GECCO: Constraint-driven Abstraction of Low-level Event Logs

Adrian Rebmann
University of Mannheim, Germany
rebmann@informatik.uni-mannheim.de

Matthias Weidlich
Humboldt-Universität zu Berlin, Germany
matthias.weidlich@hu-berlin.de

Han van der Aa
University of Mannheim, Germany
han@informatik.uni-mannheim.de

***Abstract*—Process mining enables the analysis of complex systems using event data recorded during the execution of processes. Specifically, models of these processes can be discovered from event logs, i.e., sequences of events. However, the recorded events are often too fine-granular and result in unstructured models that are not meaningful for analysis. Log abstraction therefore aims to group together events to obtain a higher-level representation of the event sequences. While such a transformation shall be driven by the analysis goal, existing techniques force users to define *how* the abstraction is done, rather than *what* the result shall be.**

In this paper, we propose GECCO, an approach for log abstraction that enables users to impose requirements on the resulting log in terms of constraints. GECCO then groups events so that the constraints are satisfied and the distance to the original log is minimized. Since exhaustive log abstraction suffers from an exponential runtime complexity, GECCO also offers a heuristic approach guided by behavioral dependencies found in the log. We show that the abstraction quality of GECCO is superior to baseline solutions and demonstrate the relevance of considering constraints during log abstraction in real-life settings.

I. Introduction

Process mining [1] comprises methods to analyze complex systems based on event data that is recorded during the execution of processes. Specifically, by discovering models of these processes from event logs [2], i.e., sequences of events, process mining yields insights into how a process is truly executed. Yet, the recorded events are often too fine-granular for meaningful analysis, and the resulting variability in the recorded event sequences leads to complex models. This trend is amplified when events originate from sources, such as real-time location systems [3] and user interface logs [4].

To tackle this issue, *log abstraction* is a technique to lift the event sequences of a log to a more abstract representation, by grouping low-level events into high-level activities. Existing techniques for log abstraction (cf., [5], [6]) differ in the adopted algorithms and employ, e.g., temporal clustering of events [7] or the detection of predefined patterns [8]. Yet, their focus is on *how* the abstraction is conducted, rather than *what* properties the abstracted log shall satisfy. Without control on the result of log abstraction, however, it is hard to ensure that an abstraction is appropriate for a specific analysis goal.

To achieve effective abstraction, a respective technique must thus enable users to incorporate dedicated constraints on the resulting log. Here, the main challenge is that such constraints may be defined at different levels of granularity, i.e., they may relate to properties of individual events, types of events, or groups of event types. Finding an optimal abstraction, i.e., a log that satisfies all constraints while being as close as possible to the original log, is a hard problem, due to the sheer number of possible abstractions and the interplay of constraints at different granularity levels. Hence, log abstraction is challenging also from a computational point of view.

In this paper, we propose GECCO, an approach for log abstraction that enables a user to impose requirements on the resulting log in terms of constraints. As such, it supports a declarative characterization of the properties the abstracted log shall adhere to, in order to be meaningful for a specific analysis purpose. We summarize our contributions as follows:

- We define the problem of optimal log abstraction. It requires minimizing the distance to the original log while satisfying a set of constraints on the abstracted log.
- We define the scope of GECCO as an instantiation of the log-abstraction problem, covering a broad set of common constraint types and a distance measure.
- As part of GECCO, we present an algorithm for exhaustive log abstraction. Striving for more efficient processing, we also provide a heuristic algorithm that is guided by behavioral dependencies found in the log.

Our evaluation demonstrates that the abstracted logs obtained with GECCO provide better abstraction and are more cohesive than those obtained with baseline techniques. As such, process discovery algorithms also yield more structured models.

In the remainder, we first motivate the need for user-defined constraints in log abstraction (§II). We then formally define the problem of optimal log abstraction (§III). Next, GECCO is presented as an instantiation of the log-abstraction problem (§IV), along with exhaustive and heuristic algorithms to address it (§V). Finally, we report on evaluation results (§VI), review related work (§VII), and conclude (§VIII).

II. Motivation

Log abstraction is motivated by the presence of fine-granular events in process mining settings. Such fine-granular events typically induce a high degree of behavioral variability, so that the application of process discovery algorithms yields so-called *spaghetti process models*, which are incomprehensible due to their complexity, as e.g., depicted in Figure 1. Log abstraction overcomes this issue by grouping events, thereby reducing the variability of the behavior to be depicted.

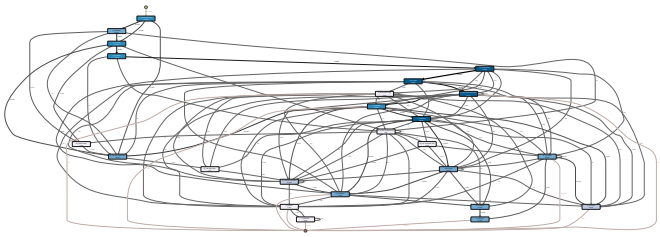

Fig. 1: A so-called *spaghetti* process model.

For illustration purposes, consider the simple event log in Table I, which consists of four event sequences corresponding to a request-handling process. Here, blue underlined events denote process steps performed by a clerk, whereas the others are performed by a manager.

The event log shows that each case starts with the receipt of a request (event *rcp*) by a clerk. The clerk checks the request either casually (*ckc*) or thoroughly (*ckt*) depending on the information provided. Then, the request is forwarded to a manager, who either accepts (*acc*) or rejects (*rej*) it. Afterwards, the clerk may or may not assign priority to a request (*prio*), before they inform the customer (*inf*) and archive the request (*arv*). The latter two activities can be performed in either order, as shown, e.g., in σ_1 and σ_2. As shown in trace σ_4, a rejected request may also be returned to the applicant, who will resubmit it, restarting the procedure.

TABLE I: Exemplary traces of an event log.

ID	Trace
σ_1	⟨rcp, ckc, acc, prio, inf, arv⟩
σ_2	⟨rcp, ckt, rej, prio, arv, inf⟩
σ_3	⟨rcp, ckc, acc, inf, arv⟩
σ_4	⟨rcp, ckc, rej, rcp, ckt, acc, prio, arv,inf⟩

Although this process consists of only eight distinct steps, its behavior is already fairly complex. This is evidenced by the *directly-follows graph* (DFG) shown in Figure 2, which depicts the steps that can directly succeed each other in the process. The graph's complexity already obscures some of the key behavioral aspects of the process. Log abstraction may alleviate this problem. However, existing techniques focus on how the abstraction shall be done. For instance, they may exploit that the steps *ckt*, *ckc*, *acc*, and *rej* are closely correlated from a behavioral perspective and abstract them to a single activity. Yet, this is not meaningful for many analysis tasks, as it would obscure the fact that the activity encompasses some steps performed by a clerk (*ckt* and *ckc*), whereas others are performed by a manager (*acc* and *rej*).

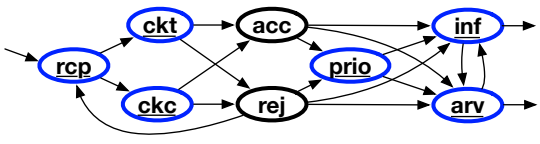

Fig. 2: Directly-follows graph (DFG) of the running example.

By incorporating user-defined constraints on what properties the abstracted log shall satisfy, such a result can be avoided. For instance, if a user wants to primarily understand the interactions between employees, while abstracting from details on the individual steps performed by them, a constraint may enforce that each activity comprises only events performed by the same employee role. If applied in a naive manner, this constraint would result in two groups of events classes, i.e., $g_{clrk} = \{rcp, ckc, ckt, prio, inf, arv\}$ and $g_{mgr} = \{acc, rej\}$. Yet, using these groups directly for log abstraction is not meaningful either. The group g_{clrk} includes steps that occur at the start of the process, as well as steps that only happen at the end. Moreover, abstracting the steps in g_{mgr} to a single activity would obfuscate that $\{acc, rej\}$ exclude each other and that only after step *rej*, the process is potentially restarted.

Against this background, our approach to log abstraction, GECCO, aims at constructing activities for groups of events that satisfy user-specified constraints, while also preserving the behavior represented in event sequences as much as possible. For the example, this would result in an abstraction that consists of four groups: $g_{clrk1} = \{rcp, ckc, ckt\}$, containing the initial steps performed by the clerk, $\{acc\}$ and $\{rej\}$, as singleton groups of steps that are mutually exclusive and both performed by the manager, and $g_{clrk2} = \{prio, inf, arv\}$, the final steps of the process performed by the clerk. The directly-follows graphs obtained with this log abstraction is shown in Figure 3. It highlights that a clerk starts working on each case, before handing it over to the manager. Accepted requests are completed by the clerk, whereas rejected requests may be completed or returned to the start of the process.

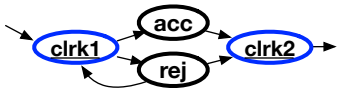

Fig. 3: DFG of the log abstracted with GECCO.

Obtaining the groups for such abstraction requires solving a computationally complex problem, though, as defined next.

III. Problem Statement

We first define an event model for our work (§III-A), before formalizing the addressed log-abstraction problem (§III-B).

A. Event model

We consider events recorded during the execution of a process and write $\mathcal{E}$ for the universe of all events. An event $e \in \mathcal{E}$ is of a certain *event class*, i.e., its type, which we denote as $e.C \in \mathcal{C}$, with $\mathcal{C}$ as the universe of event classes. For instance, the log in Table I consists of eight event classes, each corresponding to a specific process step. Furthermore, each event carries information about its context, which may include aspects such as a timestamp, the executing role, or relevant data values. We capture this context by a set of data attributes $\mathcal{D} = \{D_1, \ldots, D_p\}$, with $dom(D_i)$ as the domain of attribute D_i, $1 \leq i \leq p$. We write $e.D$ for the value of attribute D of an event e. For instance, in §II, each event class

is assigned a particular role, i.e., a *clerk* or a *manager*. As such, an event e may capture that $e.C = rcp$ and $e.role = clerk$.

A single execution of a process, called a *trace*, is modeled as a sequence of events $\sigma \in \mathcal{E}^*$, such that no event can occur in more than one trace. An event log is a set of traces, $L \subseteq 2^{\mathcal{E}^*}$, with $\mathcal{L}$ as the universe of all events logs, and $C_L \subseteq \mathcal{C}$ as the set of event classes of the events in L.

An event log can be represented as a *directly-follows graph* (DFG) that indicates if two event classes ever immediately succeed each other in the log. Given a log L, its DFG is a directed graph (V, E), with the set of vertices V corresponding to the event classes of C_L and the set of edges $E \subseteq V \times V$ representing a directly-follows relation $>_L$, defined as: $a >_L b$, if there is a trace $\sigma = \langle e_1, \ldots, e_n \rangle$ and $i \in \{1, \ldots, n-1\}$, such that $\sigma \in L$ and $e_i.C = a$ and $e_{i+1}.C = b$.

B. The log-abstraction problem

Log abstraction aims to construct groups of similar events for an event log. Formally, this is captured by a grouping, i.e., a set of groups, $G \subset 2^{\mathcal{C}}$, over the event classes C_L, such that each class $c \in C_L$ is part of exactly one group $g \in G$. Given a grouping, a function $\texttt{abstract} : \mathcal{L} \times 2^{\mathcal{C}} \to \mathcal{L}$ is applied to obtain an abstracted log L'. For instance, using the log and grouping from §II, σ_1 is abstracted to $\sigma'_1 = \langle clrk1, acc, clrk2 \rangle$.

We target scenarios in which a user formulates requirements on what properties the abstracted event log and, hence, the grouping G shall satisfy, e.g., to group only events performed by a single role (see §II). Then, we aim to identify a grouping $\hat{G}$ that meets these requirements, while preserving the behavior of the traces as much as possible. To this end, we define $\texttt{dist} : \mathcal{L} \times 2^{\mathcal{C}} \to \mathbb{R}$ as a *distance function* that quantifies the distance of a grouping to an event log. Also, using $\mathcal{R}$ to denote the universe of possible constraints, we define a predicate $\texttt{holds} : 2^{\mathcal{C}} \times \mathcal{R} \times \mathcal{L} \to \{true, false\}$ to denote whether a grouping satisfies a set of constraints for a given log. Based thereon, we define the optimal log-abstraction problem:

Problem 1 (Optimal log abstraction)**.** Given an event log L with event classes C_L, a distance function $\texttt{dist}$, and a set of constraints R, the log-abstraction problem is to find an optimal grouping $\hat{G} = \{g_1, \ldots, g_k\}$, such that:

- $\hat{G}$ is an exact cover of C_L, i.e., $\bigcap_i^k g_i = \emptyset \wedge \bigcup_i^k g_i = C_L$;
- $\hat{G}$ adheres to the desired constraints R, i.e., $\texttt{holds}(\hat{G}, R, L) = true$;
- the distance $\texttt{dist}(\hat{G}, L)$ is minimal.

IV. GECCO: Scope

To address Problem 1, we propose GECCO for the Grouping of Event Classes using Constraints and Optimization. This section shows how GECCO instantiates Problem 1 by specifying constraint types (§IV-A) and a distance function (§IV-B).

A. Covered constraint types

GECCO is able to handle a broad range of constraints on a grouping G. As shown through the examples in Table II, we consider *grouping* constraints, *class-based* constraints, and *instance-based* constraints. The table also indicates a monotonicity property of the constraints, which is important when aiming to find an optimal grouping in an efficient manner.

Grouping constraints. This constraint category can be used to bound the size of a grouping G, i.e., the number of high-level activities that will appear in the abstracted log. An upper bound restricts the size and complexity of the obtained log, whereas a lower bound can limit the applied degree of abstraction.

Satisfaction. We use $R_G \subseteq R$ to refer to the subset of grouping constraints. Whether a constraint $r \in R_G$ holds can be directly checked against the grouping size, $|G|$. As such, for the $\texttt{holds}$ predicate, we require $\forall r \in R_G : r(G) = true$.

Class-based constraints. The second category of constraints can be used to influence the characteristics of an individual group $g \in G$ in terms of the event classes that it can contain. GECCO supports any class-based constraint for which satisfaction can be checked by considering g in isolation, i.e., without having to compare g to other groups in G. As shown in Table II, this, for instance, includes constraints that each group shall comprise at least (or at most) a certain number of event classes, as well as *cannot-link* and *must-link* constraints, which may be used to specify that two event classes must or must not be grouped together.

Satisfaction. We use $R_C \subseteq R$ for class-based constraints. The satisfaction of a constraint $r \in R_C$ is directly checked by evaluating the contents of each group $g \in G$. Hence, the $\texttt{holds}$ predicate requires that $\forall g \in G, \forall r \in R_C : r(g) = true$.

Monotonicity. Class-based constraints that specify a minimum requirement on groups, e.g., a minimal group size, are monotonic: If the constraint holds for a group g, it also holds for any larger group g', with $g \subset g'$. In other words, adding event classes to a group can never result in a (new) constraint violation. By contrast, constraints that express requirements that may not be exceeded, e.g., a maximal group size or *cannot-link* constraint, are anti-monotonic: If they hold for a group g, they also hold for any subset of that group $g' \subset g$. However, if a group g violates a constraint, a larger group g', with $g \subset g'$, also violates it.

Instance-based constraints. The third category comprises constraints that shall hold for each *instance* of a group $g \in G$, i.e., a sequence of (not necessarily consecutive) events that occur in the same trace and of which the event classes are part of g. In line with the event context defined in §III-A, we use the shorthand $g.D$ to refer to the set of values of attribute D for a group g when defining constraints of this type.

As indicated in Table II, diverse constraints can be defined on the instance-level, relating to attribute values, associated roles, and duration, such as *the total cost of an instance is at most 500$* and *the average duration of group instances must be at most 1 hour*. As shown in the table's last row, also looser constraints may be expressed, such as ones that only need to hold for 95% of the respective group instances. In fact, as for class-based constraints, GECCO supports all constraints of which satisfaction can be checked for an individual group g.

TABLE II: Exemplary constraints covered by GECCO.

Category	Examples	Monotonicity
Grouping constraints	There should be at most 10 groups in the final grouping.	n/a
	There should be at least 5 groups in the final grouping.	n/a
Class-based constraints	There should be at least 5 event classes per group.	monotonic
	At most 10 event classes should be grouped together.	anti-monotonic
	The event classes *rcp* and *acc* cannot be members of the same group.	anti-monotonic
	The event classes *inf* and *arv* must be members of the same group.	non-monotonic
Instance-based constraints	At least 2 distinct document codes must be associated with a group instance.	monotonic
	The cost of a group instance must be at most 500$.	anti-monotonic
	The duration of a group instance must be at most 1 hour on average.	non-monotonic
	The time between consecutive events in a group instance must at most be 10 minutes.	anti-monotonic
	Each group instance may contain at most 1 event per event class.	anti-monotonic
	At least 95% of the group instances must have a cost below 500$.	anti-monotonic

Satisfaction. We write $R_I \subseteq R$ for the instance-based constraints. Contrary to the other categories, these constraints must be explicitly checked against the event log L, specifically for each group $g \in G$ and each instance of g in the traces of L.

Formally, we first define a function $\texttt{inst} : \mathcal{E}^* \times 2^{\mathcal{C}} \rightarrow 2^{\mathcal{E}^*}$, which returns all instances of a group in a given trace. The operationalization of $\texttt{inst}$ is straightforward for simple cases: An instance of group g is the projection of the event classes of g over a trace σ. In σ_1, σ_2, and σ_3 of our running example, exactly one instance of each group occurs per trace and, e.g., $\texttt{inst}(\sigma_1, g_{clrkI}) = \{\langle rcp, ckc \rangle\}$. However, processes often include recurring behavior, such as trace $\sigma_4 = \langle$rcp, ckc, rej, rcp, ckt, acc, prio, inf, arv$\rangle$, in which a request is first rejected, sent back to the restart the process, and then accepted in the second round. Here, to detect multiple instances of a group, we instantiate function $\texttt{inst}$ based on an existing technique [9] that recognizes when a trace contains recurring behavior and splits the (projected) sequence accordingly. For the above trace, this yields $\texttt{inst}(\sigma_4, g_{clrkI}) = \{\langle rcp, ckc \rangle, \langle rcp, ckt \rangle\}$. Note that $\texttt{inst}$ can also be used to enforce cardinality constraints, e.g., if a user desires that each group instance should contain at least 2 events of a particular event class.

Given the function $\texttt{inst}$, a constraint $r \in R_I$ is satisfied if for each group $g \in G$, r holds for each instance $\xi \in \texttt{inst}(\sigma, g)$, for each $\sigma \in L$. Note that constraints are vacuously satisfied for traces that do not include an instance of a particular group, i.e., where $\texttt{inst}(\sigma, g) = \emptyset$. Therefore, for instance-based constraints, $\texttt{holds}$ is checked for each $g \in G$ as $\forall\ r \in R_I, \forall\ \sigma_i \in L, \forall\ \xi \in \texttt{inst}(\sigma_i, g) : r(\xi) = \mathit{true}$. For looser constraints, e.g., ones that should for 95% of the group instances, predicate satisfaction is adapted accordingly.

Monotonicity. As is the case for class-based ones, instance-based constraints are monotonic when they specify a minimum requirement to be met, e.g., each instance should take *at least* one hour, and anti-monotonic when they specify something that may not be exceeded, e.g., each instance may take *at most* one hour. However, constraints in R_I may also be based on aggregations that behave in a non-monotonic manner, such as constraints that consider the *average* or *variance* of attribute values per group instance or *sums* including negative values. In these cases, adding and removing event classes from a group can result in a violated constraint to now hold or vice versa.

B. Distance measure

To determine which event classes are suitable candidates to be grouped together, we employ a distance function $\texttt{dist}(G, L)$ that quantifies the relatedness of the event classes per group. Although our work is largely independent of a specific distance function, we argue that log abstraction should group together event classes such that 1) events within a group are *cohesive*, i.e., the events belonging to a single group instance occur close to each other, meaning there are few interspersed events from other instances; 2) events within a group are *correlated*, i.e., the events belonging to a single group typically occur together in the same trace and group instance; 3) larger groups are favored over *unary groups*, i.e., the grouping G actually results in an abstraction. To capture these three aspects, we propose the following distance function for an individual group g and a log L:

$$\texttt{dist}(g, L) = \sum_{\xi \in \texttt{inst}(L,g)} \frac{\frac{\texttt{interrupts}(\xi)}{|\xi|} + \frac{\texttt{missing}(\xi,g)}{|g|} + \frac{1}{|g|}}{|\texttt{inst}(L, g)|} \quad (1)$$

The first summand in the numerator of Eq. 1 considers cohesion. Here, $\texttt{interrupts}(\xi)$ counts how many events from other instances are interspersed between the first and last events of a given group instance ξ. As such, this penalizes groups of events that are often *interrupted* by others, e.g., in a trace $\langle a, b, c, d, e \rangle$, grouping a and e together is unfavorable, since the instance $\langle a, e \rangle$ has three interspersed events. In Eq. 1, the number of interruptions is considered relative to the length of ξ. The second summand in the numerator of Eq. 1 quantifies the degree of completeness of ξ with respect to g, thus capturing the correlation between the events in g. Here, $\texttt{missing}(\xi, g)$ returns how many event classes from g are missing from its instance ξ, which is then offset against the total number of classes $|g|$. Finally, since groups with a single event class have perfect cohesion and correlation by default, we include $\frac{1}{|g|}$ to ensure that larger groups with the same cohesion and correlation are favored, thus avoiding unary groups when possible.

Finally, to quantify the entire distance of a grouping G, we sum up the distance values of all groups in G, resulting in the

following function that will be minimized in our approach:

$$\texttt{dist}(G, L) = \sum_{g \in G} \texttt{dist}(g, L) \tag{2}$$

V. The GECCO Approach

Next, we describe how GECCO achieves the goal of finding an optimal event grouping, given the distance function and constraints defined above. §V-A provides a high-level overview, while §V-B to §V-D outline the algorithmic details.

A. Approach Overview

As shown in Figure 4, GECCO takes an event log L and a set of user-defined constraints R as input. Then, GECCO applies three main steps in order to obtain an abstracted log L'.

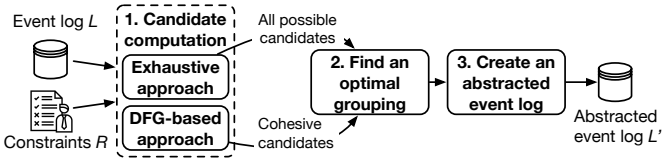

Fig. 4: The main steps of GECCO.

In Step 1, GECCO computes a set of candidate groups $\mathcal{G}$, i.e., groups of event classes that adhere to the constraints in R. As depicted in Figure 4, we propose two instantiations for this step: an exhaustive instantiation and an efficient DFG-based one. The *exhaustive* instantiation yields a set of candidates that is guaranteed to be complete and, thus, assures that it can afterwards be used to establish an optimal grouping if it exists.

However, this approach may be intractable in practice. Therefore, we also propose a *DFG-based* alternative, which only retrieves cohesive candidates, i.e., candidates likely to be part of an optimal grouping. By exploiting the process-oriented nature of the input data, cohesive candidates are identified efficiently. Any solution obtained using this instantiation is still guaranteed to satisfy the constraints in R, yet may have a sub-optimal distance score.

In Step 2, GECCO uses the identified set of candidates in order to find a single optimal grouping G that minimizes the distance function dist, while ensuring that all constraints are met and each event class in C_L is assigned to exactly one group in G. To achieve this, we formulate this task as a mixed-integer programming (MIP) problem.

Finally, having obtained a grouping G, Step 3 abstracts event log L by replacing the events in a trace with activities based on groups defined in G, yielding an abstracted log L'.

B. Step 1: Computation of candidate groups

In this step, GECCO computes a set of candidate groups of event classes, $\mathcal{G}$, i.e., subsets of C_L that adhere to constraints in R. As described in §V-A, we propose an exhaustive and a DFG-based instantiation for this step:

Exhaustive candidate computation. To obtain a complete set of candidate groups, in principle, every combination of event classes, i.e., every subset of C_L, needs to be checked against the constraints in R. However, we are able to considerably reduce this search space by (for the moment) only looking for groups $g \subseteq C_L$ that actually co-occur in at least one trace in the log (referred to as *group co-occurrence*) and by considering the monotonicity of the constraints in R. Then, candidate groups of increasing size can be identified iteratively, as outlined in Algorithm 1.

Algorithm 1 Exhaustive candidate computation

Input Event log L, user constraints R
Output Set of candidate groups $\mathcal{G}$

1: *mode* ← setCheckingMode(R)
2: *toCheck* ← { $\{c\}$ for $c \in C_L$ } ▷ Set the first groups to check
3: **while** *toCheck* $\neq \emptyset$ **do**
4: **if** *mode* = *monotonic* **then**
5: $\mathcal{G}_{new} \leftarrow \{g \in$ *toCheck* if $\exists g' \in \mathcal{G} : g' \subset g \vee \texttt{holds}(g, L, R)\}$
6: **else**
7: $\mathcal{G}_{new} \leftarrow \{g \in$ *toCheck* if $\texttt{holds}(g, L, R)\}$
8: $\mathcal{G} \leftarrow \mathcal{G} \cup \mathcal{G}_{new}$
9: **if** *mode* = *anti-monotonic* **then**
10: *toCheck* ← expandGroups($\mathcal{G}_{new}$)
11: **else**
12: *toCheck* ← expandGroups(*toCheck*)
13: *toCheck* ← $\{g \in$ *toCheck* if $\texttt{occurs}(g, L)\}$
14: **return** $\mathcal{G}$

Initialization. We first set the constraint-checking *mode* that shall be applied (line 1) based on the monotonicity of the constraints in R. Specifically, *mode* is set to *anti-monotonic* if R contains at least one such constraint, to *monotonic* if all constraints in $R \setminus R_G$ are monotonic (i.e., all constraints that must be checked *per group*), and otherwise to *non-monotonic*.

Using this constraint-checking mode, we employ two pruning strategies. First, consider a group $g_1 \subset C_L$ and a constraint set R in which all constraints $R \setminus R_G$ are monotonic. If $\texttt{holds}(g_1, R, L) = \mathit{true}$, any supergroup $g_1' \supseteq g_1$ will also adhere to the constraints, since adding more event classes to g_1 will never lead to a violation of a monotonic constraint. Therefore, in the *monotonic* mode, we can avoid the costs of constraint validation for g_1'. Second, consider a group $g_2 \subset C_L$, known to violate any anti-monotonic constraint in R, i.e, $\texttt{holds}(g_2, R, L) = \mathit{false}$. Then we also know that no supergroup $g_2' \supseteq g_2$ can adhere to R, as expanding a group can never resolve violations of anti-monotonic constraints. Thus, in the *anti-monotonic* mode, all supergroups of g_2 can be skipped.

With *mode* set, the algorithm then adds all event classes of C_L as singleton groups to the set of potential candidates, *toCheck*, which shall be checked in the first iteration (line 2).

Candidate assessment. In each iteration, Algorithm 1 first establishes a set $\mathcal{G}_{new}$, which contains all groups in *toCheck* that adhere to the constraints in R. When validating a group, we check constraints in R_C before ones in R_I, since the former do not require a pass over the event log, thus, minimizing the validation cost per candidate. In the *monotonic* mode, the algorithm employs the first pruning strategy by directly adding any group g, for which there is a $g' \subset g$ already in $\mathcal{G}$, given that we then know that the monotonic constraints will

be satisfied for g as well (line 5). For other groups and for the other two modes, we need to check $\texttt{holds}(g, L, R)$ for each group $g \in \mathit{toCheck}$ (lines 5 and 7). Having established $\mathcal{G}_{new}$, the new candidates are added to the total set $\mathcal{G}$ (line 8).

Group expansion. Next, the algorithm repopulates *toCheck* with larger groups that shall be assessed in the next iteration. In the *anti-monotonic* mode, using the second pruning strategy, the algorithm only needs to expand groups that are known to adhere to all anti-monotonic constraints in R. Therefore, in this case, we only expand the groups in $\mathcal{G}_{new}$ (line 10). This expansion involves the creation of new groups that consist of a group $g \in \mathcal{G}_{new}$ with an additional event class from C_L. Naturally, the *anti-monotonic* mode avoids the creation of groups that contain subgroups that are already known to violate *R*. For the *monotonic* and *non-monotonic* mode, we also need to expand groups that currently violate the constraints, since their supergroups may still lead to constraint satisfaction. Therefore, these modes expand all groups in *toCheck* (line 12). Afterwards, we only retain those groups in $\mathit{toCheck}(g, L)$ that actually occur in the event log, by checking if there is at least one trace in L that contains events corresponding to all event classes in g (line 13).

Termination. The algorithm stops if there are no new candidates to be checked, returning the set of all candidates, $\mathcal{G}$.

Computational complexity. While Algorithm 1 is guaranteed to yield a complete set of candidates, its time complexity is exponential with respect to the number of event classes in the event log, i.e., $2^{|C_L|}$. In the worst case, each of the subsets of C_L must be analyzed against the entire log, where primarily the number of traces is important, since each group must be separately checked against all traces. Given that each checked group may become a candidate, the algorithm's space complexity is also bounded by $2^{|C_L|}$. Hence, this exhaustive approach can quickly become infeasible.

DFG-based candidate computation. In the light of the runtime complexity of the exhaustive approach, we also propose a DFG-based approach to compute candidate groups. It exploits behavioral regularities in event logs in order to efficiently derive a set of *cohesive candidate groups*.

Intuition. Log abstraction aims to find cohesive groups of event classes and, therefore, is more likely to group together event classes that occur close to each other. In our running example, even though the *request receipt* (*rcp*) and *archive request* (*arv*) event classes meet the constraint (both are performed by a *clerk*), it is unlikely that they will end up in the same activity in an optimal grouping G, since *rcp* occurs at the start of each trace and *arv* at the end.

We exploit this characteristic of optimal groupings by identifying only candidates that occur near each other. This is achieved by establishing a DFG of the event log and traversing this graph to find highly cohesive candidates groups. Since this traversal again iteratively increases the candidate size, we can still apply the aforementioned pruning strategies.

This idea is illustrated in Figure 5, which visualizes (parts of) two iterations for the running example, highlighting candidate groups that are checked. Iteration 1 involves the assessment of paths of length two, consisting of connected event classes. This identifies, e.g., the candidate paths [*prio,inf*], [*prio,arv*], and [*inf,arv*], which all adhere to the constraint, whereas, e.g., [*acc, inf*] is recognized as a violating path, since *acc* and *inf* are performed by different roles. Given their distance from each other in the DFG, this iteration avoids checking groups such as {*rcp, arv*} and {*ckt, inf*}. In the next iteration, since the running example deals with an *anti-monotonic* constraint, we concatenate pairs of constraint-adhering paths to obtain candidate paths (i.e., groups) of length three, as shown for [*prio, inf, arv*] in Figure 5.

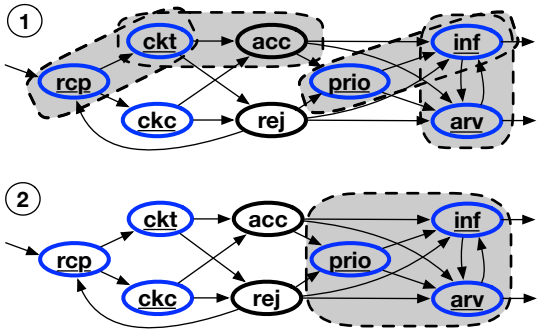

Fig. 5: DFG-based candidate computation (Iterations 1 & 2).

The DFG-based approach works as described in Algorithm 2. Next to an event log L and a constraint set *R*, it takes as input a parameter k, defining the beam-search width.

Initialization. The algorithm starts by again setting the constraint-checking mode (line 1), before establishing the log's DFG (line 2), as defined in §III-A. Then, for every node n in the DFG (i.e., for every event class), we add the trivial path $\langle n \rangle$ to the set of candidates to check in the first iteration (line 3).

Candidate assessment. In principle we could assess for each path $p \in \mathit{toCheck}$ if p's nodes form a proper candidate group, as we do in the exhaustive approach. However, we here recognize that in event logs with a lot of variability, the number of paths to check will still be considerable. Hence, we allow for a further pruning of the search space by incorporating a *beam-search* [10] component in the algorithm. In this beam search, we only keep the k most promising candidates (i.e., the *beam*) in each iteration of the algorithm.

To do this, each iteration starts by sorting the candidate paths in *toCheck*, giving priority to paths of which the nodes have the lowest distance to each other, according to $\texttt{dist}(\texttt{nodes}(p), L)$ (line 5). Then, the algorithm picks candidates from *sortedPaths* as long as there are candidates to pick and the beam width k has not been reached (line 8). Each group g, defined by the nodes in a path (i.e., $g = \texttt{nodes}(p)$), is then checked for constraint satisfaction. As for the exhaustive approach, we check constraints in R_C before ones in R_I minimizing validation cost per candidate.

Here, we employ the same pruning strategies for *monotonic* and *anti-monotonic* constraint-checking modes as done for the exhaustive approach. Therefore, in the *monotonic* mode, a group g can be directly added to the set of candidates $\mathcal{G}$

Algorithm 2 DFG-based candidate computation

Input event log L, user constraints R, pruning parameter k
Output Set of candidate groups $\mathcal{G}$

1: *mode* ← `setCheckingMode(R)`
2: $DFG \leftarrow$ `computeDFG(L)`
3: *toCheck* $\leftarrow \{\langle n \rangle$ for $n \in DFG.\texttt{nodes}\}$ ▷ First paths to check
4: **while** *toCheck* $\neq \emptyset$ **do**
5: *sortedPaths* ← `sort(toCheck, dist)` ▷ Lowest `dist` first
6: *toExpand* $\leftarrow \emptyset$
7: $i \leftarrow 0$
8: **while** $i < \min(|sortedPaths|, k)$ **do**
9: $p \leftarrow sortedPaths[i]$ ▷ Get next path
10: $g \leftarrow \texttt{nodes}(p)$ ▷ Derive nodes, i.e., event classes of p
11: **if** *mode* = *monotonic* **then**
12: **if** $\exists g' \in \mathcal{G} : g' \subset g \vee \texttt{holds}(g, L, R)$ **then**
13: $\mathcal{G} \leftarrow \mathcal{G} \cup \{g\}$
14: $toExpand \leftarrow toExpand \cup \{p\}$
15: **else if** $\texttt{holds}(g, L, R)$ **then**
16: $\mathcal{G} \leftarrow \mathcal{G} \cup \{g\}$
17: $toExpand \leftarrow toExpand \cup \{p\}$
18: **else if** *mode* $\neq$ *anti-monotonic* **then**
19: $toExpand \leftarrow toExpand \cup \{p\}$
20: $i \leftarrow i + 1$
21: *toCheck* $\leftarrow \emptyset$ ▷ Start computing new paths to check
22: **for** $p = \langle p_0, \ldots, p_m \rangle \in$ *toExpand* **do**
23: **for** $(p_m, succ) \in DFG.\texttt{outgoingEdges}(p_m)$ **do**
24: **if** $succ \notin \texttt{nodes}(p)$ **then**
25: *toCheck* $\leftarrow$ *toCheck* $\cup \{ \langle p_0, \ldots, p_m, succ \rangle \}$
26: **for** $(pred, p_0) \in DFG.\texttt{incomingEdges}(p_0)$ **do**
27: **if** $pred \notin \texttt{nodes}(p)$ **then**
28: *toCheck* $\leftarrow$ *toCheck* $\cup \{ \langle pred, p_0, \ldots, p_m \rangle \}$
29: *toCheck* $\leftarrow \{p \in$ *toCheck* if $\texttt{occurs}(\texttt{nodes}(p), L)\}$
30: **return** $\mathcal{G}$

if we have already seen a subset $g' \subset g$ that adheres to the constraints (lines 12–13), whereas in the *anti-monotonic* mode, we no longer expand paths that violate the constraints (lines 18–19).

Path expansion. The candidates for the next iteration are created by expanding paths in *toExpand* with either a predecessor of their first or a successor of their last node (lines 22–28). Again, we then only retain those paths in *toCheck*, whose groups g actually occur in the event log (line 29).

Termination. The algorithm stops if no candidates are left, i.e., *toCheck* is empty and the set $\mathcal{G}$ is returned.

Computational complexity. The DFG-based approach is considerably more efficient than the exhaustive one. In each iteration, the approach expands up to k groups, each into up to $|C_L| - 1$ new candidates. As such, given the maximum of $|C_L|$ iterations, the worst-case time and space complexity is $k * |C_L|^2$. Moreover, this worst case only occurs if the DFG is a complete digraph and no constraints are imposed.

Dealing with exclusion. Generally, it is undesirable to group exclusive event classes together, since such classes never occur in the same trace. This is why we have so far omitted these from consideration, by ensuring that $\texttt{occurs}(g, L)$ holds for every candidate group $g \in \mathcal{G}$. Yet, when exclusive event classes (or groups) are proper alternatives to each other, we make an exception for this. In these cases, grouping them together will result in a further complexity reduction of the event log, while not affecting its expressiveness.

Intuition. To illustrate this, reconsider the running example, which contains two sets of exclusive event classes, {*ckc*, *ckt*}, corresponding to two ways in which a request can be checked, and {*acc*, *rej*}, corresponding to acceptance and rejection of a request. By considering Figure 6, we see that the former two event classes are proper behavioral alternatives: both *ckc* and *ckt* are preceded and followed by the exact same sets of event classes.As such, behavioral alternatives can be defined as groups of event classes that have identical *pre-* and *postsets* in the DFG. Merging them will thus not lead to a loss of behavioral information. By contrast, *acc* and *rej* do not represent proper alternatives to each other, since their postsets differ. Particularly, while after acceptance the process always moves forward to one of the event classes in {*prio*, *inf*, *arv*}, a rejection may also result in a loop back to the start of the process (*rcp*). Therefore, if these exclusive classes were merged, we would obscure the fact that there are two different possibilities here.

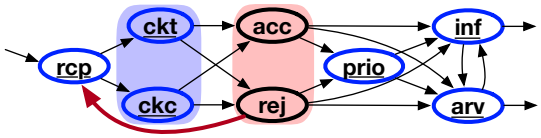

Fig. 6: DFG of the running example highlighting proper behavioral alternatives (blue) and exclusive event classes, which are no behavioral alternatives (red).

Candidate identification. We employ Algorithm 3 to determine if previously identified candidate groups in $\mathcal{G}$, with excluding event classes, can be merged to obtain additional candidates.

The algorithm establishes a set *equivGroups* consisting of candidate groups that share the same pre- and postset (line 4). Then, a stack is created consisting of all pairs of groups in this set (lines 5–7). For each pair (g_i, g_j) in this stack, we assess if g_i and g_j are indeed exclusive to each other and if their merged group, g_{ij}, still adheres to the user constraints (line 11). Both conditions can be efficiently checked. The former by ensuring that there are no edges from nodes in g_i to nodes in g_j or vice versa, while for the latter only adherence to class-based constraints (R_C) needs to be assessed, given that instance-based constraints cannot be (newly) violated when merging exclusive groups, thus avoiding a pass over the event log L.

If g_{ij} is indeed a proper, new candidate, we next determine if this group can also be combined together with its preset, postset, or with both, to create more candidates (lines 13–19). For instance, having identified {*ckt, ckc*} as a new candidate group for the running example, we would this way recognize that this new group together with its preset (event class *rcp*) also forms a proper candidate group: {*rcp, ckt, ckc*}, since both {*rcp, ckt*} and {*rcp, ckc*} were also already part of $\mathcal{G}$.

After establishing these new candidates, the algorithm adds any new pair (g_{ij}, g_k) to the stack, so that also iteratively larger candidates, consisting of three or more exclusive groups, can

be identified (lines 20–21). The algorithm terminates when all relevant pairs have been assessed, returning the updated set $\mathcal{G}$ as the final output of Step 1 of the approach.

Algorithm 3 Find exclusive candidate groups

Input Event log L, user constraints R, current candidate groups $\mathcal{G}$
Output Extended set of candidate groups $\mathcal{G}$

1: $DFG \leftarrow$ `computeDFG(`L`)`
2: $seenGroups \leftarrow \emptyset$
3: **for** $g \in \mathcal{G} \setminus seenGroups$ **do**
4: $equivGroups \leftarrow DFG$.`equalPrePost(`$g$`)` $\cup \{g\}$
5: $pairsToCheck \leftarrow$ `new Stack()`
6: **for** $(g_i, g_j) \in equivGroups \times equivGroups$ **do**
7: $pairsToCheck$.`push(`(g_i, g_j)`)`
8: **while** $\neg pairsToCheck$.`isEmpty()` **do**
9: $(g_i, g_j) \leftarrow pairsToCheck$.`pop()`
10: $g_{ij} \leftarrow g_i \cup g_j$ ▷ Merge into single group
11: **if** `exclusive(`g_i, g_j`)` $\wedge$ `holds(`g_{ij}, L, R_C`)` **then**
12: $\mathcal{G} \leftarrow \mathcal{G} \cup \{g_{ij}\}$ ▷ New candidate found
13: $(pre, post) \leftarrow (DFG$.`pre(`$g_i$`)`, DFG.`post(`g_i`))`
14: **if** $pre \cup post \cup g_i \in \mathcal{G} \wedge pre \cup post \cup g_j \in \mathcal{G}$ **then**
15: $\mathcal{G} \leftarrow \mathcal{G} \cup \{pre \cup post \cup g_{ij}\}$
16: **else if** $pre \cup g_i \in \mathcal{G} \wedge pre \cup g_j \in \mathcal{G}$ **then**
17: $\mathcal{G} \leftarrow \mathcal{G} \cup \{pre \cup g_{ij}\}$
18: **else if** $post \cup g_i \in \mathcal{G} \wedge post \cup g_j \in \mathcal{G}$ **then**
19: $\mathcal{G} \leftarrow \mathcal{G} \cup \{post \cup g_{ij}\}$
20: **for** $g_k \in equivGroups \setminus \{g_i, g_j\}$ **do**
21: $pairsToCheck$.`push(`(g_{ij}, g_k)`)`
22: $equivGroups \leftarrow equivGroups \cup \{g_{ij}\}$
23: $seenGroups \leftarrow seenGroups \cup equivGroups$
24: **return** $\mathcal{G}$

Computational complexity. Algorithm 3 has linear complexity with respect to $|\mathcal{G}|$, i.e., the number of candidate groups stemming from the previous step. As such, its worst-case time and space complexity is $2^{|C_L|}$ when previously using the exhaustive approach and $k * |C_L|^2$ for the DFG-based one.

C. Step 2: Finding an optimal grouping

Having established candidate groups $\mathcal{G}$, we set out to find an optimal grouping $G \subseteq \mathcal{G}$ based on these candidates, which is a set of disjoint groups that covers all event classes, while minimizing the overall distance. We formulate this task as a MIP problem, which can be tackled using standard solvers.

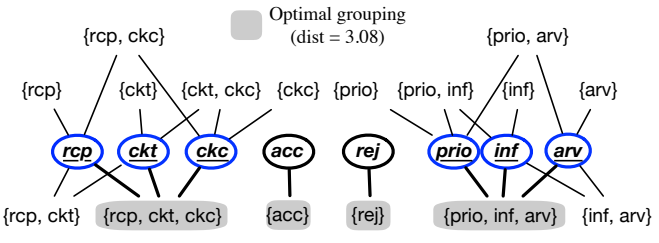

Fig. 7: Optimal grouping of the running example given all candidates computed in step 1 using the DFG-based approach.

Central to this formulation is a bipartite graph $(\mathcal{G}, C_L, E)$, which connects each candidate group to the event classes it covers, i.e., it contains an edge $(g_i, c_j) \in E$ if $c_j \in g_i$. Figure 7 visualizes this for the running example, in which the circled nodes in the middle indicate event classes in C_L, the sets indicate the candidate groups $\mathcal{G}$, and the edges their coverage relation. The grayed sets highlight the optimal grouping in this case, which is an exact cover because every event-class node is connected to exactly one of the selected groups.

Given this bipartite graph $(\mathcal{G}, C_L, E)$, we formalize a MIP problem with two decision variables:

- $selected_{g_i} \in \{0, 1\}$: 1 if $g_i \in \mathcal{G}$ is selected, else 0;
- $covered_{c_j} \in \{0, 1\}$: 1 if $c_j \in C_L$ is covered, else 0.

Then, we seek to minimize the distance of the selected groups in the objective function:

$$\text{minimize} \sum_{g_i \in \mathcal{G}} \texttt{dist}(g_i) * selected_{g_i}$$

This objective function is subject to two constraints:

$$\sum_{j=1}^{|C_L|} covered_{c_j} = |C_L| \tag{3}$$

$$\sum_{(g_i, c_j) \in E} selected_{g_i} = covered_{c_j}, \forall c_j \in C_L \tag{4}$$

These constraints jointly express that each event class shall be covered (Eq. 3), by exactly one group (Eq. 4). In case a user imposes grouping constraints in R_G, bounding the number of groups that may be selected, these are imposed by adding either or both of the following additional constraints:

$$\sum_{i=1}^{|\mathcal{G}|} selected_{g_i} \leq x \quad \text{resp.} \quad \sum_{i=1}^{|\mathcal{G}|} selected_{g_i} \geq y \tag{5}$$

The selected groups, i.e., $G = \{g_i \in \mathcal{G} : selected_{g_i} = 1\}$, then form the obtained grouping.

Note that, depending on the characteristics of log L and the imposed constraints R, a grouping may not be found, since there is no guarantee that a feasible solution exists. In that case, GECCO returns the initial log. To allow users to then refine their constraints appropriately, GECCO also indicates possible causes of the infeasibility, e.g., the affected event classes that lead to violations for constraints in R_C, or the fraction of cases for which constraints in R_I are violated. If a solution is found, GECCO continues with its third and final step.

Computational complexity. Most MIP problems are NP-hard, although an assessment of the exact complexity depends on the concrete problem and is poorly characterized by input size [11]. However, solvers like *Gurobi* are often able to solve MIP problems efficiently, by applying pre-solvers and heuristics. Our experiments confirm this, showing that Step 2 only contributes marginally to the overall runtime of GECCO.

D. Step 3: Creating an abstracted event log

Finally, we use the grouping G to establish abstracted versions of the traces in L to obtain an abstracted log L'.

For each trace $\sigma \in L$, we identify all activity instances in the trace, i.e., all instances of groups in G, $\mathcal{I}^\sigma =$

$\bigcup_{g \in G} \texttt{inst}(\sigma, g)$. Each activity instance, $\xi_i \in \mathcal{I}^\sigma$, corresponds to an ordered sequence of events, $\langle e_1, ..., e_k \rangle$.

Next, our approach creates an abstracted trace σ', which reflects the activity instances in $\mathcal{I}^\sigma$, instead of the events of its original counterpart, σ. A common abstraction strategy is to let σ' capture only the *completion* of activity instances, by creating a projection of σ that only retains the last event, e_k, per activity instance. For instance, for trace $\sigma_1 = \langle$rcp, ckc, acc, prio, inf, arv$\rangle$ of the running example, this abstraction would yield $\sigma_1^c = \langle$clrk1, acc, clrk2$\rangle$.

Yet, this strategy may obscure information when activities are executed in an interleaving manner. For this, consider a new trace $\sigma_5 = \langle$rcp, ckc, prio, acc, inf, arv$\rangle$. Here, events belonging to the *clrk2* group occur both before (*prio*) and after (*inf, arv*) the unary activity instance *acc*. When only retaining completion events, this yields the trace $\sigma_{5'}^c = \langle$clrk1, acc, clrk2$\rangle$, which hides the interleaving nature of the activities.

Therefore, we also propose an alternative strategy, which retains both the *start (s)* and *completion (c)* events per activity instance $\xi_i \in \mathcal{I}^\sigma$. This yields a trace $\sigma_{5'}^{s+c} = \langle clrk1_s, clrk1_c, clrk2_s$, acc, $clrk2_c \rangle$, which thus shows that activity *clrk2* starts before *acc* and completes afterwards.

The choice for a particular strategy depends on the relevance of parallelism in a particular analysis context, given that the latter strategy also leads to longer traces, thus partially mitigating the benefits of the obtained log abstraction.

The log L' that results from this last step represents the final output of GECCO. It is an event log in which the high-level activities are guaranteed to satisfy the user-defined constraints in R while providing a maximal degree of abstraction.

VI. Experimental Evaluation

We evaluated GECCO through evaluation experiments using a collection of real-world event logs. §VI-A outlines the evaluation setup. §VI-B reports on the results obtained for our approach and its configurations, whereas §VI-C compares our work against three baselines. Finally, §VI-D further illustrates the value of constraint-driven log abstraction through a case study. The employed implementation, evaluation pipelines, and additional experimental results are all publicly available.[1]

A. Evaluation Setup

Implementation and environment. We implemented our approach in Python, using *PM4Py* [12] for event log handling and *Gurobi* [13] as a solver for MIP problems. All experiments were conducted single-threaded on an Intel Xeon 2.6 GHz processor with up to 768GB of RAM available.

Data collection. We use a collection of 13 publicly-available event logs. To be able to cover various constraints, all logs have at least one categorical event attribute, as well as timestamps used for numerical constraints. As shown in Table III, the logs vary considerably in terms of key characteristics, such as the number of event classes, traces, and variants.

[1] https://gitlab.uni-mannheim.de/processanalytics/gecco

TABLE III: Properties of the real-life log collection.

Ref	$\lvert C_L \rvert$	Traces	Variants	$\lvert E \rvert$	Avg $\lvert \sigma \rvert$
[14]	11	150,370	231	70	3.73
[15]	40	75,928	3,453	357	6.35
[16]	39	46,616	22,632	772	10.01
[17]	24	31,509	5,946	180	16.41
[18]	39	14,550	8,627	407	52.48
[19]	24	13,087	4,366	125	20.04
[20]	8	10,035	1	14	15.00
[21]	51	7,065	1,478	553	12.25
[22]	4	1,487	183	10	4.47
[23]	27	1,434	116	99	5.98
[24]	16	1,050	846	115	14.49
[25]	70	902	295	124	24.00
[26]	29	20	20	164	69.70

Constraints. We use ten constraint sets in our experiments, covering the various constraint types that GECCO supports. Each set includes the class-based constraint $|g| \leq 8$, which is used to limit the number of abstraction problems that time out. This constraint is combined with each of the sets from Table IV, covering anti-monotonic (A), monotonic (M), and non-monotonic (N) instance-based constraints, a grouping constraint (Gr), as well as two sets of their combinations ($C1$ and $C2$). Table IV also contains additional constraints ($BL1$ to $BL4$) used in baseline comparisons, described below. By combining these constraint sets with the 13 event logs, we establish a total of 121 abstraction problems to be solved.[2]

TABLE IV: Constraints used in the experiments.

ID	Categories	Constraint(s)
A	R_I	$\lvert g.role \rvert \leq 3$
M	R_I	$\texttt{sum}(g.duration) \geq 10^1$
N	R_I	$\texttt{avg}(g.duration) \leq 5 * 10^5$
Gr	R_G	$\lvert G \rvert \leq 3$
$C1$	R_I, R_G	$A \wedge N \wedge Gr$
$C2$	R_I, R_G	$A \wedge M \wedge N \wedge Gr$
$BL1$	R_C	$\lvert g \rvert \leq 5$
$BL2$	R_C	$BL1 \wedge \texttt{cannotLink}(e_1.C, e_2.C)$
$BL3$	R_C	$\lvert g.D \rvert = 1$
$BL4$	R_G	$\lvert G \rvert = {}^{\lvert L \rvert}/_2$

Configurations. We test three configurations that differ in the instantiation of Step 1 of GECCO (cf., §V-B):

- **Exh**, using exhaustive candidate computation;
- **DFG$_\infty$**, using the DFG-based instantiation without beam search (i.e., unlimited beam width);
- **DFG$_k$**, using the DFG-based instantiation with a beam width that adapts to the number of event classes in the given log, i.e., $k = 5 * |C_L|$.

Note that we let candidate computation time out after 5 hours. GECCO then continues with the candidates identified so far.

Baselines. We compare GECCO against three baselines. These represent alternative approaches to solve the log-abstraction

[2] The class-based $BL3$ constraint can only be applied to 4 out of 13 logs, due to the absence of class-level attributes in the others.

problem and differ in the scope of constraints they can handle.[3]

*Graph querying (**BL**$_Q$).* GECCO's DFG-based candidate computation traverses a DFG to find candidate groups that adhere to imposed constraints. Recognizing the overlap of this with graph querying, BL$_Q$ replaces replaces Step 1 of GECCO with an instantiation using graph querying. For this, the DFG is stored in a graph database, which is queried for candidate groups using constraints formulated in a state-of-the-art graph querying language [27]. Given that a DFG captures a log on the class-level, BL$_Q$ can only support class-based constraints, though. Thus, we assess BL$_Q$ using a constraint on the maximum group size ($BL1$), an additional cannot-link constraint between event classes ($BL2$), and a constraint over a class-level attribute ($BL3$). By comparing against BL$_Q$, we aim to show that GECCO yields more comprehensive sets of candidate groups than those obtained by adopting existing solutions.

*Graph partitioning (**BL**$_P$).* GECCO's goal to find a disjoint set of cohesive groups for log abstraction is similar to the goal of graph partitioning, which aims to partition a graph such that edges between different groups have a low weight [28]. Therefore, we compare GECCO against a baseline using such partitioning, BL$_P$. Given a DFG, BL$_P$ aims to minimize the sum of directly-follows frequencies of cut edges, while cutting the graph into n partitions. For this, BL$_P$ applies *spectral partitioning* [28], where the weighted adjacency matrix is populated using normalized directly-follows frequencies. Since graph partitioning simply splits a DFG into a certain number of groups, BL$_P$ can only support strict grouping constraints, whereas instance-based, class-based, and flexible grouping constraints (e.g., constraint Gr), cannot be handled. Therefore, we compare BL$_P$ against GECCO using the constraint $BL4$, which aims to reduce number of event classes of a log by half. This comparison aims to show that GECCO's three-step approach leads to better log-abstraction results, while also supporting a considerably broader range of constraints.

*Greedy approach (**BL**$_G$).* Finally, we compare GECCO against a greedy abstraction strategy. BL$_G$ starts by assigning all event classes from C_L to a set of singleton groups, G_0. Then, in each iteration, BL$_G$ merges those two groups from G_i that lead to the lowest overall distance, i.e., $\texttt{dist}(G_{i+1}, L)$, without resulting in any constraint violations. BL$_G$ stops if the overall distance cannot improve in an iteration. Unlike the other baselines, BL$_G$ can handle instance-based constraints, since it works directly on the event log rather than the DFG, although, grouping constraints cannot be enforced in this iterative strategy. Therefore, we compare BL$_G$ against GECCO using the instance-based constraint set A, M, and N. This comparison against BL$_G$ aims to show the importance of striving for a global optimum in the log-abstraction problem.

Measures. To assess the results obtained by the various configurations and baselines, we consider the following measures:

*Solved abstraction problems (**Solved**).*: We report on the fraction of solved problems, to reflect the general feasibility of abstraction problems and the ability of a specific configuration to find such feasible solutions.

*Size reduction (**S. red.**).*: We measure the obtained size reduction by comparing the number of high-level activities in an obtained grouping to the number of original event classes, i.e., $|G|/|C_L|$. Given the strong link between model size and process understandability [29], this measure provides a straightforward but clear quantification of the abstraction degree.

*Complexity reduction (**C. red.**).*: We also assess the abstraction degree through the reduction in *control-flow complexity*, using an established complexity measure [29]. Since this measure requires a process model as input, we discover a model for both the original and the abstracted log using the state-of-the-art *Split Miner* [30] and then compare their complexity.

*Silhouette coefficient (**Sil.**).*: We quantify the intra-group cohesion and inter-group separation of a grouping G using the *silhouette coefficient* [31], an established measure for cluster quality. To avoid bias, we compute this coefficient using a standard measure for the pair-wise distance between event classes [32], which considers their average positional distance.

*Runtime (**T(m)**).*: Finally, we measure the time in minutes required to obtain an abstraction result, from the moment a log L is imported until the abstracted log L' is returned.

B. Evaluation Results

This section reports on the results for the different constraint sets, followed by a comparison of the different configurations.

Overall results. Table V presents the results obtained using the Exh configuration of GECCO per constraint set. For the anti-monotonic (A, $BL1$-3) and grouping constraint sets (Gr,$BL4$) GECCO finds a solution to all of the problems. Infeasible problems primarily occur for the monotonic M constraint set and the combination sets, $C1$ and $C2$, since these are more restrictive. Interestingly, $C1$ has more than twice as many solved problems (54%) than $C2$ (23%), clearly showing the impact of $C2$'s additional monotonic constraint on feasibility.

The other measures in the table report on the results obtained for the solved problems. We observe that GECCO achieves a considerable degree of abstraction, reflected in the reductions in size and complexity. Groupings are reasonably cohesive and well separated from each other, indicated by silhouette coefficients ≥ 0.12. These results are stable for the less restrictive constraint sets, such as A, N, and Gr, as well as their combination $C1$. For instance, for A a size reduction of 0.68, complexity reduction of 0.63, and silhouette coefficient of 0.15 is achieved. In line with expectations, for more restrictive constraint sets, e.g., *C2*, the impact of abstraction is less significant (0.50, 0.40, and 0.09 resp.). Finally, the impact of the constraint-checking modes on efficiency can also be observed.[4] For instance, while the *Gr* constraint set requires 144m on average to be solved, the anti-monotonic $BL2$ constraint cases are solved in 121m. In this mode candidates do not have to be expanded if they already violate the constraint, which leads to improved runtimes.

[3]More details on the baselines and their implementation can be found in our repository linked in §VI-A

[4]For constraint sets with unsolved problems, runtimes must be compared carefully, as they strongly depend on the specific logs with feasible solutions.

Overall, GECCO is thus able to greatly reduce the size and complexity of event logs, while respecting various constraints. Although the solution feasibility and the abstraction degree depends on the employed constraints, GECCO consistently finds groups that have strong cohesion and good separation.

TABLE V: Results for Exh, averaged over solved problems.

Const.	Solved	S. red.	C. red.	Sil.	T(m)
A	1.00	0.68	0.63	0.15	146
M	0.31	0.58	0.55	0.15	75
N	0.77	0.68	0.65	0.12	154
Gr	1.00	0.66	0.61	0.13	144
$C1$	0.54	0.68	0.59	0.12	134
$C2$	0.23	0.50	0.40	0.09	100
$BL1$	1.00	0.67	0.61	0.12	122
$BL2$	1.00	0.66	0.61	0.12	121
$BL3$	1.00	0.38	0.29	-0.02	38
$BL4$	1.00	0.51	0.46	0.05	147

Exhaustive versus efficient configurations. Table VI depicts the evaluation results for the three GECCO configurations, again providing the averages over the solved problems. Notably, the configurations were able to solve the same problems, except for a single problem in the non-monotonic N constraint set, which the DFG_k configuration failed to solve.

We observe that the DFG-based configurations achieve substantial efficiency gains in comparison to the exhaustive one, where in particular DFG_k needs only about 40% of the time in comparison to Exh (49m vs. 130m on average).

With respect to the abstraction degree, we observe that DFG_∞ maintains results comparable to Exh for size (0.62 vs. 0.63) and complexity reduction (0.56 vs. 0.57). It even obtains better results for the silhouette coefficient (0.16 vs. 0.11), which shows the ability of the DFG-based approach to find candidate groups that are cohesive and well-separated. The results achieved by DFG_k suggest a trade-off between optimal abstraction and efficiency, as the abstraction degree is about 7% lower compared to the other configurations.

Finally, we observe that the DFG-based configurations are particularly useful for anti-monotonic and grouping constraints. In these cases, the results differ only marginally, even for DFG_k, while achieving considerable efficiency gains.

TABLE VI: Results per configuration over solved problems.

Conf.	Solved	S. red.	C. red.	Sil.	T(m)
Exh	0.78	0.63	0.57	0.11	130
DFG_∞	0.78	0.62	0.56	0.16	108
DFG_k	0.77	0.56	0.50	0.08	49

C. Baseline results

Table VII depicts the results obtained using the baseline approaches against the most relevant configurations of GECCO.

Comparison to graph querying. The results of BL_Q indicate that the candidate groups obtained using graph queries are not as comprehensive as those found by GECCO's DFG_∞ configuration. BL_Q's solutions are therefore subpar with respect to size and complexity reduction. Furthermore, the negative silhouette coefficients (-0.2 avg. vs. 0.17 for DFG_∞) indicate that the groupings found by BL_Q are neither cohesive nor separated, which highlights the ability of DFG-based candidate computation to find better sets of candidates for abstraction.

Comparison to graph partitioning. With respect to BL_P we find that partitioning the DFG by minimizing edge cuts naturally reduces the size of the DFG and, thus, achieves a certain degree of abstraction. However, the groupings created by BL_P are not as cohesive, indicated by the silhouette coefficient of 0.01 compared to GECCO (0.05). Moreover, the complexity reduction achieved by BL_P (0.41) is lower than achieved by GECCO (0.46), even though their groupings contain the same number of activities. This highlights the benefits of the three-step approach GECCO takes and the suitability of its distance measure to obtain meaningful groupings for log abstraction.

Comparison to greedy approach. When considering the results of BL_G, the downsides of a greedy solution strategy quickly become apparent. BL_G finds solutions to fewer abstraction problems (64%) than even the most efficient configuration, DFG_k, whereas the solutions that are identified are far subpar. For example, for the anti-monotonic A constraint set, BL_G achieves an average size reduction of 0.47, whereas DFG_k yields a size reduction of 0.64, which clearly shows that a greedy strategy often yields solutions that are far from optimal.

TABLE VII: Baseline comparison over the applicable constraint sets. Results are averaged over solved problems.

Const.	Conf.	Solved	S. red.	C. red.	Sil.	T(m)
BL[1-3]	**DFG_∞**	1.00	0.63	0.55	0.17	77
	BL_Q	0.96	0.55	0.43	-0.20	24
$BL4$	**Exh**	1.00	0.51	0.46	0.05	147
	BL_P	1.00	0.51	0.42	0.01	1
A, M, N	**DFG_k**	0.67	0.59	0.52	0.08	58
	BL_G	0.64	0.45	0.37	0.02	24

Discussion. Overall, these results demonstrate that GECCO outperforms all three baselines with respect to their applicable constraints, whereas it can, furthermore, handle a much broader range of process-oriented abstraction constraints.

D. Case Study

In this section we apply GECCO in a case study to give an illustration of the value of constraint-driven log abstraction.

We use an event log [33] capturing a loan application process at a large financial institution. Although the log only contains 24 event classes, its complexity is considerable, as evidenced by its 160 DFG edges. As shown in Figure 1 this issue even remains for a so-called 80/20 model, which omits the 20% least frequent edges, since this visualization still provides few useful insights into the underlying process.

We recognize that the needed log abstraction can here be guided by considering the three IT systems from which events

in the log originate: an *application-handling* system (A), the *offer* system (O), and a general *workflow* system (W). Since these systems each relate to a distinct part of a process, we impose a constraint that avoids mixing up events from different systems into a single activity, i.e., $|g.origin| \leq 1$.

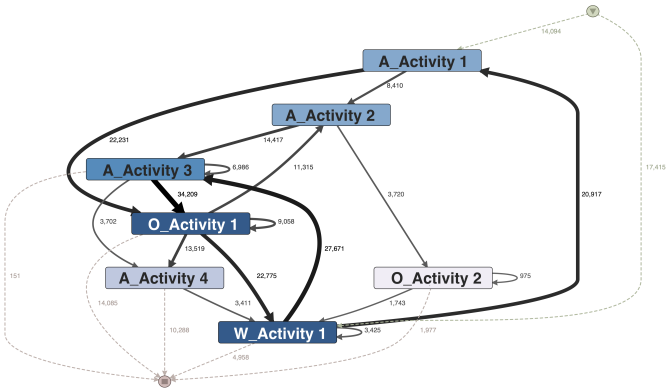

Fig. 8: 80/20 DFG of the abstracted loan application log.

Figure 8 depicts the DFG obtained by GECCO in this manner, where the activity labels reflect their origin systems. Having grouped the events into seven high-level activities, the DFG shows a considerable reduction in terms of size and complexity. Due to this simplification, we observe clear inter-relations between the different sub-systems. For instance, the process most often starts with the execution of steps in the application-handling system (*A_Activity 1* to *3*), followed by a part in the offer system (*O_Activity 1*), and again concluded in the application-handling system (*A_Activity 4*). Next to this main sequence, the workflow-related steps (*W_Activity 1*) occur in parallel to the other activities, whereas the refusal of an offer (*O_Activity 2*) represents a clear alternative path.

It is important to stress that such insights are only possible due to the constraint-driven nature of our work. In fact, when applying GECCO without imposing any constraints, the intertwined nature of the process even yielded high-level activities that contain events from all three sub-systems, thus obfuscating the key inter-relations in the process.

VII. Related Work

Our work primarily relates to the following streams:

Log abstraction in process mining. In the context of process mining, a broad range of techniques have been developed for log abstraction, also referred to as *event abstraction*, of which Van Zelst et al. [5] and Diba et al. [6] recently provided comprehensive overviews. Unsupervised techniques mostly employ clustering [34], [35] or generic abstraction patterns [36], [37] to group low-level events into high-level activities. Other techniques are supervised, requiring users to provide information about the high-level activities to be discovered, captured ,e.g., in the form of a process model [38], specific event annotations [39], or domain hierarchies [40].

Compared to GECCO, existing unsupervised techniques do not guarantee any characteristics for the abstracted logs, which can result in a considerable loss of information (cf., §VI-D) , while the supervised techniques require a user to explicitly specify *how* abstraction should be performed, whereas our work only needs a specification of *what* properties they desire.

Behavioral pattern mining from event logs. Behavioral pattern mining also lifts low-level events to a higher degree of abstraction by identifying interesting patterns in event logs, including constructs such as exclusion, loops, and concurrency. Local Process Models (LPMs) [41] provide an established foundation for this. LPMs are mined according to pattern frequency, while extensions have been proposed to employ interest-aware utility functions [42] and incorporate user constraints [43]. Behavioral pattern mining has been also addressed through the discovery of maximal and compact patterns in logs [44] and their context-aware extension [45].

While their purposes are similar, a key difference between behavioral pattern mining and log abstraction is that the former *cherry-picks* interesting parts from an event log, whereas the latter strives for comprehensive abstraction over an entire log.

Sequential pattern mining. Approaches for sequential pattern mining [46], [47] identify interesting patterns in sequential data. As for LPMs, *interesting* is typically defined as *frequent* [48], while techniques for *high-utility sequential pattern mining* also support utility functions specific to the data attributes of events [49]–[51]. Cohesion, comparable to distance measures used in log abstraction, has also been applied as a utility measure for pattern mining in single long sequences [52], [53]. Furthermore, research in *constrained sequential pattern mining* primarily focuses on exploiting constraint characteristics such as monotonicity to improve efficiency [54], which we leverage during the first step of our approach as well. Frequently the focus is on specific constraints, e.g., time-based gap constraints [55].

In contrast to GECCO, pattern mining techniques do not consider concurrency and exclusion. Moreover, like for behavioral pattern mining, the focus is on the identification of individual patterns, while our work strives for global abstraction.

VIII. Conclusion

This paper proposed the GECCO approach for constraint-driven abstraction of event logs. With GECCO, users can define the desired characteristics and requirements of abstracted logs in terms of constraints, thus enabling the meaningful and purpose-driven abstraction of low-level event data. We provide two primary instantiations of our approach, allowing users to trade-off computational efficiency and result optimality. Our evaluation experiments using real-life event logs reveal that our work considerably outperforms baseline techniques, whereas a case study demonstrates its usefulness in practical settings.

There are several promising directions for future research. We aim to extend the scope of our work with instance-based constraints over the entire grouping (rather than per group). Furthermore, we aim to develop an approach to suggest interesting constraints to users for a given log. Finally, we plan to lift our work to online settings, so that identified groupings are dynamically adapted to new arrivals in a stream.

References

[1] W. M. P. van der Aalst, *Process Mining: Data Science in Action*. Berlin / Heidelberg: Springer, 2016.

[2] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo, "Automated discovery of process models from event logs: Review and benchmark," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 4, pp. 686–705, 2019. [Online]. Available: https://doi.org/10.1109/TKDE.2018.2841877

[3] A. Senderovich, A. Rogge-Solti, A. Gal, J. Mendling, and A. Mandelbaum, "The ROAD from sensor data to process instances via interaction mining," in *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, ser. Lecture Notes in Computer Science, S. Nurcan, P. Soffer, M. Bajec, and J. Eder, Eds., vol. 9694. Springer, 2016, pp. 257–273. [Online]. Available: https://doi.org/10.1007/978-3-319-39696-5_16

[4] V. Leno, A. Augusto, M. Dumas, M. La Rosa, F. M. Maggi, and A. Polyvyanyy, "Identifying candidate routines for robotic process automation from unsegmented ui logs," in *2020 2nd International Conference on Process Mining (ICPM)*. IEEE, 2020, pp. 153–160.

[5] S. J. van Zelst, F. Mannhardt, M. de Leoni, and A. Koschmider, "Event abstraction in process mining: literature review and taxonomy," *Granular Computing*, vol. 2, 2020. [Online]. Available: https://doi.org/10.1007/s41066-020-00226-2

[6] K. Diba, K. Batoulis, M. Weidlich, and M. Weske, "Extraction, correlation, and abstraction of event data for process mining," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 10, no. 3, pp. 1–31, 2020.

[7] M. De Leoni and S. Dündar, "Event-log abstraction using batch session identification and clustering," *Proceedings of the ACM Symposium on Applied Computing*, pp. 36–44, 2020.

[8] F. Mannhardt, M. de Leoni, H. A. Reijers, W. M. van der Aalst, and P. J. Toussaint, "Guided process discovery – a pattern-based approach," *Information Systems*, vol. 76, pp. 1–18, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437916306299

[9] H. van der Aa, A. Rebmann, and H. Leopold, "Natural language-based detection of semantic execution anomalies in event logs," *Information Systems*, vol. 102, p. 101824, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030643792100065X

[10] C. Wilt, J. Thayer, and W. Ruml, "A comparison of greedy search algorithms," *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SoCS 2010*, pp. 129–136, 2010.

[11] T. J. Van Roy and L. A. Wolsey, "Solving mixed integer programming problems using automatic reformulation," *Oper. Res.*, vol. 35, no. 1, pp. 45–57, Feb. 1987.

[12] A. Berti, S. J. van Zelst, and W. van der Aalst, "Process mining for python (PM4Py): Bridging the gap between process-and data science," in *ICPM Demo Track 2019*, 2019, pp. 13–16.

[13] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: https://www.gurobi.com

[14] M. M. de Leoni and F. Mannhardt, "Road traffic fine management process," Feb 2015. [Online]. Available: https://data.4tu.nl/articles/dataset/Road_Traffic_Fine_Management_Process/12683249/1

[15] B. van Dongen, "Bpi challenge 2019," Jan 2019. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2019/12715853/1

[16] ——, "Bpi challenge 2014: Activity log for incidents," Apr 2014. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2014_Activity_log_for_incidents/12706424/1

[17] ——, "Bpi challenge 2017," Feb 2017. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884/1

[18] B. van Dongen and F. F. Borchert, "Bpi challenge 2018," Mar 2018. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2018/12688355/1

[19] B. van Dongen, "Bpi challenge 2012," Apr 2012. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2012/12689204/1

[20] A. Djedović, "Credit requirement event logs," Sep 2017. [Online]. Available: https://data.4tu.nl/articles/dataset/Credit_Requirement_Event_Logs/12693005/1

[21] B. van Dongen, "Bpi challenge 2020: Travel permit data," Mar 2020. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Travel_Permit_Data/12718178/1

[22] W. Steeman, "Bpi challenge 2013, closed problems," Apr 2013. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2013_closed_problems/12714476/1

[23] J. Buijs, "Environmental permit application process ('wabo'), coselog project," May 2014. [Online]. Available: https://data.4tu.nl/collections/Environmental_permit_application_process_WABO_CoSeLoG_project/5065529/1

[24] F. Mannhardt, "Sepsis cases - event log," Dec 2016. [Online]. Available: https://data.4tu.nl/articles/dataset/Sepsis_Cases_-_Event_Log/12707639/1

[25] A. A. Augusto, R. R. Conforti, M. M. Dumas, M. M. La Rosa, F. F. M. Maggi, A. A. Marrella, M. M. Mecella, and A. A. Soo, "Data underlying the paper: Automated discovery of process models from event logs: Review and benchmark, bpic 2015 log 1," Jun 2019. [Online]. Available: https://data.4tu.nl/articles/dataset/Data_underlying_the_paper_Automated_Discovery_of_Process_Models_from_Event_Logs_Review_and_Benchmark/12712727/1

[26] J. Munoz-Gama, R. R. de la Fuente, M. M. Sepúlveda, and R. R. Fuentes, "Conformance checking challenge 2019 (ccc19)," Feb 2019. [Online]. Available: https://data.4tu.nl/articles/dataset/Conformance_Checking_Challenge_2019_CCC19_/12714932/1

[27] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. [Online]. Available: https://doi.org/10.1145/3183713.3190657

[28] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[29] H. A. Reijers and J. Mendling, "A study into the factors that influence the understandability of business process models," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 41, no. 3, pp. 449–462, 2010.

[30] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and A. Polyvyanyy, "Split miner: automated discovery of accurate and simple business process models from event logs," *Knowledge and Information Systems*, vol. 59, no. 2, pp. 251–284, 2019.

[31] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[32] C. W. Günther and W. M. P. van der Aalst, "Fuzzy mining – adaptive process simplification based on multi-perspective metrics," in *Business Process Management*, G. Alonso, P. Dadam, and M. Rosemann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 328–343.

[33] B. van Dongen, "Bpi challenge 2017," Feb 2017. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884/1

[34] F. Folino, M. Guarascio, and L. Pontieri, "Mining multi-variant process models from low-level logs," in *International Conference on Business Information Systems*. Springer, 2015, pp. 165–177.

[35] J.-R. Rehse and P. Fettke, "Clustering business process activities for identifying reference model components," in *International Conference on Business Process Management*. Springer, 2018, pp. 5–17.

[36] R. P. Jagadeesh Chandra Bose and W. M. P. van der Aalst, "Abstractions in process mining: A taxonomy of patterns," in *Business Process Management*, U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 159–175.

[37] B. Wiegand, D. Klakow, and J. Vreeken, "Mining Easily Understandable Models from Complex Event Logs," in *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, 2021, pp. 244–252.

[38] T. Baier, J. Mendling, and M. Weske, "Bridging abstraction layers in process mining," *Information Systems*, vol. 46, pp. 123–139, 2014.

[39] S. J. Leemans, K. Goel, and S. J. van Zelst, "Using multi-level information in hierarchical process mining: Balancing behavioural quality and model complexity," in *2020 2nd International Conference on Process Mining (ICPM)*, 2020, pp. 137–144.

[40] F. Klessascheck, T. Lichtenstein, M. Meier, S. Remy, J. Sachs, L. Pufahl, R. Miotto, E. P. Böttinger, and M. Weske, "Domain-specific event abstraction," in *24th International Conference on Business Information Systems, BIS 2021, Hannover, Germany, June 15-17, 2021*, W. Abramowicz, S. Auer, and E. Lewanska, Eds., 2021, pp. 117–126. [Online]. Available: https://doi.org/10.52825/bis.v1i.39

[41] N. Tax, N. Sidorova, R. Haakma, and W. M. van der Aalst, "Mining local process models," *Journal of Innovation in Digital*

Ecosystems, vol. 3, no. 2, pp. 183–196, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.jides.2016.11.001

[42] N. Tax, B. Dalmas, N. Sidorova, W. M. van der Aalst, and S. Norre, "Interest-driven discovery of local process models," *Information Systems*, vol. 77, pp. 105–117, 2018. [Online]. Available: https://doi.org/10.1016/j.is.2018.04.006

[43] N. Tax, N. Sidorova, R. Haakma, and W. M. Van Der Aalst, "Mining Local Process Models with Constraints Efficiently: Applications to the Analysis of Smart Home Data," *Proceedings - 2018 International Conference on Intelligent Environments, IE 2018*, pp. 56–63, 2018.

[44] M. Acheli, D. Grigori, and M. Weidlich, *Efficient Discovery of Compact Maximal Behavioral Patterns from Event Logs*. Springer International Publishing, 2019, vol. 11483 LNCS. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-21290-2_36

[45] ——, "Discovering and analyzing contextual behavioral patterns from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. Early access, 2021.

[46] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the eleventh international conference on data engineering*. IEEE, 1995, pp. 3–14.

[47] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *IEEE Transactions on knowledge and data engineering*, vol. 16, no. 11, pp. 1424–1440, 2004.

[48] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data mining and knowledge discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[49] T. Truong-Chi and P. Fournier-Viger, "A survey of high utility sequential pattern mining," in *High-Utility Pattern Mining*. Springer, 2019, pp. 97–129.

[50] J. Yin, Z. Zheng, L. Cao, Y. Song, and W. Wei, "Efficiently mining top-k high utility sequential patterns," in *2013 IEEE 13th international conference on data mining*. IEEE, 2013, pp. 1259–1264.

[51] J. Yin, Z. Zheng, and L. Cao, "Uspan: an efficient algorithm for mining high utility sequential patterns," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 660–668.

[52] B. Cule, B. Goethals, and C. Robardet, "A new constraint for mining sets in sequences," in *Proceedings of the 2009 SIAM international conference on data mining*. SIAM, 2009, pp. 317–328.

[53] B. Cule, L. Feremans, and B. Goethals, "Efficient discovery of sets of co-occurring items in event sequences," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9851 LNAI, pp. 361–377, 2016.

[54] J. Pei, J. Han, and W. Wang, "Constraint-based sequential pattern mining: The pattern-growth methods," *Journal of Intelligent Information Systems*, vol. 28, no. 2, pp. 133–160, 2007.

[55] J. O. Aoga, T. Guns, and P. Schaus, "Mining time-constrained sequential patterns with constraint programming," *Constraints*, vol. 22, no. 4, pp. 548–570, Oct. 2017. [Online]. Available: https://doi.org/10.1007/s10601-017-9272-3