weerdt JD (2017) Fodina: a robust and flexible heuristic process discovery technique. Decis Support Syst 100:109–118
5. Leemans SJJ, Fahland D, van der Aalst WMP (2013) Discovering block-structured process models from event logs—a constructive approach. In: PETRI NETS 2013. Lecture notes in computer science, vol 7927. Springer, pp 311–329. https://doi.org/10.1007/978-3-642-38697-8_17
6. Augusto A, Conforti R, Dumas M, Rosa ML (2017) Split miner: discovering accurate and simple business process models from event logs. In: ICDM 2017. IEEE Computer Society, pp 1–10. https://doi.org/10.1109/ICDM.2017.9
7. Weidlich M, van der Werf JMEM (2012) On profiles and footprints—relational semantics for petri nets. In: Petri Nets
8. Polyvyanyy A, Armas-Cervantes A, Dumas M, García-Bañuelos L (2016) On the expressive power of behavioral profiles. Form Asp Comput 28:597–613
9. Leemans SJJ, Fahland D, van der Aalst WMP (2014) Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann N, Song M, Wohed P (eds) Business process management workshops. Lecture notes in business information processing, vol 171. Springer, pp 66–78
10. Augusto A, Conforti R, Dumas M, Rosa ML, Maggi FM, Marrella A, Mecella M, Soo A (2018) Automated discovery of process models from event logs: review and benchmark. IEEE Trans Knowl Data Eng. arXiv:1705.02288
11. OMG (2011) Business Process Model and Notation (BPMN), Version 2.0. http://www.omg.org/spec/BPMN/2.0. Accessed 8 July 2019
12. van Zelst SJ, van Dongen BF, van der Aalst WMP (2018) Event stream-based process discovery using abstract representations. Knowl Inf Syst 54(2):407–435. https://doi.org/10.1007/s10115-017-1060-2
13. Syamsiyah A, van Dongen BF, van der Aalst WMP (2016) DB-XES: enabling process discovery in the large. In: SIMPDA 2016. LNBIP, vol 307. Springer, pp 53–77. https://doi.org/10.1007/978-3-319-74161-1_4
14. Syamsiyah A, van Dongen BF, van der Aalst WMP (2017) Recurrent process mining with live event data. In: BPM Workshops 2017. LNBIP, vol 308. Springer, pp 178–190
15. Weerdt JD, Backer MD, Vanthienen J, Baesens B (2012) A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf Syst 37:654–676
16. Badouel E, Bernardinello L, Darondeau P (2015) Petri net synthesis. Springer, Berlin
17. de Medeiros, AKA, van Dongen BF, van der Aalst WMP, Weijters AJMM (2004) Process mining for ubiquitous mobile systems: an overview and a concrete algorithm. In: Baresi L, Dustdar S, Gall HC, Matera M (eds) Ubiquitous mobile information and collaboration systems, second CAiSE workshop, UMICS 2004, Riga, Latvia, 7–8 June 2004, Revised selected papers. Lecture notes in computer science, vol 3272. Springer, pp 151–165. https://doi.org/10.1007/978-3-540-30188-2_12
18. Wen L, Wang J, Sun J (2006) Detecting implicit dependencies between tasks from event logs. In: Zhou X, Li J, Shen HT, Kitsuregawa M, Zhang Y (eds) Frontiers of WWW Research and Development—APWeb 2006, 8th Asia-Pacific Web Conference, Harbin, China, 16–18 January 2006, Proceedings. Lecture notes in computer science, vol 3841. Springer, pp 591–603. https://doi.org/10.1007/11610113_52
19. Wen L, van der Aalst WMP, Wang J, Sun J (2007) Mining process models with non-free-choice constructs. Data Min Knowl Discov 15(2):145–180. https://doi.org/10.1007/s10618-007-0065-y
20. Wen L, Wang J, van der Aalst WMP, Huang B, Sun J (2010) Mining process models with prime invisible tasks. Data Knowl Eng 69(10):999–1021. https://doi.org/10.1016/j.datak.2010.06.001
21. Wen L, Wang J, Sun J (2007) Mining invisible tasks from event logs. In: Dong G, Lin X, Wang W, Yang Y, Yu JX (eds) Advances in data and web management, Joint 9th Asia-Pacific Web Conference, APWeb 2007, and 8th international conference, on web-age information management, WAIM 2007, Huang Shan, China, 16–18 June 2007, Proceedings. Lecture notes in computer science, vol 4505. Springer, pp 358–365. https://doi.org/10.1007/978-3-540-72524-4_38
22. Guo Q, Wen L, Wang J, Yan Z, Yu PS (2015) Mining invisible tasks in non-free-choice constructs. In: Motahari-Nezhad HR, Recker J, Weidlich M (eds) Business process management—13th international conference, BPM 2015, Innsbruck, Austria, August 31–September 3 2015, Proceedings. Lecture notes in computer science, vol 9253. Springer, pp 109–125. https://doi.org/10.1007/978-3-319-23063-4_7
23. Leemans SJJ, Fahland D, van der Aalst WMP (2014) Discovering block-structured process models from incomplete event logs. In: Ciardo G, Kindler E (eds) Application and theory of petri nets and concurrency—35th international conference, PETRI NETS 2014, Tunis, Tunisia, 23–27 June 2014. Pro-

ceedings. Lecture notes in computer science, vol 8489. Springer, pp 91–110. https://doi.org/10.1007/978-3-319-07734-5_6
24. Russell N, van der Aalst WMP, ter Hofstede AHM (2016) Workflow patterns: the definitive guide. MIT Press, Cambridge
25. Zha H, Wang J, Wen L, Wang C, Sun JG (2010) A workflow net similarity measure based on transition adjacency relations. Comput Ind 61:463–471
26. Sun J, Gu T, Qian J (2017) A behavioral similarity metric for semantic workflows based on semantic task adjacency relations with importance. IEEE Access 5:15609–15618
27. van Dongen BF, Dijkman RM, Mendling J (2008) Measuring similarity between business process models. In: CAiSE 2008. Lecture notes in computer science, vol 5074. Springer, pp 450–464. https://doi.org/10.1007/978-3-540-69534-9_34
28. Polyvyanyy A, Weidlich M Conforti R, Rosa ML, ter Hofstede AHM (2014) The 4c spectrum of fundamental behavioral relations for concurrent systems. In: Petri Nets
29. Wang J, He T, Wen L, Wu N, ter Hofstede AHM, Su J (2010) A behavioral similarity measure between labeled petri nets based on principal transition sequences—(short paper). In: OTM 2010. Lecture notes in computer science, vol 6426. Springer, pp 394–401
30. Becker M, Laue R (2012) A comparative survey of business process similarity measures. Comput Ind 63(2):148–167
31. Kunze M, Weidlich M, Weske M (2011) Behavioral similarity—a proper metric. In: Business process management 2011. Lecture Notes in Computer Science, vol 6896. Springer, pp 166–181
32. Kunze M, Weidlich M, Weske M (2015) Querying process models by behavior inclusion. Softw Syst Model 14(3):1105–1125. https://doi.org/10.1007/s10270-013-0389-6
33. Polyvyanyy A, Weidlich M, Weske M (2012) Isotactics as a foundation for alignment and abstraction of behavioral models. In: BPM
34. Weidlich M, Mendling J, Weske M (2012) Propagating changes between aligned process models. J Syst Softw 85:1885–1898
35. van der Werf JMEM, van Dongen BF, Hurkens CAJ, Serebrenik A (2009) Process discovery using integer linear programming. Fundam Inf 94(3–4):387–412. https://doi.org/10.3233/FI-2009-136
36. Schunselaar DMM, Verbeek E, van der Aalst WMP, Reijers HA (2013) A framework for efficiently deciding language inclusion for sound unlabelled wf-nets. In: Joint proceedings of the international workshop on petri nets and software engineering (PNSE'13) and the international workshop on modeling and business environments (ModBE'13), Milano, Italy, 24–25 June 2013. CEUR Workshop Proceedings, vol 989. CEUR-WS.org, pp 135–154
37. Leemans SJJ, Fahland D, van der Aalst WMP (2018) Scalable process discovery and conformance checking. Softw Syst Model 17(2):599–631. https://doi.org/10.1007/s10270-016-0545-x
38. Buijs JCAM, van Dongen BF, van der Aalst WMP (2012) A genetic algorithm for discovering process trees. In: Proceedings of the IEEE congress on evolutionary computation, CEC 2012, Brisbane, Australia, 10–15 June. IEEE, pp 1–8. https://doi.org/10.1109/CEC.2012.6256458
39. Molka T, Redlich D, Gilani W, Zeng X, Drobek M (2015) Evolutionary computation based discovery of hierarchical business process models. In: Abramowicz W (ed) Business information systems—18th international conference, BIS 2015, Poznań, Poland, 24–26 June 2015, Proceedings. Lecture notes in business information processing, vol 208. Springer, pp 191–204. https://doi.org/10.1007/978-3-319-19027-3_16
40. Leemans SJJ (2017) Robust process mining with guarantees. Ph.D. thesis, Eindhoven University of Technology
41. Polyvyanyy A, Vanhatalo J, Völzer H (2010) Simplified computation and generalization of the refined process structure tree. In: Bravetti M, Bultan T (eds) Web services and formal methods—7th international workshop, WS-FM 2010, Hoboken, NJ, USA, 16–17 September 2010. Revised Selected Papers. Lecture notes in computer science, vol 6551. Springer, pp 25–41. https://doi.org/10.1007/978-3-642-19589-1_2
42. Reisig W (1985) Petri nets: an introduction. Springer, New York
43. Gallo G, Longo G, Pallottino S (1993) Directed hypergraphs and applications. Discrete Appl Math 42(2):177–201. https://doi.org/10.1016/0166-218X(93)90045-P
44. Conforti R, Rosa ML, ter Hofstede AHM (2017) Filtering out infrequent behavior from business process event logs. IEEE Trans Knowl Data Eng 29(2):300–314. https://doi.org/10.1109/TKDE.2016.2614680
45. Leemans SJ, Fahland, D (2019) dfahland/exp-abstractions-in-pm-KAIS: original experiment. https://doi.org/10.5281/zenodo.3243981
46. Leemans SJJ, Fahland D (2019) Process models obtained from event logs with different information-preserving abstractions. https://doi.org/10.5281/zenodo.3243988

47. van Dongen B (2012) BPI challenge 2012 dataset. https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

Sander J. J. Leemans is an Assistant Professor (Lecturer) in Process Mining in the Business Process Management discipline of the School of Information Systems at Queensland University of Technology (QUT). He has research interests in both theoretical and practical aspects of process mining. He is researching the theoretical foundations of process mining, how to bring advanced academic techniques to end users in easy to use tools, how to help organizations to apply process mining, and how to increase the transparency of process mining using stochastic methods.

Dirk Fahland is an Associate Professor in Process Analytics on Multi-Dimensional Event Data of the Analytics for Information Systems group at Eindhoven University of Technology (TU/e). His research interests are in describing and analysing complex and distributed systems and processes driven by an interplay of multiple components, data objects, and entities in complex relations. He is researching foundational concepts and techniques for processing and analysing the multi-dimensional event data produced by such systems, including querying and pre-processing event data as well as discovering and querying models, and predicting future behaviour.