Modeling and Discovering Cancelation Behavior

Maikel Leemans(✉) and Wil M.P. van der Aalst

Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands
m.leemans@tue.nl

Abstract. This paper presents a novel extension to the process tree model to support cancelation behavior, and proposes a novel process discovery technique to discover sound, fitting models with cancelation features. The proposed discovery technique relies on a generic error oracle function, and allows us to discover complex combinations of multiple, possibly nested cancelation regions based on observed behavior. An implementation of the proposed approach is available as a ProM plugin. Experimental results based on real-life event logs demonstrate the feasibility and usefulness of the approach.

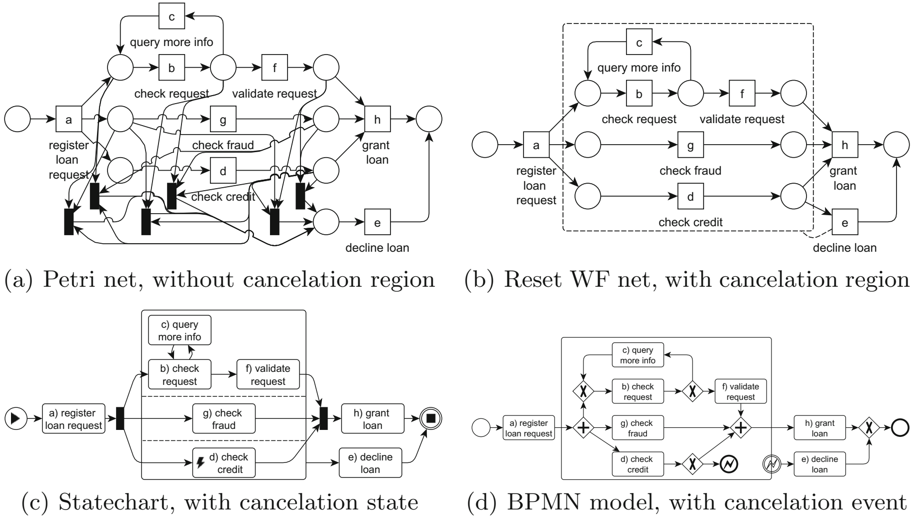
Keywords: Process mining · Process discovery · Cancelation discovery · Cancelation modeling · Process trees · Event logs · Reset nets

1 Introduction

Process mining provides a powerful way to discover and analyze operational processes based on recorded event data stored in *event logs*. These event logs can be found everywhere: in enterprise information systems and business transaction logs, in web servers, in high-tech systems such as X-ray machines, in warehousing systems, etc. [17]. The majority of such real-life event logs contain some form of cancelation or error-handling behavior. A bank loan request may be canceled or declined, a webserver needs to handle a connection error, an X-ray machine may detect a sensor problem, etc. These cancelations can easily be expressed in workflow languages (BPMN, YAWL) and formal models such as reset workflow nets (RWF-nets). When formal process descriptions are not available, outdated or otherwise inaccurate, we turn to *process discovery* techniques. Process discovery aims to learn a process model from example behavior in event logs. Many discovery techniques have been proposed in literature, but few take into account cancelation features.

To illustrate the need for cancelation features, consider a bank loan request example, as modeled in Fig. 1. After a request is registered (a), in parallel the client's credit is checked (d), the request is processed (b, c, f), and a fraud check is performed (g). Once all parallel branches succeed, the loan is granted (h). If the credit check failed, the loan is declined (e), and there is no need to wait for the other activities. Since the credit check can fail at any stage during the request processing and fraud checking, there are $3 \cdot 2 = 6$ scenarios where a decline loan has to be modeled (see the six black transitions in Fig. 1(a)).

© Springer International Publishing AG 2017
H. Panetto et al. (Eds.): OTM 2017 Conferences, Part I, LNCS 10573, pp. 93–113, 2017.
https://doi.org/10.1007/978-3-319-69462-7_8



(a) Petri net, without cancelation region (b) Reset WF net, with cancelation region

(c) Statechart, with cancelation state (d) BPMN model, with cancelation event

Fig. 1. Small loan application example with cancelation behavior, modeled in different languages, illustrating the advantage of using cancelation features. As we will show in this paper, these models can be compactly represented using the proposed cancelation process tree (see Definition 4): $\to(a, \overset{\star}{\to}(\to(\wedge(\to(\circlearrowleft(b, c), f), g, \star_d^{\{e\}}), h), e))$

The model in Fig. 1(a) is already rather complex. However, we assumed that the activities are atomic. This is not realistic, because the check credit, request processing and check fraud happen in parallel, and if the check fails, any processing tasks need to be withdrawn. Assuming we also model the start and end for each activity (thus modeling the running state of an activity), the number of scenarios/black transitions increases to $6 \cdot 3 = 18$. Using the cancelation features available in the various languages, we get a much simpler and more precise model, as shown in Figs. 1(b), (c) and (d).

Clearly, there is a need to model cancelation features explicitly, and process discovery should be able to discover these cancelation features. Furthermore, it is important that the discovered model meets certain quality criteria. Obviously, the model should be sound, i.e., all process steps can be executed and an end state is always reachable. Moreover, it is desirable to discover models that can replay all the behavior in the event log, i.e., to discover fitting models that relate to the event log. Recent work proposed a framework for discovery algorithms which find sound, fitting models in finite time, based on process trees [14]. The process tree notation is tailored towards process discovery and is a compact way to represent block-structured models that can easily be represented in terms of, for example: workflow nets, statecharts and BPMN models. However, in its current form, process trees cannot effectively capture cancelation features.

In this paper, *we propose a novel extension to the process tree model to support cancelation behavior, and propose a novel process discovery technique to discover sound, fitting models with cancelation features.* The proposed

discovery technique relies on a generic error oracle function, and allows us to discover complex combinations of multiple, possibly nested cancelation regions based on observed behavior.

The approach is outlined in Fig. 2. An implementation of the proposed algorithm is tested and made available via the *Statechart* plugin for the ProM framework [12].

The remainder of this paper is organized as follows. Section 2 positions the work in existing literature. Section 3 introduces formal definitions and the proposed cancelation model. In Sect. 4, we discuss several heuristics for our error oracle. The novel cancelation process discovery technique is explained in Sect. 5. The approach is evaluated in Sect. 6. Section 7 concludes the paper.

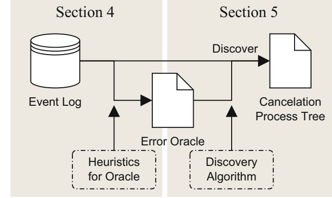


Fig. 2. Approach outline.

2 Related Work

To relate our work to existing approaches in process mining, we provide a systematic comparison of discovery approaches based on four criteria (Sect. 2.1). That is, the approach should provide expressive and sound models, and allow for a trade-off between fitness and simplicity. Next, we discuss and compare the related work (Sect. 2.2). Table 1 summarizes the comparison.

2.1 Criteria for Comparison

For comparing the related work, we define several comparison criteria.

As a basis, any discovery algorithm should yield *sound* process models. That is, all process steps can be executed and an end state is always reachable. In addition, it is desirable to be able to discover models that can replay all the behavior in the event log, i.e., to discover *fitting* models that relate to the event log [17].

Real-life event logs are often messy and challenging for discovery algorithms. In these cases, it may be necessary to trade off fitness for simplicity by filtering out *infrequent* behavior. In addition, some behavior can only be captured by *non-local* constructs such as long-distance dependencies or, to a degree, cancelation features.

For the related work techniques that do discover cancelation features, we look into *how the corresponding cancelation regions are discovered* from the event log. In addition, we will also compare on the complexity of the discovered features. That is, if *multiple regions* can be discovered, possibly in a *nested* configuration.

2.2 Discussion of the Related Work

We divided the related work discussion into several groups based on their comparison characteristics, see also Table 1.

Table 1. Comparison of related techniques from Sect. 2, according to the comparison criteria detailed in Sect. 2.

	Author	Algorithm	Formalism	Sound	Fitting	Infrequent	Non-local	Cancel disc.	#Regions	Nested
[20]	Aalst, van der	Alpha miner	Petri net	-	-	-	-	-	-	-
[19]	Aalst, van der	TS Regions	Petri net	-	-	-	✓	-	-	-
[24]	Weijters	(Flexible) Heuristics miner	Heuristics net	-	-	✓	✓	-	-	-
[18]	Alves de Medeiros	Genetic miner	Heuristics net	-	-	✓	✓	-	-	-
[2]	Augusto	Structured miner	BPMN	-	-	✓	✓	-	-	-
[21]	Werf, van der	ILP miner	Petri net	-	✓	-	✓	-	-	-
[23]	Zelst, van	ILP with filtering	Petri net	-	✓	✓	✓	-	-	-
[5]	Carmona	Genet	Petri net	-	✓	✓	✓	-	-	-
[15]	Redlich	Constructs Competition miner	BP(MN)	✓	-	✓	-	-	-	-
[4]	Buijs	ETMd miner	Process tree	✓	-	✓	-	-	-	-
[14]	Leemans, S.J.J.	Inductive miner (IM)	Process tree	✓	✓	✓	-	-	-	-
[10]	Fluxicon / Günther	Disco	Fuzzy Model†	n/a	n/a	✓	-	-	-	-
[6]	Celonis GmbH	Celonis Process Mining	Fuzzy Model†	n/a	n/a	✓	-	-	-	-
[9]	Gradient ECM	Minit	Fuzzy Model†	n/a	n/a	✓	-	-	-	-
[11]	Kalenkova	TS Cancel	RWF-net	-	-	-	±	✓	1	-
[16]	Aalst, van der	Generic post-processing	RWF-net	-	-	-	±	±	1	-
[7]	Conforti	BPMN miner	BPMN	✓	-	✓	±	±	n	✓
	This paper	Cancelation Discovery	C. process tree*	✓	✓	✓	±	✓	n	✓

† Fuzzy models have no executable semantics. * Cancelation process tree, see Definition 4.

There is a wide variety of Petri net based algorithms that can discover non-local constructs [2,5,18,19,21,23,24]. Even though these algorithms do not support cancelation features, they can discover and model complex, real-life behavior. The major downside is that none of these techniques guarantee a sound model.

A small group of algorithms focus on the process tree representation [4,14]. The use of process trees guarantees the discovery of sound workflow nets. However, by design, this limits the discovery search space to models with structured, local features only. Less structured behavior, like cancelation features, cannot be modeled or discovered.

In recent years, several commercial process mining tools emerged on the market [6,9,10]. Compared to academic tools, these commercial tools are easier to use, but provide less functionality. In particular, the commercial models provide no executable semantics, and do not support concurrency [17].

There have been a few attempts at supporting the discovery of cancelation features. In the work of [11], cancelation discovery is based on the behavior found in the event log. By analyzing a transition system (TS) abstraction of the event log, [11] searches for a single cancelation region. In contrast, the technique outlined in [16] uses a post-processing strategy based on conformance techniques.

That is, given a discovered model (using an existing algorithm), it tries to determine where a cancelation region should have been, based on unsuccessful event log replays (i.e., remaining tokens in the Petri net). In the work of [7], another post-processing heuristic is proposed. It assumes the underlying discovery algorithm (the BPMN miner) can correctly identify subprocesses (based on the relations in additional data attributes), and then checks which of these subprocesses should be "upgraded" to a cancelation region.

3 Event Logs and Process Trees

Before we explain the proposed discovery technique, we first introduce some definitions and our novel extension to process trees. We start with some preliminaries in Subsect. 3.1. In Subsect. 3.2 we introduce event logs (our input). Finally, in Subsects. 3.3 and 3.4, we will discuss the process tree model and our novel extension: *cancelation process trees*.

3.1 Preliminaries

We denote the powerset over some set A as $\mathcal{P}(A)$. We denote the set of all multisets over some set A as $\mathcal{B}(A)$. Note that the ordering of elements in a set or multiset is irrelevant.

Given a set X, a sequence over X of length n is denoted as $t = \langle a_1, \ldots, a_n \rangle \in X^*$. The empty sequence is denoted as ε, and we define $head(t) = a_1$, $end(t) = a_n$. We define $a \in t$ iff there is at least one a_i such that $a_i = a$. We write $\cdot$ to denote sequence concatenation, for example: $\langle a \rangle \cdot \langle b, c \rangle = \langle a, b, c \rangle$, and $\langle a \rangle \cdot \varepsilon = \langle a \rangle$. We write $\diamond$ to denote sequence interleaving (shuffle). For example: $\langle a, b \rangle \diamond \langle c, d \rangle = \{ \langle a, b, c, d \rangle , \langle a, c, b, d \rangle , \langle a, c, d, b \rangle , \langle c, a, b, d \rangle , \langle c, a, d, b \rangle , \langle c, d, a, b \rangle \}$.

We write $f : X \mapsto Y$ for a function with domain $dom(f) = X$ and range $rng(f) = \{ f(x) \mid x \in X \} \subseteq Y$.

3.2 Event Logs

The starting point for any process mining technique is an *event log*, a set of *events* grouped into *traces*, describing what happened when. Each trace corresponds to an execution of a process. Events may be characterized by various *attributes*, e.g., an event may have a timestamp, correspond to an activity, denote a start or end, is executed by a particular resource, etc.

For the sake of clarity, we will ignore most event attributes, and use sequences of activities directly, as defined below.

Definition 1 (Event Log). Let $\mathbb{A}$ be a set of activities. Let $L \in \mathcal{B}(\mathbb{A}^*)$ be an event log, a multiset of traces. A trace $t \in L$, with $t \in \mathbb{A}^*$, is a sequence of activities. ⌟

Given a set $\Sigma \subseteq \mathbb{A}$ and trace $t \in L$, we write $\Sigma(t) = \Sigma \cap \{ a \in t \}$ to denote the set of activities in the intersection of Σ and t.

3.3 Process Trees

In this subsection, we introduce *process trees* as a notation to compactly represent *block-structured models*. An important property of block-structured models is that they are *sound by construction*; they do not suffer from deadlocks, livelocks and other anomalies. In addition, process trees are tailored towards process discovery, and have been used previously to discover block-structured workflow nets [14]. A process tree describes a language; an operator describes how the languages of its subtrees are to be combined.

Definition 2 (Process Tree). We formally define *process trees* recursively. We assume a finite alphabet $\mathbb{A}$ of activities and a set $\bigotimes$ of operators to be given. Symbol $\tau \notin \mathbb{A}$ denotes the silent activity.

- a with $a \in (\mathbb{A} \cup \{ \tau \})$ is a process tree;
- Let $P_1, \ldots, P_n$ with $n > 0$ be process trees and let $\otimes \in \bigotimes$ be a process tree operator, then $\otimes(P_1, \ldots, P_n)$ is a process tree. We consider the following operators for process trees:
  - $\rightarrow$ denotes the *sequential execution* of all subtrees;
  - $\times$ denotes the *exclusive choice* between one of the subtrees;
  - $\circlearrowleft$ denotes the *structured loop* of loop body P_1 and alternative loop back paths $P_2, \ldots, P_n$ (with $n \geq 2$);
  - $\wedge$ denotes the *parallel (interleaved) execution* of all subtrees.

┘

Definition 3 (Process Tree Semantics). To describe the semantics of process trees, the language of a process tree P is defined using a recursive monotonic function $\mathcal{L}(P)$, where each operator $\otimes \in \bigotimes$ has a language join function $\otimes^l : (\mathcal{P}(\mathbb{A}^*) \times \ldots \times \mathcal{P}(\mathbb{A}^*)) \mapsto \mathcal{P}(\mathbb{A}^*)$:

$$\mathcal{L}(a) = \{ \langle a \rangle \} \textbf{ for } a \in \mathbb{A} \qquad \mathcal{L}(\otimes(P_1, \ldots, P_n))$$
$$\mathcal{L}(\tau) = \{ \varepsilon \} \qquad = \otimes^l(\mathcal{L}(P_1), \ldots, \mathcal{L}(P_n))$$

Each operator has its own language join function $\otimes^l$. The language join functions below are borrowed from [14,17], with $L_i \subseteq \mathbb{A}^*$:

$$\rightarrow^l(L_1, \ldots, L_n) = \{ t_1 \cdot \ldots \cdot t_n \mid \forall 1 \leq i \leq n : t_i \in L_i \}$$
$$\times^l(L_1, \ldots, L_n) = \textstyle\bigcup_{1 \leq i \leq n} L_i$$
$$\circlearrowleft^l(L_1, \ldots, L_n) = \{ t_1 \cdot t'_1 \cdot t_2 \cdot t'_2 \cdot \ldots \cdot t_{m-1} \cdot t'_{m-1} \cdot t_m \mid \forall i : t_i \in L_1, t'_i \in \textstyle\bigcup_{2 \leq j \leq n} L_j \}$$
$$\wedge^l(L_1, \ldots, L_n) = \{ t' \in (t_1 \diamond \ldots \diamond t_n) \mid \forall 1 \leq i \leq n : t_i \in L_i \}$$

┘

Example models and their languages:

$$\mathcal{L}(\wedge(a,b)) = \{\langle a,b\rangle, \langle b,a\rangle\} \qquad \mathcal{L}(\rightarrow(a,\times(b,c))) = \{\langle a,b\rangle, \langle a,c\rangle\}$$

$$\mathcal{L}(\circlearrowleft(a,b)) = \{\langle a\rangle, \langle a,b,a\rangle, \langle a,b,a,b,a\rangle, \dots\} \qquad \mathcal{L}(\wedge(a,\rightarrow(b,c))) = \{\langle a,b,c\rangle, \langle b,a,c\rangle, \langle b,c,a\rangle\}$$

3.4 Cancelation Process Trees

We extend the process tree representation to support cancelation behavior. We add two new tree operators to represent a cancelation region, and a new tree leaf to denote a cancelation trigger.

Definition 4 (Cancelation Process Tree). We formally define *cancelation process trees* recursively. We assume a finite alphabet $\mathbb{A}$ of activities to be given.

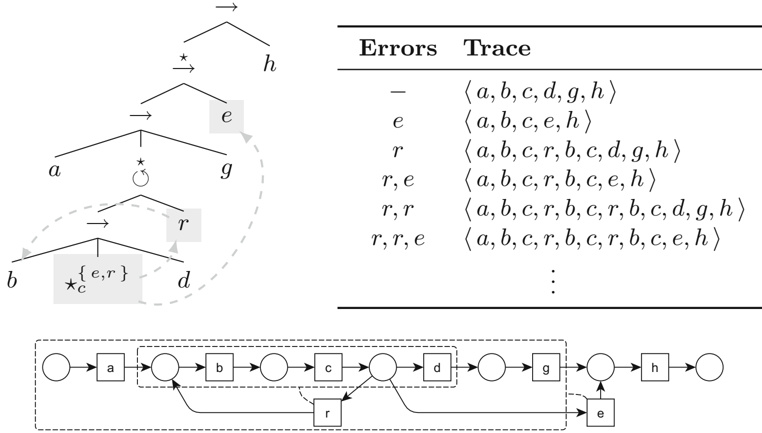
- Any process tree is also a cancelation process tree;
- Let $P_1, \dots, P_n$ with $n \geq 2$ be cancelation process trees, then:
  - $\overset{\star}{\rightarrow}(P_1, \dots, P_n)$ denotes the sequence-cancel of cancelation body P_1 and mutually exclusive error alternative paths $P_2, \dots, P_n$;
  - $\overset{\star}{\circlearrowleft}(P_1, \dots, P_n)$ denotes the loop-cancel of cancelation body P_1 and mutually exclusive error loop back paths $P_2, \dots, P_n$;
- $\star_a^E$ with $a \in \mathbb{A}, E \subseteq \mathbb{A}$ denotes the cancelation trigger. Combined with a cancelation operator $\overset{\star}{\rightarrow}, \overset{\star}{\circlearrowleft}$, this leaf denotes the point where we execute activity a, and have the option to trigger an error $e \in E$, firing a corresponding cancelation region. This concept is explained and defined in detail below. ⌟

The intuition behind the cancelation operators is described below. We reference to Table 2 for a concrete example tree with a step by step construction of its language. Observe that the new operators enable the modeling of semi-block structured behavior (see the semi-structured loop modeling activity r).

Assume a tree $\overset{\star}{\otimes}(P_1, \dots, P_n)$, $\overset{\star}{\otimes} \in \{\overset{\star}{\rightarrow}, \overset{\star}{\circlearrowleft}\}$ with a leaf $\star_a^E$ somewhere in the subtree P_1. When we want to "execute" this tree (i.e., generate a trace in its language), we start with the subtree P_1. At any $\star_a^E$ point, we do activity a as normal (happy flow), and have the option to trigger any error $e \in E$. For example, in Table 2, the leaf $\star_c^{\{e,r\}}$ can trigger either error e or r.

In case an error $e \in E$ is triggered, we need to find a matching cancelation region. A cancelation region $\overset{\star}{\otimes}(P_1, \dots, P_n)$ matches an error $e \in E$ iff e is the start activity for a trace in $t \in \mathcal{L}(P_2, \dots, P_n)$. When we trigger the error $e \in E$ at $\star_a^E$, we perform the activity a, but ignore the rest of the subtree P_1. I.e., we take the prefix up to and including a, and fire the cancelation region. We follow

Table 2. Example *cancelation process tree* (left) with its language (right) and corresponding Reset WF net (bottom). Shown are the traces in the language and the corresponding errors that are triggered to generate the trace. The grey arrows in the cancelation process tree indicate the possible error trigger “jumps”.



Errors	**Trace**
–	$\langle a, b, c, d, g, h \rangle$
e	$\langle a, b, c, e, h \rangle$
r	$\langle a, b, c, r, b, c, d, g, h \rangle$
r, e	$\langle a, b, c, r, b, c, e, h \rangle$
r, r	$\langle a, b, c, r, b, c, r, b, c, d, g, h \rangle$
r, r, e	$\langle a, b, c, r, b, c, r, b, c, e, h \rangle$
	$\vdots$

with a trace $t \in \mathcal{L}(P_2, \ldots, P_n)$ such that e is the start activity of t. I.e., we make a “jump” and execute a matching trace from one of the non-first subtrees. In Table 2, the node $\overset{\star}{\rightarrow}$ matches with error e, and the node $\overset{\star}{\circlearrowleft}$ matches with error r.

The difference between $\overset{\star}{\rightarrow}$ and $\overset{\star}{\circlearrowleft}$ is as follows: In case of $\overset{\star}{\rightarrow}$, we have sequential behavior, i.e., after a happy flow or error path, we continue with the rest of the process tree. In case of $\overset{\star}{\circlearrowleft}$, we have looping behavior, i.e., after an error path, we loop back and try executing P_1 again. For instance, in the example of Table 2 at the leaf $\star_c^{\{e,r\}}$, we can either continue as normal (happy flow), or jump to r (repetitively) or e (only once).

Definition 5 (Cancelation Process Tree Semantics). We define the semantics of cancelation process trees in multiple steps, and provide an adaptation for the existing process tree semantics.

First, we define the language of the cancelation trigger leaf $\star_a^E$. At this leaf, we can either execute activity a as normal, or execute it and trigger an error $e \in E$.

$$\mathcal{L}(\star_a^E) = \{ \left\langle \star_a^E \right\rangle, \langle a \rangle \} \textbf{ for } a \in \mathbb{A}\,,\ E \subseteq \mathbb{A}$$

Next, we define the language for the cancelation operators $\overset{\star}{\rightarrow}, \overset{\star}{\circlearrowleft}$. There are two common cases for these operators: (1) no error is triggered, and (2) an error is not caught by this operator. We define $\overset{\star}{\otimes}^{l}$ to represent these common cases.

$$\overset{\star}{\otimes}^{l}(L_1,\ldots,L_n) = \{\, t_1 \mid t_1 \in L_1\,,\ end(t_1) \neq \star_a^E \,\} \cup \{\, t_1 \cdot \left\langle \star_a^{E\setminus S} \right\rangle \mid t_1 \cdot \left\langle \star_a^E \right\rangle \in L_1, \\ S = \{\, head(t) \mid t \in \textstyle\bigcup_{2\leq j\leq n} L_j \,\}\,,\ E \setminus S \neq \emptyset \,\}$$

For the sequence-cancel operator $\overset{\star}{\rightarrow}$, we extend upon the language of $\overset{\star}{\otimes}^{l}$ by allowing a matching error path, after which we continue with the rest of the process tree.

$$\overset{\star}{\rightarrow}^{l}(L_1,\ldots,L_n) = \{\, t_1 \cdot \langle\, a \,\rangle \cdot t_e \mid t_1 \cdot \left\langle \star_a^E \right\rangle \in L_1\,,\ head(t_e) \in E\,,\ t_e \in \textstyle\bigcup_{2\leq j\leq n} L_j \,\} \\ \cup \overset{\star}{\otimes}^{l}(L_1,\ldots,L_n)$$

For the loop-cancel operator $\overset{\star}{\circlearrowleft}$, we extend upon the language of $\overset{\star}{\otimes}^{l}$ by allowing a matching error path, after which we loop back and try executing P_1 again.

$$\overset{\star}{\circlearrowleft}^{l}(L_1,\ldots,L_n) = \{\, t_1 \cdot \langle\, a_1 \,\rangle \cdot t_1' \cdot t_2 \cdot \langle\, a_2 \,\rangle \cdot t_2' \cdot \ldots \cdot t_{m-1} \cdot \langle\, a_{m-1} \,\rangle \cdot t_{m-1}' \cdot t_m \\ \mid t_m \in \overset{\star}{\otimes}^{l}(L_1,\ldots,L_n)\,,\ \forall i < m : t_i \cdot \left\langle \star_{a_i}^{E_i} \right\rangle \in L_1\,,\ head(t_i') \in E_i, \\ t_i' \in \textstyle\bigcup_{2\leq j\leq n} L_j \} \cup \overset{\star}{\otimes}^{l}(L_1,\ldots,L_n)$$

The existing process tree semantics can easily be adapted for these cancelation semantics by applying a prefix function ϕ to all traces in $\otimes^l$ (for $\otimes^l \in \left\{ \rightarrow^l, \times^l, \circlearrowleft^l, \wedge^l \right\}$). The idea is to remove any activity after a $\star_a^E$ symbol. For instance $\phi(\langle\, a, b, c \,\rangle) = \langle\, a, b, c \,\rangle$, but $\phi(\left\langle\, a, \star_b^E, c \,\right\rangle) = \left\langle\, a, \star_b^E \,\right\rangle$. ⌟

Below are some simple models and their corresponding language:

$$\mathcal{L}(\rightarrow(a, \star_b^{\{\,e\,\}}, c)) = \{\, \langle\, a, b, c \,\rangle, \langle\, a, \star_b^{\{\,e\,\}} \,\rangle \,\}$$

$$\mathcal{L}(\overset{\star}{\rightarrow}(\rightarrow(a, \star_b^{\{\,e\,\}}, c), e) = \{\, \langle\, a, b, c \,\rangle, \langle\, a, b, e \,\rangle \,\}$$

$$\mathcal{L}(\overset{\star}{\circlearrowleft}(\rightarrow(a, \star_b^{\{\,r\,\}}, c), r) \\ = \{\, \langle\, a, b, c \,\rangle, \langle\, a, b, r, a, b, c \,\rangle, \\ \langle\, a, b, r, a, b, r, a, b, c \,\rangle, \ldots \}$$

4 Heuristics for Error Oracle

We rely on explicitly modeling cancelation triggers and error activities (see Definition 4). For the algorithm in Sect. 5, we assume that the error activities are also explicit in the input. However, for any given event log, this is usually not the case (see Definition 1). To make error activities explicit in the input, we will assume a so-called error oracle function as an additional input.

Definition 6 (Error Oracle). Let $\mathbb{A}$ be a set of activities. Let $\mathit{isError} : \mathbb{A} \mapsto \{\, \mathit{true}, \mathit{false} \,\}$ be an error oracle function, yielding *true* iff an activity $a \in \mathbb{A}$ is an error activity. ⌟

There are numerous ways to instantiate such an error oracle function. A simple heuristics is to rely on domain knowledge or keywords in the activity names. For example, a negative activity name like "Cancelled" or "Declined" is often a good candidate. In addition, one can also check activities that break the normal flow: timed triggers, (external) events or asynchronous activities. Alternatively, exception or error data attributes may prove useful.

When none of these heuristics are an option, one can always fall back to an optimization strategy. The intuition is that, if cancelation behavior is present, modeling this behavior with cancelation operators will yield a more fitting and possibly more precise process tree. This is easy to see considering the fact that process trees traditionally capture only block-structured behavior, and cancelation behavior breaks this block-structuredness. Such an optimization strategy would feed several candidate error oracle functions to the discovery algorithm, compute the fitness and precision of the resulting model, and return the best scoring candidate.

Future work should look into more behavioral oriented error oracle heuristics.

5 Model Discovery

In this section, we will detail our discovery approach. We start by introducing the directly follows abstraction over an event log in Subsect. 5.1. Next, we briefly cover the framework our technique is based on in Subsect. 5.2. After that, we detail our proposed approach in Subsect. 5.3.

5.1 Directly Follows Graph and Cuts

The *directly follows relation* describes when two activities directly follow each other in a process. This relation can be expressed in the *directly follows graph* of a log L, written $G(L)$. Nodes in $G(L)$ are the activities of L. An edge (a, b) is present in $G(L)$ iff some trace $\langle \dots, a, b, \dots \rangle \in L$. We define the start and end nodes of G, $\mathit{Start}(G)$ and $\mathit{End}(G)$ respectively, based on the start and end activities in L. An *n-ary cut* of $G(L)$ is a partition of the nodes of the graph into disjoint sets $\Sigma_1, \dots, \Sigma_n$.

Consider a directly follows graph $G(L)$ and error oracle function $\mathit{isError} : \mathbb{A} \mapsto \{\, \mathit{true}, \mathit{false} \,\}$. If an edge (a, b) has $\mathit{isError}(b)$, then we call (a, b) an *error edge*. In any subgraph $G' \subseteq G$, a node a can only be an end node of G' iff it would be an end node without error edges.

5.2 Discovery Framework

Our technique is based on the Inductive Miner (IM) framework for discovering process tree models, as described in [14]. Given a set $\bigotimes$ of process tree operators, [14] defines a framework to discover models using a divide and conquer approach. Given a log L, the framework searches for possible splits of L into sublogs $L_1, \ldots, L_n$, such that these logs combined with an operator $\otimes \in \bigotimes$ can (at least) reproduce L again. The split search is based on finding cuts in the directly follows graph $G(L)$ of the log L. For each operator, a different cut is characterized based on the edges between the nodes in $G(L)$. The framework then recurses on the corresponding sublogs and returns the discovered submodels. Logs with empty traces or traces with a single activity form the base cases for this framework. Note that, by design, each activity only appears once in the produced process tree, and this tree can be a generalization of the original event log.

We use this framework as a basis because of its extensibility (it works independently of the chosen process tree operators), as well as the following properties: the log fits the resulting model, and there exists an implementation with a polynomial run time complexity [14].

5.3 Cancelation Discovery

We generalize the above approach to also support the discovery of cancelation behavior by including the error oracle function *isError* as an additional input, and tracking *cancelation triggers* during discovery. Note that we maintain the rediscoverability and fitness guarantees of the original framework [14]. In Algorithm 1, an overview of the discovery approach is given. In Table 3, an example run is given.

We will first discuss *cancelation triggers* in more detail. The three extension points labelled in Algorithm 1 are discussed next: (1) the cancelation trigger base case (line 5), (2) the cut finding extensions (line 10), and (3) the log splitting for cancelation (line 11).

Cancelation Triggers. A key observation is that we can track cancelation triggers during discovery. Whenever we observe an error activity e with $\mathit{isError}(e)$ in the log, it had to be triggered by the last activity before e. This naturally follows from the prefix cancelation semantics and the fact that each activity only appears once in the produced process tree. We keep track of these predecessors via the *triggers* mapping defined below.

Definition 7 (Cancelation Triggers). Let $\mathit{triggers} : \mathbb{A} \mapsto \mathcal{P}(\mathbb{A})$ be a cancelation triggers mapping, mapping an activity to the set of error activities such that $\forall E \in \mathit{rng}(\mathit{triggers}), e \in E : \mathit{isError}(e)$. ⌟

Algorithm 1: Cancelation Discovery

Input: An event log L and error oracle function $isError : \mathbb{A} \mapsto \{ true, false \}$
Output: A cancelation process tree P such that L fits P
Description: Extended framework that takes into account cancelation operators.

```
DISCOVER(L, isError)
(1)   if ∀σ ∈ L : σ = ε
(2)       // the log is empty or only contains empty traces
(3)       return τ
(4)   else if ∃x ∈ 𝔸 : ∀σ ∈ L : σ = ⟨x⟩
(5)       // the log only has a single activity - - - - - - - - - - - - - - - - - - - - - (1)
(6)       if triggers(x) ≠ ∅ then return ⋆_x^{triggers(x)}   // cancelation trigger case
(7)                        else return x                      // normal base case
(8)   else
(9)       // the normal framework cases
(10)      (⊗, (Σ_1, ..., Σ_n)) = findCut′(G(L), isError) - - - - - - - - - - - - - - (2)
(11)      ((L_1, ..., L_n), triggers) = splitLog′(L, (Σ_1, ..., Σ_n), isError, triggers) - (3)
(12)      return ⊗(M_1, ..., M_n)  where  M_i = DISCOVER(L_i, isError)
```

(1) Base Case. In the case the sublog consists of only a single activity, we have two options. To discover a cancelation trigger $\star_a^E$ for an activity a, we simply check the *triggers* mapping. If this is mapping empty, we have a normal activity leaf a, else we have a cancelation trigger leaf with $E = triggers(a)$.

(2) Finding Cuts. We include support for our cancelation tree operators by adding new cuts, and only slightly adapting existing cut definitions from [14]. In Fig. 3, all the graph cuts are depicted informally.

In our cancelation discovery, any non-cancelation cut *cannot* have an error edge between two partitions. In contrast, a cancelation cut is characterized by having error edges from its first partition to all non-first partitions. That is, in a cancelation cut, the first partition is the normal (happy flow) behavior inside the cancelation region. The non-first partitions are the mutually exclusive error paths after triggering the cancelation. The sequence and loop cancelation cuts are formally defined below.

Definition 8 (Sequence Cancel Cut). A sequence cancel ($\xrightarrow{\star}$) cut is a partially ordered cut $\Sigma_1, \ldots, \Sigma_n$, with $n \geq 2$, of a directly-follows graph G such that:

1. All start activities are in the body Σ_1:
$$Start(G) \subseteq \Sigma_1$$
2. Every partition Σ_i has some end activities:
$$\forall i \geq 1 : End(G) \cap \Sigma_i \neq \emptyset$$
3. There are only error edges from Σ_1 to $\Sigma_{i>1}$:
$$\forall i > 1, a_i \in \Sigma_i, a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow isError(a_i)$$
4. There are no edges from $\Sigma_{i>1}$ to $\Sigma_{j\geq 1}$:
$$\forall i > 1, j \geq 1, i \neq j, a_i \in \Sigma_i, a_j \in \Sigma_j : (a_i, a_j) \notin G$$

⌟

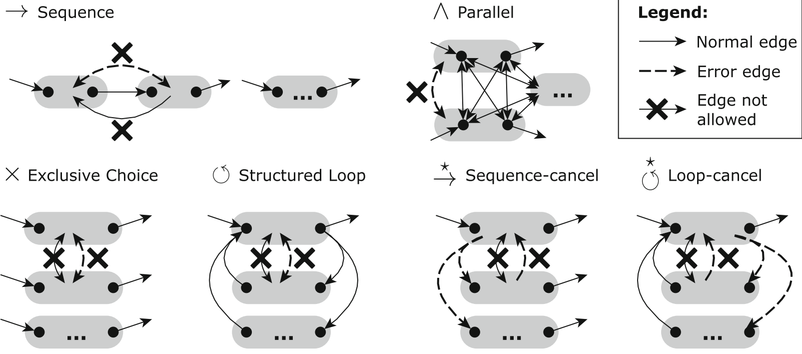


Fig. 3. Cuts of the directly-follows graph for all operators. The grey areas indicate partitions; the arrows indicate required and disallowed edges characterizing the cut.

Definition 9 (Loop Cancel Cut). A loop cancel ($\overset{\star}{\circlearrowleft}$) cut is a partially ordered cut $\Sigma_1, \ldots, \Sigma_n$, with $n \geq 2$, of a directly-follows graph G such that:

1. All start and end activities are in the body Σ_1:
$$Start(G) \cup End(G) \subseteq \Sigma_1$$
2. There are only error edges from Σ_1 to $\Sigma_{i>1}$:
$$\forall i > 1, a_i \in \Sigma_i, a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow isError(a_i)$$
3. There are only edges from Σ_i to start nodes in Σ_1:
$$\forall i > 1, a_i \in \Sigma_i, a_1 \in \Sigma_1 : (a_i, a_1) \in G \Rightarrow a_1 \in Start(G)$$
4. There are no edges from $\Sigma_{i>1}$ to $\Sigma_{j>1}$:
$\forall i > 1, j > 1, i \neq j, a_i \in \Sigma_i, a_j \in \Sigma_j : (a_i, a_j) \notin G$
5. If Σ_i has an edge to Σ_1, it connects to all start activities:
$$\forall i > 1, a_i \in \Sigma_i, a_1 \in Start(G) : (\exists a_1' \in \Sigma_1 : (a_i, a_1') \in G) \Leftrightarrow (a_i, a_1) \in G$$
⌟

(3) Splitting Logs. Once a cut $\Sigma_1, \ldots, \Sigma_n$ has been found for an operator $\otimes$, we need to split the log L into sublogs $L_1, \ldots, L_n$, such that these logs combined with operator $\otimes$ can (at least) reproduce L again. For the new cancelation operators, we define the log splits and cancelation trigger update below.

Definition 10 (Sequence Cancel Split). Given a sequence cancelation cut $\Sigma_1, \ldots, \Sigma_n$:

1. Sublog L_1 consists of all maximal prefix subtraces with activities in Σ_1:
$$L_1 = \{\, t_1 \mid t_1 \cdot t_2 \in L, \Sigma(t_1) \subseteq \Sigma_1, (t_2 = \varepsilon \vee (t_2 = \langle\, e, \ldots \rangle \wedge e \notin \Sigma_1))\,\}$$
2. Sublog $L_{i>1}$ consists of all maximal postfix subtraces with activities in Σ_i:
$$L_{i>1} = \{\, t_2 \mid t_1 \cdot t_2 \in L, \Sigma(t_2) \subseteq \Sigma_i, (t_1 = \varepsilon \vee (t_1 = \langle \ldots, a_1 \rangle \wedge a_1 \in \Sigma_1))\,\}$$
3. Update the triggers mapping such that any activity $a \in \mathbb{A}$ ending a trace in L_1 is mapped to all error activities following it in L:
$$triggers(a) = triggers(a) \cup \{\, e \mid t_1 \cdot t_2 \in L, \Sigma(t_1) \subseteq \Sigma_1, e \notin \Sigma_1, t_1 = \langle \ldots, a \rangle, t_2 = \langle\, e, \ldots \rangle \,\}$$
⌟

For example, consider a log $L = [\langle b, c, d \rangle, \langle c, b, d \rangle, \langle c, e, f \rangle]$ (taken from step 2 in Table 3) and sequence cancelation cut $\Sigma_1 = \{ b, c, d \}$, $\Sigma_2 = \{ e, f \}$. The resulting log splits are $L_1 = [\langle b, c, d \rangle, \langle c, b, d \rangle, \langle c \rangle]$, $L_2 = [\langle e, f \rangle]$, and the resulting triggers mapping is $\mathit{triggers} = \{ c \mapsto \{ e \} \}$.

Definition 11 (Loop Cancel Split). Given a loop cancelation cut $\Sigma_1, \ldots, \Sigma_n$:

1. Sublog L_i consists of all maximal subtraces with activities in Σ_i:

$$L_i = \{ t_2 \mid t_1 \cdot t_2 \cdot t_3 \in L, \Sigma(t_2) \subseteq \Sigma_i, (t_1 = \varepsilon \vee (t_1 = \langle \ldots, a_1 \rangle \wedge a_1 \notin \Sigma_i)), (t_3 = \varepsilon \vee (t_3 = \langle a_3, \ldots \rangle \wedge a_3 \notin \Sigma_i)) \}$$

2. Update the triggers mapping such that any activity $a \in \mathbb{A}$ ending a trace in L_1 is mapped to all error activities following it in L:

$$\mathit{triggers}(a) = \mathit{triggers}(a) \cup \{ e \mid t_1 \cdot t_2 \cdot t_3 \in L, \Sigma(t_2) \subseteq \Sigma_1, e \notin \Sigma_1, t_2 = \langle \ldots, a \rangle, t_3 = \langle e, \ldots \rangle \}$$

⌟

For example, consider a log $L = [\langle b, c, r, b, c, d \rangle]$ (a small snippet from Table 2) and loop cancel cuts $\Sigma_1 = \{ b, c, d \}$, $\Sigma_2 = \{ r \}$. The resulting log splits are $L_1 = [\langle b, c \rangle, \langle b, c, d \rangle]$, $L_2 = [\langle r \rangle]$, and the resulting triggers mapping is $\mathit{triggers} = \{ c \mapsto \{ r \} \}$.

6 Evaluation

In this section, we compare our technique against related, implemented techniques. The proposed algorithm is implemented in the *Statechart* plugin for the process mining framework ProM [12]. In the remainder of this section, we will refer to Algorithm 1 as *cancelation*. We end the evaluation by showing example results obtained using our tool.

6.1 Input and Methodology for Comparative Evaluation

In this comparative evaluation, we focus on the quantitative aspects. That is, the models discovered are precise and fit the actual system. We compare a number of techniques and input event logs on: (1) the *running time* of the technique, (2) the *model quality* (fitness and precision), and (3) the *model simplicity*.

For the running time, we measure the average running time and associated 95% confidence interval over 30 micro-benchmark executions, after 10 warmup rounds for the Java JVM. Each technique is allowed at most 30 seconds for completing a single model discovery. Fitness and precision are calculated using the technique described in [1]. In short, *fitness* expresses the part of the log that is represented by the model; *precision* expresses the behavior in the model that

Table 3. Example Cancelation Discovery on the log $[\langle a, b, c, d, g\rangle, \langle a, c, b, d, g\rangle, \langle a, c, e, f, g\rangle]$ and error oracle *isError* with *isError*(e) = *true* and *false* otherwise. The rows illustrate how the discovery progresses. The highlights indicate the parts of the log and directly follows graph used, and relate them to the corresponding partial model that is discovered. The dashed arrow is an error edge, and the dashed lines indicate the cuts. The resulting Reset WF net is shown at the bottom.

Step	Discovered Model	Event Log	Directly Follows Graph
1	$\rightarrow$ a ? g	a b c d g a c b d g a c e f g	a b c d g e f
2	$\rightarrow$ a $\overset{\star}{\rightarrow}$ g ? ?	a b c d g a c b d g a c e f g	a b c d g e f
3	$\rightarrow$ a $\overset{\star}{\rightarrow}$ g ? $\rightarrow$ e f	a b c d g a c b d g a c e f g	a b c d g e f
4/5	$\rightarrow$ a $\overset{\star}{\rightarrow}$ g $\rightarrow$ $\rightarrow$ $\wedge$ d e f b $\star_c^{\{e\}}$	a b c d g a c b d g a c e f g	a b c d g e f

a b c d g e f

is present in the log. For these experiments we used a laptop with an i7-4700MQ CPU @ 2.40 GHz, Windows 8.1 and Java SE 1.7.0 (64 bit) with 8 GB of RAM.

We selected several real-life event logs as experiment input, covering a range of input problem sizes and complexities. The input problem size is typically measured in terms of four metrics: number of traces, number of events, number of activities (size of the alphabet), and average trace length. The event logs and their sizes are shown in Table 4.

Table 4. The event logs used in the evaluation, with input sizes and applied filters

Event Log	# Traces	# Events	# Acts	Avg. |T|	Filter
[13] NASA CEV	2	48	17	24.00	Only test cases 1 and 10
[3] WABO	1,434	8,577	27	5.98	–
[22] BPIC12, A	13,087	60,849	10	4.64	"A_" subprocess only
[8] Road fine, a	150,370	561,470	11	3.73	–
[8] Road fine, f	150,370	404,009	9	2.68	No "no payment" and "add penalty"

Table 5. The error oracles used for the event logs in the evaluation.

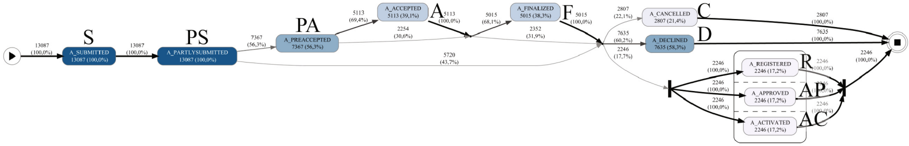
Event Log	Error Oracle Activities
[13] NASA CEV	"cev.ErrorLog.last()"
[3] WABO	"T15 Print document X request unlicensed", "T16 Report reasons to hold request"
[22] BPIC12, A	"A_CANCELLED", "A_DECLINED"
[8] Road fine, a	"Send for Credit Collection"
[8] Road fine, f	"Send for Credit Collection"

The *NASA CEV* [13] event log describes two executions of a software process with errors. This small log was obtained from two existing NASA CEV software tests. The *WABO* [3] event log describes the receipt phase of an environmental permit application process ('WABO') at a Dutch municipality. The *BPIC12* [22] event log is a BPI challenge log that describes three subprocesses of a loan application process. In this evaluation, we only focus on the "A_" subprocess. The *Road fine* [8] event log was obtained from an information system managing road traffic fines. We use two variants of this large event log. The *Road fine, a* variant is the largest, most complex event log in our experimental setup. In variant *Road fine, f*, we filtered out two asynchronous activities to decrease the (directly-follows) complexity.

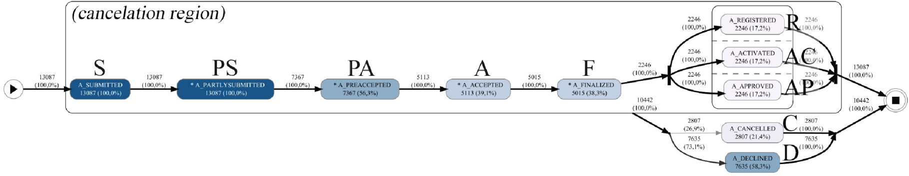
We compare our discovery algorithm against most of the techniques mentioned in Sect. 2. Unfortunately, we could not compare against the work of [5–7,9,10,15,16] due to invalid input assumptions, absence of semantics, or the lack of a reference implementation. The Inductive Miner (IM) [14] is our baseline comparison algorithm, since our approach builds upon the IM framework. For the Inductive Miner and our derived techniques, we also consider the *paths* setting. This is the frequency cutoff for discovering an 80/20 model: 1.0 means all behavior, 0.8 means 80% of the behavior. In Table 5, we have listed the error oracles we used for our cancelation discovery techniques.

6.2 Comparative Evaluation Results and Discussion

Runtime Analysis. In Table 6, the results for the runtime benchmark are given.



(a) IM (baseline) model result, no cancelation. Skips obfuscate the happy flow.



(b) Cancel model result, with cancelation region. Happy flow in cancel reagion.

Fig. 4. Models mined from the BPIC12 log with the Path filter at 0.8, both produced by our ProM plugin [12], and visualized in the Statechart language. Legend: (S) A_Submitted, (PS) A_PartlySubmitted, (PA) A_PreAccepted, (A) A_Accepted, (F) A_Finalized, (C) A_Cancelled, (D) A_Declined, (R) A_Registered, (AP) A_Approved, (AC) A_Activated

The first thing we notice is that, in contrast to the TS Cancel technique, our Cancelation algorithm always discovers a model within the allotted time. When compared to the baseline Inductive Miner, there seems to be a small overhead in running time. There are two explanations for this small overhead. One being the fact that more tree operator cuts have to be checked at each recursive call of the algorithm. But more importantly, the new cancelation operators potentially uncover more structures in the directly follows graph. In cases where the original Inductive Miner might give up and falls back to loops with skips and/or flower models, we can find a cancelation pattern, and recurse on a more structured subproblem. The end result is that we have more recursive calls to uncover all the structures/tree operators, and hence have a larger running time. Nevertheless, our technique successfully scales to larger logs and consistently yields results within seconds.

Model Quality Analysis. In Table 7, the results of the model quality measurements are given. Note that in order to compute model quality scores, the model should be sound.

Observe that, compared to the original Inductive Miner, our Cancelation algorithm always yields an equal or more fitting model. Moreover, we preserve the perfect fitness guarantee of the original Inductive Miner (for *path* 1.0). In addition, in most cases, the resulting model is also more precise.

Table 6. Runtime for the different algorithms, paths filter settings, and event logs.

	Algorithm	Path	NASA CEV	WABO	BPIC12, A	Road fine, a	Road fine, f
	[20] Alpha miner	-	0.3	8.6	18.7	467.8	260.6
	[24] Heuristics	-	2.3	40.5	162.8	1641.7	1047.7
	[21] ILP	-	371.0	560.3	501.5	3426.1	2517.8
	[23] LP, filtering	-	381.9	673.2	497.2	3395.1	2583.1
	[18] Genetic miner	-	3498.1	26033.8	3402.9	–[T]	25592.0
	[4] ETMd miner	-	27836.5	27516.7	27130.5	28722.1	27557.6
	[11] TS Cancel	-	–[T]	–[T]	139.6	–[T]	–[T]
	[19] TS Regions	-	18.9	–[T]	555.1	5089.5	3455.2
	[14] IM (baseline)	1.0	1.4	120.1	308.0	5668.9	2859.0
	[14] IM (baseline)	0.8	0.9	138.1	300.7	4049.5	2540.2
	[14] IM (baseline)	0.5	0.9	135.4	301.4	4132.9	2537.1
Ours	Cancelation	1.0	2.0	176.7	373.2	5972.5	3047.9
	Cancelation	0.8	1.5	156.4	379.8	6145.4	3289.3
	Cancelation	0.5	1.7	154.1	377.7	6180.8	3558.8
			10^0 10^2 10^4	10^2 10^4	10^1 10^2 10^3 10^4	10^3 10^4	10^3 10^4

Avg. runtime (in milliseconds, with log scale plot), over 30 runs, with 95% confidence interval
[T] Time limit exceeded (30 sec.)

Table 7. Fitness (Fit.) and Precision (Prec.) scores for the different algorithms, paths filter settings, and event logs. Scores range from 0.0 to 1.0, higher is better.

			NASE CEV		WABO		BPIC12, A		Road fine, a		Road fine, f	
	Algorithm	Path	Fit.	Prec.	Fit.	Prec.	Fit.	Prec.	Fit.	Prec.	Fit.	Prec.
	[20] Alpha miner	-	0.89	0.08	–[U]	–[U]	–[U]	–[U]	–[U]	–[U]	–[U]	–[U]
	[24] Heuristics	-	–[U]	–[U]	0.61	0.98	–[U]	–[U]	–[U]	–[U]	0.74	1.00
	[21] ILP	-	1.00	0.33	1.00	0.12	1.00	0.22	1.00	0.50	1.00	0.53
	[23] ILP, filtering	-	1.00	0.33	1.00	0.35	1.00	0.28	0.78	1.00	0.81	1.00
	[18] Genetic miner	-	–[U]	–[U]	–[U]	–[U]	–[U]	–[U]	–[N]	–[N]	–[U]	–[U]
	[4] ETMd miner	-	0.74	1.00	0.83	1.00	1.00	0.86	0.79	1.00	0.75	1.00
	[11] TS Cancel	-	–[N]	–[N]	–[N]	–[N]	0.91	0.78	–[N]	–[N]	–[N]	–[N]
	[19] TS Regions	-	0.27	0.61	–[N]	–[N]	0.93	0.88	0.86	0.76	0.76	0.82
	[14] IM (baseline)	1.0	1.00	0.69	1.00	0.43	1.00	0.89	1.00	0.69	1.00	0.83
	[14] IM (baseline)	0.8	0.74	0.73	0.94	0.64	1.00	0.92	0.99	0.48	1.00	0.82
	[14] IM (baseline)	0.5	0.62	0.75	0.94	0.63	0.82	1.00	0.76	0.48	0.74	0.77
Ours	Cancelation	1.0	1.00	0.70	1.00	0.62	1.00	1.00	1.00	0.66	1.00	0.76
	Cancelation	0.8	0.76	0.58	0.94	0.67	1.00	1.00	1.00	0.35	1.00	0.68
	Cancelation	0.5	0.64	0.67	0.94	0.66	1.00	1.00	0.90	0.39	0.81	0.73

[U] Unsound model [N] No model (see Table 6)

In all cases, we can see that we outperform the ILP algorithms on precision, and we outperform the ETMd miner and TS based miners on fitness. Overall, we can conclude that the added expressiveness of modeling the cancelation region have a positive impact on the model quality.

On the Simplicity of Models. We compared the discovered models both using a simplicity metric [4] and manually. In most cases, the discovered models are comparably complex, with the cancelation models usually being slightly simpler.

In Fig. 4, two discovered models for the BPIC12 log at paths 0.8 are shown. Note that in the Cancelation model (Fig. 4(b)), we see that the main, happy flow behavior is neatly discovered inside the cancelation region, and the "negative" behavior is modeled separately after triggering the cancelation region. In the IM model (Fig. 4(a)), skips obfuscate the normal happy flow behavior.

Overall, we can conclude that the added expressiveness of modeling the cancelation region has, in most cases, a positive impact on the model simplicity.

7 Conclusion

In this paper, we presented a novel extension to the process tree model to support cancelation behavior, and proposed a novel process discovery technique to discover sound, fitting models with cancelation features. The proposed discovery technique relies on a generic error oracle function, and allows us to discover complex combinations of multiple, possibly nested cancelation regions based on observed behavior. An implementation of the proposed algorithm has been tested and made available via the *Statechart* plugin for the ProM framework [12]. Our experimental results, based on real-life event logs, demonstrate the feasibility and usefulness of the approach.

Future work aims to further aid the user in selecting an error oracle, and (partially) automate the error oracle instantiation. In addition, we aim to support reliability analysis around cancelation features, using additional event log data. Lastly, enabling the proposed techniques in a streaming context could provide valuable real-time insights into (business) processes in their natural environment. Techniques able to operate in a streaming context need less memory and are therefore also valuable for other types of analysis.

References

1. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Eindhoven University of Technology (2014)
2. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Bruno, G.: Automated discovery of structured process models: discover structured vs. discover and structure. In: Comyn-Wattiau, I., Tanaka, K., Song, I.-Y., Yamamoto, S., Saeki, M. (eds.) ER 2016. LNCS, vol. 9974, pp. 313–329. Springer, Cham (2016). doi:10.1007/978-3-319-46397-1_25
3. Buijs, J.C.A.M.: Receipt phase of an environmental permit application process ('WABO'), CoSeLoG project (2014). https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6
4. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) OTM 2012. LNCS, vol. 7565, pp. 305–322. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33606-5_19

5. Carmona, J., Cortadella, J., Kishinevsky, M.: A region-based algorithm for discovering Petri nets from event logs. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 358–373. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85758-7_26
6. Celonis GmbH: Celonis Process Mining. https://www.celonis.com. Accessed 06 July 2017
7. Conforti, R., Dumas, M., García-Banuelos, L., La Rosa, M.: BPMN miner: automated discovery of BPMN process models with hierarchical structure. Inf. Syst. **56**, 284–303 (2016)
8. De Leoni, M., Mannhardt, F.: Road traffic fine management process (2015). https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5
9. Gradient ECM: Minit. https://www.minit.io/. Accessed 06 July 2017
10. Günther, C.W., Rozinat, A.: Disco: discover your processes. In: Lohmann, N., Moser, S. (eds.) Proceedings of the Demonstration Track of the 10th International Conference on Business Process Management (BPM 2012), vol. 940, pp. 40–44. CEUR Workshop Proceedings (2012)
11. Kalenkova, A.A., Lomazova, I.A.: Discovery of cancellation regions within process mining techniques. Fundamenta Informaticae **133**(2–3), 197–209 (2014)
12. Leemans, M.: Statechart plugin for ProM 6. https://svn.win.tue.nl/repos/prom/Packages/Statechart/. Accessed 24 May 2017
13. Leemans, M.: NASA Crew Exploration Vehicle (CEV) software event log (2017). http://doi.org/10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76
14. Leemans, S.J.J.: Robust process mining with guarantees. Ph.D. thesis, Eindhoven University of Technology, May 2017
15. Redlich, D., Molka, T., Gilani, W., Blair, G., Rashid, A.: Constructs competition miner: process control-flow discovery of BP-domain constructs. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 134–150. Springer, Cham (2014). doi:10.1007/978-3-319-10172-9_9
16. van der Aalst, W.M.P.: Discovery, verification and conformance of workflows with cancellation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 18–37. Springer, Heidelberg (2008). doi:10.1007/978-3-540-87405-8_2
17. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer, Heidelberg (2016)
18. van der Aalst, W.M.P., de Medeiros, A.K.A., Weijters, A.J.M.M.: Genetic process mining. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005). doi:10.1007/11494744_5
19. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Soft. Syst. Model. **9**(1), 87–111 (2010)
20. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004)
21. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 368–387. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68746-7_24
22. van Dongen, B.F.: BPI Challenge 2012 (2012). http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

23. van Zelst, S.J., van Dongen, B.F., van der Aalst, W.M.P.: Avoiding over-fitting in ILP-based process discovery. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 163–171. Springer, Cham (2015). doi:10.1007/978-3-319-23063-4_10
24. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible Heuristics Miner (FHM). In: 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pp. 310–317, April 2011