MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 3, 2024
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Discovering Hierarchical Process Models: An Approach Based on Events Partitioning

A. K. Begicheva[1], I. A. Lomazova[1], R. A. Nesterov[1]

DOI: 10.18255/1818-1015-2024-3-294-315

[1]National Research University Higher School of Economics, Moscow, Russia

MSC2020: 68Q85
Research article
Full text in English

Received June 25, 2024
Revised July 18, 2024
Accepted July 24, 2024

Process mining is a field of computer science that deals with the discovery and analysis of process models based on automatically generated event logs. Currently, many companies are using this technology to optimize and improve their business processes. However, a discovered process model may be too detailed, sophisticated, and difficult for experts to understand. In this paper, we consider a problem of discovering the hierarchical business process model from a low-level event log, i. e., the problem of the automatic synthesis of more readable and understandable process models based on the data stored in the event logs of information systems.

The discovery of better-structured and more readable process models is extensively studied in the framework of process mining research from different perspectives. In this paper, we present an algorithm for discovering hierarchical process models represented as two-level workflow Petri nets. The algorithm is based on predefined event partitioning so that this partitioning defines a sub-process corresponding to a high-level transition at the top level of a two-level net. In contrast to existing solutions, our algorithm does not impose restrictions on the process control flow and allows for concurrency and iterations.

Keywords: process mining; Petri nets; workflow nets; process discovery; hierarchical process model; event log

INFORMATION ABOUT THE AUTHORS

Begicheva, Antonina K. (corresponding author)	ORCID iD: 0000-0001-6657-1760. E-mail: abegicheva@hse.ru Lecturer, M. Sc.
Lomazova, Irina A.	ORCID iD: 0000-0002-9420-3751. E-mail: ilomazova@hse.ru Professor, Dr. Sc.
Nesterov, Roman A.	ORCID iD: 0000-0002-4162-9070. E-mail: rnesterov@hse.ru Associate professor, PhD

Funding: Basic Research Program at HSE University.

For citation: A. K. Begicheva, I. A. Lomazova, and R. A. Nesterov, “Discovering hierarchical process models: an approach based on events partitioning”, *Modeling and Analysis of Information Systems*, vol. 31, no. 3, pp. 294–315, 2024. DOI: 10.18255/1818-1015-2024-3-294-315.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 3, 2024
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Синтез иерархических моделей процессов: подход на основе разбиения событий на множества

А. К. Бегичева[1], И. А. Ломазова[1], Р. А. Нестеров[1]

DOI: 10.18255/1818-1015-2024-3-294-315

[1]Национальный исследовательский университет «Высшая школа экономики», Москва, Россия

УДК 004.942
Научная статья
Полный текст на английском языке

Получена 25 июня 2024 г.
После доработки 18 июля 2024 г.
Принята к публикации 24 июля 2024 г.

Process mining — это область компьютерных наук, которая занимается синтезом и анализом моделей процессов на основе автоматически генерируемых журналов событий. В настоящее время многие организации используют эту технологию для оптимизации и совершенствования бизнес-процессов. Однако синтезированная модель процесса может быть слишком подробной, сложной и трудной для понимания экспертами. В работе мы рассматриваем задачу синтеза иерархической модели бизнес-процесса из низкоуровневого журнала событий, то есть, задачу автоматического синтеза более удобочитаемых и понятных моделей процессов на основе данных, хранящихся в журналах событий информационных систем.

Построение более структурированных и удобочитаемых моделей процессов широко изучается в рамках исследований в области process mining с разных точек зрения. В этой статье мы представляем алгоритм синтеза иерархических моделей процессов, представленных в виде двухуровневых сетей потоков работ. Алгоритм основан на предопределенном разбиении событий на множества, которые определяют подпроцессы, соответствующие высокоуровневым переходам на верхнем уровне двухуровневой сети потоков работ. В отличие от существующих решений, представленный алгоритм не накладывает ограничений на поток управления процессом, а также допускает параллелизм и итерации.

Ключевые слова: синтез моделей процессов; сети Петри; сети потоков работ; иерархические модели процессов; журнал событий

ИНФОРМАЦИЯ ОБ АВТОРАХ

Бегичева, Антонина Константиновна (автор для корреспонденции)	ORCID iD: 0000-0001-6657-1760. E-mail: abegicheva@hse.ru Преподаватель, магистр
Ломазова, Ирина Александровна	ORCID iD: 0000-0002-9420-3751. E-mail: ilomazova@hse.ru Профессор, доктор физико-математических наук
Нестеров, Роман Александрович	ORCID iD: 0000-0002-4162-9070. E-mail: rnesterov@hse.ru Доцент, канд. физ.-мат. наук

Финансирование: Программа фундаментальных исследований Национального исследовательского университета «Высшая школа экономики».

Для цитирования: A. K. Begicheva, I. A. Lomazova, and R. A. Nesterov, “Discovering hierarchical process models: an approach based on events partitioning”, *Modeling and Analysis of Information Systems*, vol. 31, no. 3, pp. 294–315, 2024. DOI: 10.18255/1818-1015-2024-3-294-315.

Introduction

Over the past decade, companies whose processes are supported by various information systems have become convinced of the need to store as much potentially useful information about the process executions within a system as possible. This was facilitated by qualitative improvement in the areas related to the extraction of valuable information from the recorded data, which helps to adjust the operation of companies over time and thus save and increase their resources. Process mining is a field of computer science that provides a palette of tools to extract the logic of the system behavior as well as to model and optimize the processes that occur in a system. In particular, process mining methods allow one to find inconsistencies between the planned and actual behavior of a system and to track the occurrence of the inefficient or incorrect behavior.

Despite the fact that increasing attention is being paid to preserving the optimal amount of the necessary information about processes, the actual data on process executions is not always available in a convenient format and with the necessary degree of detail, since system logs are generated for a lot of different purposes.

Process discovery aims at extracting processes from event logs and constructing models of these processes. Most of the available process discovery methods produce a model with the same level of detail provided by the initial event log [1].

Therefore, a promising area of research is the problem of discovering a more readable process model from a detailed event log, while preserving the important information about the process execution for experts. Readability of process models can be achieved in various ways. The most commonly used methods are filtering rare behavior from the original event log, skipping "minor" events (the significance of an event is assessed according to the chosen methodology); and abstraction, when some events are considered indistinguishable from others. We will discuss existing methods in more detail in Section 1. In our study, we consider the latter approach, when more readable models are the result of model abstraction — they are more compact and have the optimal level of detail for the work of experts in comparison to the level of model detail that could be obtained by direct discovery methods. To preserve the important data, we are dealing not only with abstract (high-level) models, but also with hierarchical models storing the low-level information in the form of sub-processes.

In this paper, we propose an algorithm for discovering hierarchical process models from event logs. Processes are represented using workflow nets [2], a special subclass of Petri nets used for modeling the control flow of business processes. This study extends our previously achieved results [3] where we proposed an approach to discovering abstract models for processes without cycles. Here, we provide a more general solution by overcoming the prohibition of cyclic behavior.

Hierarchical models allow us to have a high-level view of the model by "folding" the behavior of an individual sub-process into a high-level transition with the ability to unfold it back. Thus, at the top level, there is a high-level model in which every individual transition corresponds to a sub-process built from low-level events. The history of detailed behavior of the process is recorded in a low-level event log. Regarding the number of levels in the hierarchy, we will only use two levels — high and low, but the algorithm can naturally be extended to any number of levels.

The paper is structured as follows. Section 1 presents the review of related research. Section 2 gives theoretical preliminaries and the definitions used in the text. In Section 3, we discuss the basics of the hierarchical process discovery algorithm. Section 4 presents the main discovery algorithm and the proof of its correctness in the light of the perfect fitness preservation. Section 5 reports the outcomes from the experimental evaluation. In Section 6, we conclude the paper and discuss the possible future work directions.

1. Related work

Research connected with our paper can be classified into approaches to abstracting event logs and process models and approaches to constructing hierarchical process models from event logs.

One of the recent surveys [4] gives a comprehensive review of approaches and methods that can be applied for low-level event abstraction. The authors divide the methods according to: the learning strategy (supervised or unsupervised), the structure of the process models (strictly sequential or with interleaving), the low-level events grouping approach (deterministic or probabilistic) and the nature of the processed data (discrete or continuous data).

For example, the method presented in [5] is a supervised method that aligns the model complexity with the needs of different stakeholders. Another example of a supervised approach to event abstraction was presented in [6]. This method takes a low-level event log and transforms it to an event log at the desired level of abstraction, using the following behavioral patterns: sequence, choice, parallel, interleaving and repetition of events. This technique allows one to obtain a reliable mapping from low-level events to activity patterns automatically and construct a high-level event log using these patterns. Detecting high-level events based on the patterns of behavior in an event log does not make it possible to refine the accuracy of abstraction, based on the general knowledge of the system, or provide it only partially. Patterns provide the ability to change the scale but not to participate in the selection of correct high-level events. This could only be useful for a superficial analysis. However, there is a risk of combining unrelated low-level events into a single high-level event only because they are executed sequentially, but not because they belong to the same logical component of a system.

Another supervised event abstraction method was discussed in [7]. The nature of this method is as follows. The authors annotate a low-level event with the correct high-level event using the domain knowledge from the actual process model by the special attribute in the event log. In addition, this paper assumes that multiple high-level events are executed in parallel. This allows us to interpret a sequence of identical values as a single instance of a high-level event.

Unsupervised techniques do not require additional information beyond the input log. For example, in [8], the authors specify a fully unsupervised framework for partially ordered event data that detects abstraction classes using event data based on its observed execution context. In [9], the authors offer a framework for evaluating unsupervised abstraction techniques and evaluate the state-of-the-art methods using 400 event logs. One of the conclusions drawn from these evaluations is that there is typically a trade-off between high precision and high comprehensibility in the resulting model. The less abstract the model is the higher its calculated precision will be.

An example of the multi-perspective approach that combines features of the unsupervised and supervised methodologies is provided in [10]. After automatic identification of event groups, this method allows users to select the groups that are relevant and can be used for low-level log abstraction.

A general approach to the representation of multi-level event logs and the corresponding multi-level hierarchical models was studied in [11]. The authors highlighted the fact that this approach can combine multiple modeling notation for representing different levels in multi-level process models.

There are many ways of abstracting process models by reducing their size in order to make them more convenient to work with. Each method may be useful depending on a group of interrelated factors: the abstraction purposes, the presence of certain patterns and constructs, and the specifics of modeling notation. Reducing the size of the model by abstraction can be done as the “convolution” of groups of elements, or implemented by throwing some parts of the model away (insignificant in a particular case) [12]. The importance of the low-level event log abstraction is emphasized, among others, in [13].

Researchers determine which level of abstraction is appropriate for a particular case in different ways, but the main criterion is that the model should be readable and understandable. In [14], the abstraction of a process model occurs through “simplification” automatically: the user determines only the desired degree of detail, but not the actual correctness of identifying high-level events. Conversely, the paper [5] stressed the importance of the abstraction level dependence on the domain expert knowledge.

Petri nets [15] can also be extended by adding the hierarchy as, e.g., in Colored Petri nets (CPN) [16]. Hierarchical events allow one to construct more compact, readable and understandable process models. The hierarchy of CPN models can be used as an abstraction, in the case of two levels: a high-level *abstract* model and a low-level *refined* model. In our paper, the high-level model is a model with abstract transitions. An abstract transition refers to a Petri net sub-process which refines the activity represented by this high-level transition. A complete low-level, also referred to as *classical*, process model can be obtained from a high-level model by substituting sub-processes for high-level transitions. By the classical process model, we mean a model that is not loaded with information about the hierarchy, which has the same level of detail as the original event log.

Synthesis of a classical process model is a standard process discovery problem that has been extensively studied in the literature. A wide range of process discovery algorithms supports the automated classical process model synthesis [1].

Inductive Miner [17] is one of the most widely used process discovery algorithms that produces well-structured process models, built recursively from building blocks for standard behavioral patterns. They can be potentially used for constructing high-level process models. However, this technique does not take the actual correspondence between low-level events and sub-processes. In [18], the authors also used the recognition of behavioral patterns in a process by a structural partitioning algorithm and then defined a specific workflow schema for each pattern.

In [19], a two-phase approach to mining hierarchical process models was presented. Process models were considered as interactive and context-dependent maps based on common execution patterns. In the first phase, an event log is abstracted to the desired level by detecting relevant execution patterns. An example of such a pattern is the maximal repeat that captures typical sequences of activities in the log. Every pattern is then estimated by its frequency, significance, or some other metric needed for accurate abstraction. In the second phase, the *Fuzzy Miner* discovery algorithm [14], adapted to process map discovery, is applied to the transformed log.

FlexHMiner [20] is a general algorithm based on process trees implemented in ProM software. The authors stress the flexibility of this approach: to identify the hierarchy of events, the method supports both supervised methods and methods using the general knowledge of a process. The limitations of this method include the fact that each of the sub-processes can be executed only once, which means that the method is not suitable for processes with cycles.

A large volume of literature is devoted to the problem of discovering structured models from event logs. Researchers offer different techniques to improve the structure of discovered models, e. g., in [21], and to produce already well-structured process models [22, 23]. Different ways of detecting sub-processes in event logs, using low-level transition systems, were discussed in [24–26]. However, these works did not consider mining hierarchical process models from event logs.

One way to use process discovery techniques for abstract model synthesis is log pre-processing. For example in [27] the authors divide the initial log into sub-processes using activity instances information. The limitation of the proposed method is in the activity instance partitioning: as the authors only consider cases where a subprocess always begins and ends with fixed events.

In [3], the authors presented an algorithm for the discovery of a high-level process model from the event log for acyclic processes. This method takes the initial data on abstraction in the form of a set of detailed events grouped into high-level ones, which means that any method of identifying abstract events can potentially be used, including those based on expert knowledge. After pre-processing, this algorithm allows the use of any existing process discovery approach that is suitable for the synthesis of a classical process model. The possibility of using existing approaches as components makes the proposed algorithm flexible.

This paper extends the conditions of applicability of the algorithm from [3] since it works only for acyclic models. For the algorithm to find and process potential cycles in the event log, we will reuse the method

for detecting the repetitive behavior in a event log proposed and tested in [28, 29], which partially covers the general solution of the cycle detection problem.

2. Preliminaries

By $\mathbb{N}$ we denote the set of non-negative integers.

Let X be a set. A *multiset* m over the set X is a mapping: $m : X \to \mathbb{N}$, i. e., a multiset may contain several copies of the same element. For an element $x \in X$, we write $x \in m$, if $m(x) > 0$. For two multisets m, m' over X we write $m \subseteq m'$ iff $\forall x \in X : m(x) \leq m'(x)$ (the inclusion relation). The sum, the union and the subtraction of two multisets m and m' are defined as usual: $\forall x \in X : (m + m')(x) = m(x) + m'(x), (m \cup m')(x) = max(m(x), m'(x)), (m - m')(x) = m(x) - m'(x)$, if $m(x) - m'(x) \geq 0$, otherwise $(m - m')(x) = 0$. By $\mathcal{M}(X)$ we denote the set of all multisets over X.

For a set X, by X^* with elements of the form $\langle x_1, \ldots, x_k \rangle$ we denote the set of all finite sequences (words) over X, $\langle\rangle$ denotes the empty word, i. e., the word of zero length. The concatenation of two words w_1 and w_2 is denoted by $w_1 \cdot w_2$.

Let $Q \subseteq X$ be a subset of X. The projection $\restriction_Q$: $X^* \to Q^*$ is defined recursively as follows: $\langle\rangle \restriction_Q = \langle\rangle$, and for $\sigma \in X^*$ and $x \in X$:

$$(\sigma \cdot \langle x \rangle) \restriction_Q = \begin{cases} \sigma \restriction_Q \text{ if } x \notin Q \\ \sigma \restriction_Q \cdot \langle x \rangle \text{ if } x \in Q \end{cases}$$

We say that $X = X_1 \cup X_2 \cup \cdots \cup X_n$ is a partition of the set X if for all $1 \leq i, j \leq n$ such that $i \neq j$ we have $X_i \cap X_j = \emptyset$.

2.1. Petri nets

Let P and T be two finite disjoint sets of places and transitions, respectively, and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ be an arc-weight function. Let also A be a finite set of *event names* (or *activities*) representing observable actions or events, τ — a special label for *silent* or invisible action, $\lambda : T \to A \cup \{\tau\}$ is a transition labeling function. Then $N = (P, T, F, \lambda)$ is a *labeled Petri net*.

Graphically, a Petri net is designated as a bipartite graph, where places are represented by circles, transitions by boxes, and the flow relation F by directed arcs.

A *marking* in a Petri net $N = (P, T, F, \lambda)$ is a function $m : P \to \mathbb{N}$ mapping each place to some number of tokens (possibly zero). Hence, a marking in a Petri net may be considered as a multiset over its set of places. Tokens are graphically designated by black circles. A current marking m is represented by putting $m(p)$ tokens into each place $p \in P$. A *marked Petri net* (N, m_0) is a Petri net N together with its initial marking m_0.

For transition $t \in T$, its *preset* (denoted ${}^\bullet t$) and its *postset* (denoted $t^\bullet$) are defined as sets of its input and output places respectively, i. e., ${}^\bullet t = \{p \mid F(p, t) \neq 0\}$ and $t^\bullet = \{p \mid F(t, p) \neq 0\}$.

A transition $t \in T$ is *enabled* in a marking m, if for all $p \in {}^\bullet t$, $m(p) \geq F(p, t)$. An enabled transition t may *fire* yielding a new marking m', such that $m'(p) = m(p) - F(p, t) + F(t, p)$ for each $p \in P$ (denoted $m \xrightarrow{\lambda(t)} m'$, or just $m \to m'$). A marking m' is reachable from a marking m, if there exists a sequence of firings $m = m_0 \to m_1 \to \ldots m_k = m'$. By $\mathcal{R}(N, m)$ we denote the set of all markings reachable from marking m in a net N. A transition $t \in T$ is called *dead* for a marked net (N, m_0), if for each reachable marking $m \in \mathcal{R}(N, m_0)$, t is not enabled in m.

Let (N, m_0) be a marked Petri net with transitions labeled with activities from $A \cup \{\tau\}$, and let $m_0 \xrightarrow{a_1} m_1 \xrightarrow{a_2} \ldots$ be a finite or infinite sequence of firings in N, which starts from the initial marking m_0 and cannot be extended. Then a sequence of observable activities ρ, such that $\rho = \langle a_1, a_2, \ldots \rangle \restriction_A$, is called a *run*. For a finite run ρ, which corresponds to a sequence of firings $m_0 \xrightarrow{a_1} \ldots \xrightarrow{a_k} m_k$, we call m_0 and m_k its initial and final markings respectively.

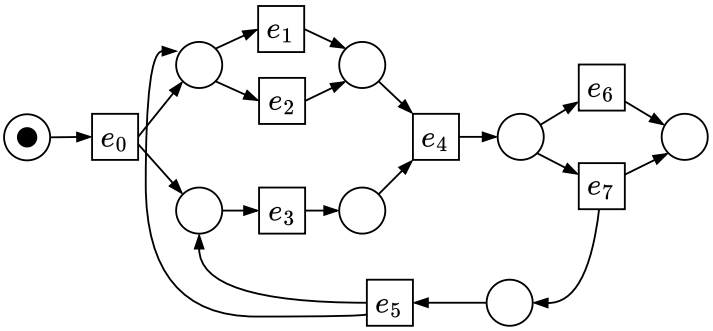


Fig. 1. A workflow net for handling compensation requests

In our study, we consider *workflow nets* — a special subclass of Petri nets [2] for workflow modeling. A *workflow net* is a (labeled) Petri net with two special places: i and f. These places mark the beginning and the ending of a workflow process.

A (labeled) marked Petri net $N = (P, T, F, \lambda, m_0)$ is called a workflow net (WF-net) if the following conditions hold:

1. There is one source place $i \in P$ and one sink place $f \in P$, such that ${}^\bullet i = f^\bullet = \emptyset$.
2. Every node from $P \cup T$ is on a path from i to f.
3. The initial marking m_0 in N contains the only token in its source place.

Given a WF-net, by $[in]$ we denote its initial marking with the only token in place i, and by $[fin]$ — its *final* marking with the only token in place f.

The example of a workflow net that simulates a simple process of handling ticket refund requests, is shown in Fig. 1 [30].

Soundness [2] is the main correctness property for workflow nets. A WF-net $N = (P, T, F, \lambda, [in])$ is called *sound*, if

1. For any marking $m \in R(N, [in])$, $[fin] \in \mathcal{R}(N, m)$;
2. If for some $m \in R(N, [in])$, $[fin] \subseteq m$, then $m = [fin]$;
3. There are no dead transitions in N.

2.2. Event logs

Most information systems record the history of their process execution into event logs. An *event record* usually contains case ID, an activity name, a time step, and some information about resources, data, etc. In the light of our research, we use case IDs for splitting an event log into traces, timestamps — for ordering events within each trace, and abstract from all event attributes except event names (activities).

Let A be a finite set of activities. A *trace* σ is a finite sequence of activities from A, i. e., $\sigma \in A^*$. By $\#a(\sigma)$ we denote the number of occurrences of activity a in trace σ.

An *event log* L is a finite multi-set of traces, i. e., $L \in \mathcal{M}(A^*)$. Let $X \subseteq A$. We extend projection $\restriction_X$ to event logs, i. e., for an event log $L \in M(A^*)$, its projection is the event log $L\restriction_X$, defined as the multiset of projections of all traces in L. In other words, $L\restriction_X(\sigma\restriction_X) = L(\sigma)$ for all $\sigma \in L$.

An important question is whether the event log matches the behavior of the process model and vice versa. There are several metrics to measure conformance between a WF-net and an event log. Specifically, *fitness* defines to what extend the log can be replayed by the model.

Let N be a WF-net with transition labels from A, an initial marking $[in]$, and a final marking $[fin]$. Let σ be a trace over A. We say that trace $\sigma = \langle a_1, \ldots, a_k \rangle$ *perfectly fits* N, if σ is a run in N with initial marking $[in]$ and final marking $[fin]$. An event log L perfectly fits N, if every trace from L perfectly fits N.

3. Discovering hierarchical WF-nets

3.1. Hierarchical WF-nets

Here, we define *hierarchical workflow* (HWF) nets with two levels of representing the process behavior. Transitions in a high-level WF-net are labeled by activities from $\tilde{A}$, while transitions in a set of low-level WF-nets are labeled by the corresponding low-level activities from A.

An HWF-net is a tuple $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, \ell)$, where:

1. $\tilde{N} = (\tilde{P}, \tilde{T}, \tilde{F}, \tilde{\lambda}, [\tilde{in}])$ is a WF-net, called a *high-level WF-net*, where $\tilde{\lambda}\colon \tilde{T} \to \tilde{A}$ is a transition labeling function;
2. $N_i = (P_i, T_i, F_i, \lambda_i, [in]_i)$ is a WF-net, called a *low-level WF-net* for $i = 1, 2, \ldots, k$ with a transition labeling function $\lambda_i\colon T_i \to A_i$, where $A_i \subseteq A$ is a subset of low-level activities for N_i, such that $A_1, A_2, \ldots, A_k$ — is a partitioning of A;
3. $\ell\colon \tilde{A} \to \{N_1, N_2, \ldots, N_k\}$ is a partial injective function mapping certain activities in $\tilde{A}$ to low-level WF-nets.

We refer to $\tilde{N}, N_1, N_2, \ldots, N_k$ as the components of $\mathcal{N}$. A marking for $\mathcal{N}$ is defined by the markings of its components. A *marking* in $\mathcal{N}$ is a set $\mathcal{M} = \{M, m_1, m_2, \ldots, m_k\}$, where M is a marking in $\tilde{N}$ and $m_1, m_2, \ldots, m_k$ are markings in $N_1, N_2, \ldots, N_k$ respectively.

The initial marking $\mathcal{M}_0$ for HWF-net $\tilde{N}$ contains exactly one token in the source place of $\tilde{N}$.

Let $dom(\ell)$ denote the domain of ℓ. We call the activities in $dom(\ell)$ — *high-level activities*. We also call low-level WF-nets *sub-processes*. Accordingly, every transition in a high-level WF-net $\mathcal{N}$ is assigned the corresponding low-level WF-net modeling the behavior of a sub-process.

Transition $t \in \tilde{T} \cup T_1 \cup \ldots T_k$ is enabled if it is enabled in its component by the ordinary firing rule, described in the previous section, for Petri nets.

There are the three following alternatives of transition firing in HWF-nets:

1. Let $t \in (\tilde{T} \setminus dom(\ell)) \cup T_1 \cup \ldots T_k$ be a transition enabled in $\mathcal{M}$. Then, the firing step $\mathcal{M} \xrightarrow{\lambda(t)} \mathcal{M}'$ is completed according to the standard rules, i. e., it only changes the marking in the low-level WF-net containing t.
2. Let $t \in dom(\ell)$ be a transition enabled in $\mathcal{M}$. Then, a silent firing step $\mathcal{M} \xrightarrow{\tau} \mathcal{M}'$ may be done, where $\mathcal{M}' = \{M', m_1', m_2', \ldots, m_k'\}$, such that $M'(p) = M(p) - F(p, t)$ for all $p \in \tilde{P}$, $m_i' = m_i + [in]_i$, where $[in]_i$ — is an initial marking for the sub-process N_i corresponding to the enabled transition t, $m_j' = m_j, \forall j \neq i$, and τ — is the invisible action, that takes all tokens from the input places for t and add the initial marking to the low-level WF-net N_i.
3. Let m_i be a marking in the low-level WF-net N_i that contains a token in its final place f. Then, a silent firing step $\mathcal{M} \xrightarrow{\tau} \mathcal{M}'$ may be done, where $\mathcal{M}' = \{M', m_1', m_2', \ldots, m_k'\}$, such that $M'(p) = M(p) + F(t, p)$ for all $p \in \tilde{P}$, where t is the transition in the high-level WF-net $\tilde{N}$ corresponding to N_i, and $m_i' = m_i' - [fin]_i$, where $[fin]_i$ – is a final marking for N_i, $m_j' = m_j, \forall j \neq i$.

The example of an HWF-net is provided in Fig. 2. We do not impose specific restrictions on the number of input and output places a transition in a high-level WF-net can have. In Fig. 2, we only show the refinement of two transitions t_1 and t_2 in the high-level WF-net $\tilde{N}$ with two low-level WF-nets N_1 and N_2. They represent the low-level behavior of two sub-processes α_1 and α_2, respectively. Note that, if a low-level WF-net corresponding to a high-level activity contains the single transition, we still represent such a sub-process with an individual WF-net.

We next consider the operational semantics of an HWF-net by defining its run. For what follows, let $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, \ell)$ be an HWF-net , where $\tilde{N}$ — is a high-level net and $N_1, N_2, \ldots, N_k$ — are nets for its sub-processes.

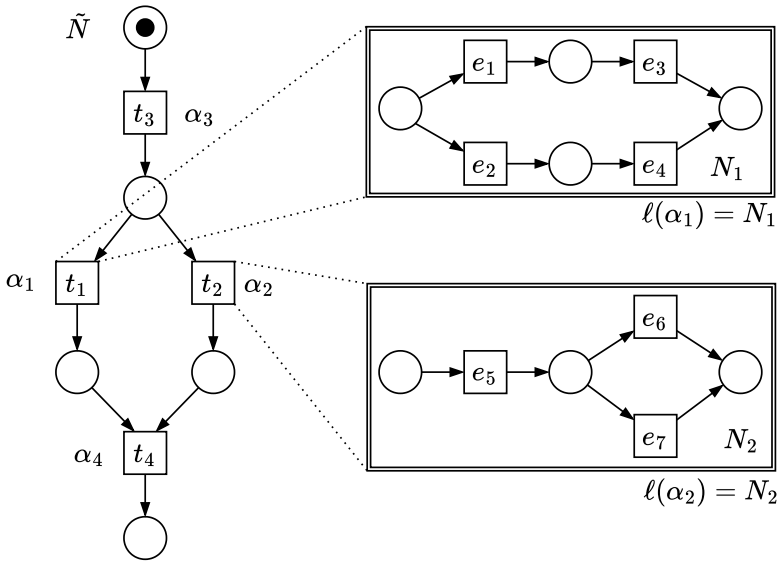


Fig. 2. An HWF-net with two refined transitions

Intuitively, a set of transitions enabled in in a high-level WF-net determines the set of sub-processes for which we can start to fire their low-level transitions. Transition firing, as described above, corresponds to starting, executing and terminating sub-processes, which can be run concurrently.

Let t' be a transition enabled in the current marking $\mathcal{M}$ of an HWF-net. The following options of $\tilde{\mathcal{M}} \xrightarrow{\lambda(t')} \tilde{\mathcal{M}}'$ are possible.

If there are high-level transitions, enabled at $\tilde{m}$ in a high-level WF-net, sharing common places, then there is a *conflict*. We can choose, which sub-process to start, while the other sub-processes corresponding to conflicting transitions in a high-level WF-net will not be able to be executed. For example, high-level transitions t_1 and t_2, once enabled, will be in conflict, and we can start only one of the corresponding sub-processes, N_1 or N_2. Firing a transition in a high-level WF-net is complete if the corresponding low-level WF-net reaches its final marking.

For instance, let us again consider the HWF-net shown in Fig. 2. After firing high-level transition t_3 and executing a corresponding sub-process α_3 (not provided in Fig. 2), two high-level transitions t_1 and t_2 become enabled. They share a common place, i. e., high-level transitions t_1 and t_2 are in conflict. Thus, we can execute exactly one of the corresponding sub-processes α_1 (low-level WF-net N_1) and α_2 (low-level WF-net N_2). We can, for instance, obtain a sequence $\rho = \langle \alpha_3, e_5, e_6, \alpha_4 \rangle$ which will represent a possible run of the HWF-net from Fig. 2. Note that high-level activities α_3 and α_4 should also be replaced with corresponding sub-process runs.

Lastly, we give a straightforward approach to transforming an HWF-net $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, \ell)$ to the corresponding *equivalent* classic WF-net denoted by $\mathbf{eq}(\mathcal{N}) = (P, T, F, \lambda, [in])$. We need to replace transitions in a high-level WF-net with their sub-process implementation given by low-level WF-net corresponding by ℓ. In addition, we need to remove tokens from the input places of low-level WF-nets. When a transition t in a high-level WF-net $\tilde{N}$ is replaced by a low-level WF-net N_i, we need to fuse a source place in N_i with all input places of t and to fuse a sink place in N_i with all output places of t. By construction, $\mathbf{eq}(\mathcal{N})$ is a WF-net.

For instance, the WF-net $\mathbf{eq}(\mathcal{N})$ equivalent the HWF-net $\mathcal{N}$, shown in Fig. 2, is provided in Fig. 3. We replaced transition t_1 with N_1 and transition t_2 with N_2 as determined by the labels of low-level WF-nets. This figure also shows the double-line contours of corresponding high-level transitions.

Proposition 1 gives the main connection between an HWF-net and its classical representation.

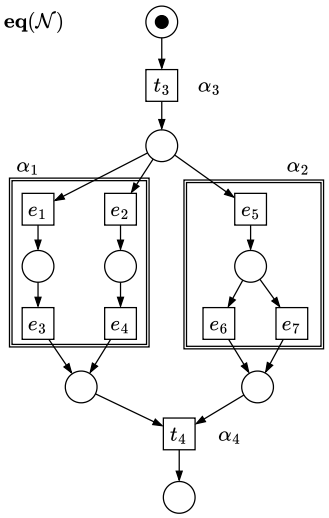


Fig. 3. The WF-net equivalent to the HWF-net in Fig. 2

Proposition 1. *Let $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, \ell)$ be an HWF-net, and* $\mathbf{eq}(\mathcal{N})$ *be the corresponding equivalent WF-net. A sequence ρ of activities is a run in $\mathcal{N}$ if and only if ρ is a run in* $\mathbf{eq}(\mathcal{N})$.

In other words, a run in HWF-net $\mathcal{N}$ is also a run in the corresponding classical WF-net N and vice versa. Proof of this proposition directly follows from the construction of the classical WF-net and from the way we define the sequential semantics of a hierarchical WF-net and from semantic definition.

To sum up, for each HWF-net we can effectively build a classical WF-net having exactly the same behavior.

3.2. Events partitioning

We suppose that partitioning the set of low-level activities A into subsets $A_1, \ldots A_k$ is made either by an expert, or automatically based on some information contained in extended action records, such as resources or data. In Section 5, we give two examples of partitioning activities for a real log. Then we suppose that a sub-process is defined by its set of activities, and we suppose that sets of activities for two sub-processes do not intersect. If it is not the case and two sub-processes include some common activities like "close the file", one can easily distinguish them by appending the resource or file name to the activity identifier.

Let L be a log over a set A of activities, and let $A = A_1 \cup A_2 \cup \cdots \cup A_k$ be a partition of A. Let also $\tilde{A} = \{\alpha_1, \alpha_2, \ldots \alpha_k\}$ be a set of high-level activities (sub-process names).

The problem is to construct an HWF-net $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, l)$, where for each $i \in [1, k]$, N_i is a sub-process (WF-net), labeled by α_i, with transitions labeled by low-level activities from A_i. The runs of $\mathcal{N}$ should conform to traces from L.

Another important remark concerning partitioning activities: we suppose that it does not violate the log control flow. Specifically, if there are iterations in the process, then for a set of iterated activities B and for each sub-process activities set A_i, we assume that either $B \cap A_i = \emptyset$, or $B \subseteq A_i$, or $A_i \subseteq B$. Note that this is a reasonable constraint, taking into account the concept of a sub-process. If it is still not the case, i. e., , only a part of A_i activities are iterated, then the partition can be refined, such that A_i is split into two subsets: a subset of iterated activities and the remainder.

For example, consider a low-level WF-net discovered from an event log shown in Fig. 4. Suppose that the set B of the iterated activities includes $\{a_3, a_4, b_3, b_4\}$ and that the low-level events are partitioned into two subsets $A_1 = \{a_1, a_2, a_3, a_4\}$ and $A_2 = \{b_1, b_2, b_3, b_4\}$.

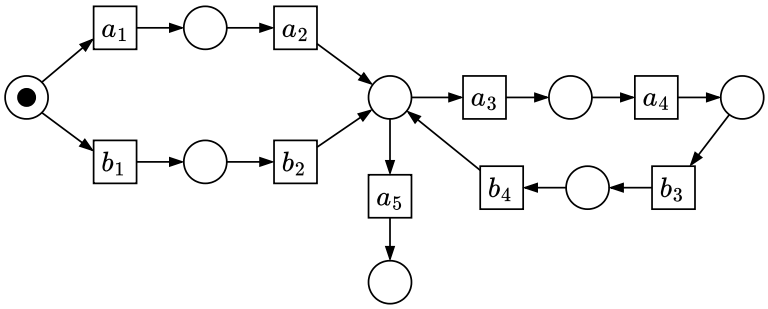


Fig. 4. The example cycle and high-level activity inconsistency

Using the proposed events partitioning, we cannot represent this model as a high-level WF-net, since the iterated activities belong to different high-level events. In addition, in the set of iterated activities B, low-level events a_3 and a_4 are always executed before b_3, b_4. Thus, one needs to revise this partitioning of low-level events in such a way that either B is fully included into a high-level activity A_i, or a high-level activity A_i is a part of a cycle.

3.3. The proposed solution

Here, we describe the main ideas and the structure of the algorithm to discover the hierarchical WF-net from an event log.

Let L be a log with activities from A, and let $A = A_1 \cup A_2 \cup \cdots \cup A_k$ be a partition of A. Let $\tilde{A} = \{\alpha_1, \alpha_2, \ldots \alpha_k\}$ be a set of high-level activities (sub-process names).

A hierarchical WF-net $\mathcal{N}$ (HWF-net) consists of a high-level WF-net $\tilde{N}$ with activities $\tilde{A} = \{\alpha_1, \ldots, \alpha_k\}$, and k sub-process WF-nets $N_1, N_2, \ldots, N_k$, where for each N_i, all its activities belong to A_i.

Sub-process WF-nets $N_1, N_2, \ldots, N_k$ can be discovered directly. To discover N_i, we filter log L to $L_i = L\uparrow_{A_i}$. Then we apply one of popular algorithms (e. g., Inductive Miner) to discover a WF-net from event log L_i. The fitness and precision of the obtained model depend solely on the choice of the discovery algorithm.

Example 1. *Consider an event log L of a business process over the set of low-level activities $A = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\}$. Let L be an event log, such that $L = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6\}$, where $\sigma_1 = t_1 t_2 t_5 t_3 t_4 t_3 t_6 t_{12} t_7 t_3 t_{11}$, $\sigma_2 = t_1\ t_5\ t_6\ t_2\ t_3\ t_7\ t_8\ t_9\ t_{10}\ t_5\ t_6\ t_{11}$, $\sigma_3 = t_1\ t_2\ t_3\ t_5\ t_6\ t_{11}$, $\sigma_4 = t_1\ t_2\ t_5\ t_3\ t_{12}\ t_6\ t_3\ t_{11}$, $\sigma_5 = t_1\ t_2\ t_5\ t_3\ t_4\ t_6\ t_7\ t_3\ t_{12}\ t_8\ t_9\ t_{10}\ t_5\ t_6\ t_3\ t_{11}$, $\sigma_6 = t_1\ t_2\ t_3\ t_4\ t_3\ t_5\ t_6\ t_{11}$. A partition for this low-level activity set is $A = A_0 \cup A_1 \cup \cdots \cup A_5$, where $A_0 = \{t_1\}$, $A_1 = \{t_2\}$, $A_2 = \{t_3, t_4, t_{12}\}$, $A_3 = \{t_5, t_6\}$, $A_4 = \{t_7, t_8, t_9, t_{10}\}$, $A_5 = \{t_{11}\}$. A set of high-level activities for our example is $\tilde{A} = \{\alpha_0, \ldots, \alpha_5\}$, such that for every high-level activity from $\tilde{A}$ the corresponding sub-process $\alpha_i \in \tilde{A}, i \leq |\tilde{A}|$ contains activities only from $A_i \in A$.*

As the high-level WF-net of the hierarchical WF-net we consider the workflow net shown in Fig. 5 with activities labeled with $\tilde{A}$.

The existing process mining algorithm should be able to discover the workflow net presented in Fig. 6 directly from the low-level event log L. In this low-level net, we can also see that sub-processes, corresponding to high-level activities $\alpha_0, \alpha_1, \ldots \alpha_5$, have the same relations between activities as in the net from Fig. 5. For simplicity, the sub-processes α_0, α_1 and α_5, consisting of the single transition, are not highlighted with double rectangles.

Discovering a high-level WF-net is not so easy and is quite a challenge. The main problem with it is caused by the possible interleaving of concurrent sub-processes and iteration. A naive solution could be to replace each activity $t_j \in A_i$ by α_i — the name of the corresponding sub-process — in the log L. Then we need to remove *stuttering*, i. e., to replace, wherever possible, several sequential occurrences of the same high-level activity by a single activity. Then we apply the one of popular discovery algorithms to the obtained log over the set $\tilde{A}$ of activities. However, this does not work, due to the presence of the patterns of behavior other than the simple sequential execution.

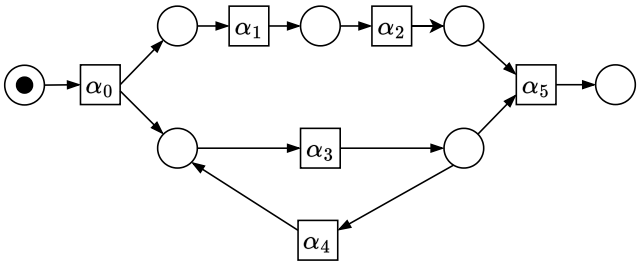


Fig. 5. The high-level workflow net for Fig. 6

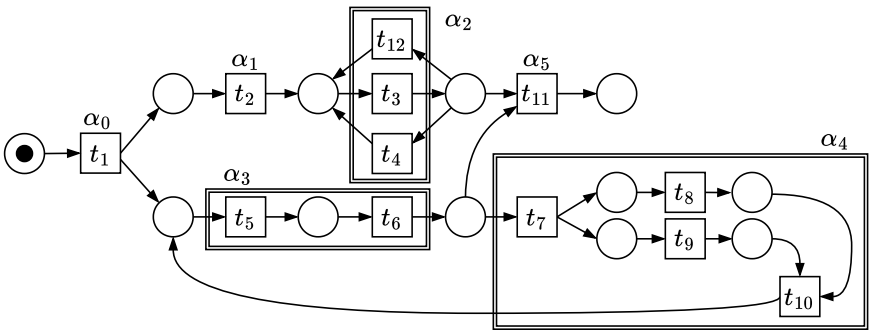


Fig. 6. An example of a low-level net with cycles inside the subprocess and between two subprocesses corresponding to the high-level net from Fig. 5

Consider the examples in Fig. 7. Fragment (a) in Fig. 7 shows two concurrent sub-processes β and γ, executing after sub-process α, which consists of the single transition. After replacing low-level activities with the corresponding sub-process names and removing stuttering, for the fragment (a), we get the following runs: $\langle \alpha, \beta, \gamma, \dots \rangle$, $\langle \alpha, \gamma, \beta, \dots \rangle$, $\langle \alpha, \beta, \gamma, \beta, \gamma, \dots \rangle$, $\langle \alpha, \gamma, \beta, \gamma, \beta, \dots \rangle$ etc. Fragment (b) in Fig. 7 shows a cycle. The body of this cycle is the sequence of two sub-processes β and γ. Among runs for the fragment (b) we also have $\langle \alpha, \beta, \gamma, \dots \rangle$, $\langle \alpha, \beta, \gamma, \beta, \gamma, \dots \rangle$. That is why iterations should be considered separately.

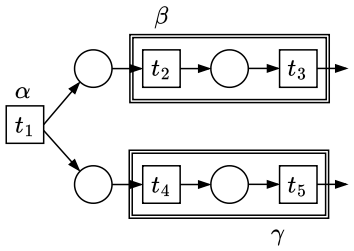
Discovering high-level WF-nets for acyclic processes, i. e., logs without iteration, was studied earlier in [3] where all details can be found. Here, we refer to this algorithm as Algorithm $\mathfrak{A}_0$ and illustrate it with the example in Fig. 7(a). Algorithm $\mathfrak{A}_0$, discovering a high-level WF-model from a log L without cycles, reduces this problem to the classical discovery problem, which can be solved by many popular algorithms, such as Inductive Miner. Therefore, Algorithm $\mathfrak{A}_0$ can be parameterized by Algorithm $\mathfrak{D}$, i. e., an already known algorithm for solving the classical discovery problem.

Algorithm $\mathfrak{A}_0(\mathfrak{D})$:

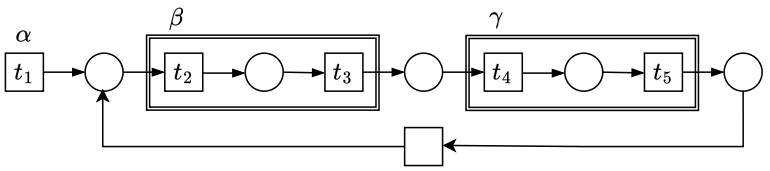
Step 1. For all traces in L, replace each activity with the corresponding sub-process names and remove stuttering.

Step 2. For each trace σ that contains $t_i \in \sigma, i \leq |\sigma|$ such as $\#t_i > 1$, find all occurrence of t_i in σ. For each occurrence's position k create a clone of σ, delete from it every t_i except the one at the position k and remove (newly formed) stuttering. Replace σ with the set of all obtained clones of σ.
For example, the trace $\langle \alpha, \beta, \gamma, \beta, \gamma, \dots \rangle$ will be replaced by two traces $\langle \alpha, \beta, \gamma, \dots \rangle$ and $\langle \alpha, \gamma, \beta, \dots \rangle$ obtained by keeping the first occurrences of β and γ, and correspondingly by keeping the first occurrence of γ and the second occurrence of β. In this example, constructing clones by keeping other occurrences of γ does not generate new traces.

Step 3. Let $\tilde{L}$ be the resulting log from executing two previous steps. To obtain a high-level WF-net $\tilde{N}$, apply the given as the input parameter Algorithm $\mathfrak{D}$, to discover a WF-net from event log $\tilde{L}$.



(a) concurrent sub-processes



(b) sub-processes inside a loop

Fig. 7. Interleaving and iteration of sub-processes

It was proven in [3] that if an algorithm used in Step 3 of Algorithm $\mathfrak{A}_0$ for each input log L discovers a WF-net perfectly fitting L, then Algorithm $\mathfrak{A}_0$, given a log L without repetitive behavior, produces an HWF-net $\mathcal{N}$ such that $\mathbf{eq}(\mathcal{N})$ perfectly fits L.

3.4. Detecting cycles in event logs

Now we come to logs with the repetitive behavior. The main idea here is to represent a loop body as a subset of its activities. Then a body of a loop can be considered as a sub-process with a new loop sub-process name. To discover the repetitive behavior, we use the method from [28], which allow us to determine causal, concurrency, and repetitive relations between events in an event log. Actually, for our purpose we need only repetitive relations and the loop discovery based on them. We call the loop discovery algorithm — Algorithm $\mathfrak{B}$. The strategy of this algorithm includes the pruning of interleaving tasks in an event trace to separate them by the supports of minimal transition invariants (t-invariants) — firing sequences, which represent potential cycles (see [31] for details). The procedure to obtain the t-invariants operates recursively on every task in trace σ_i from the most external cycle in every trace to the smaller nested cycles.

Let us consider the application of Algorithm $\mathfrak{B}$ in the following example.

Consider the log L from Example 1. Extracting the information about cycles is to derive the direct causal and concurrency relations between transitions in the event log L. The result of this step is shown in Table 1.

Algorithm $\mathfrak{B}$ finds all sub-sequences containing the repetitive behaviour, i. e., if in a trace σ there is an event t_i such that $\#t_i > 1$, then the sub-sequence, which contains repetitive behavior, should start with t_i until the next occurrence of t_i in σ. In [28], such sequences are called *cycs*. In L, the following cycs are detected: $cyc_1 = \{t_3\ t_4\}$, $cyc_2 = \{t_3\ t_6\ t_{12}\}$ in σ_1, $cyc_3 = \{t_5\ t_6\ t_2\ t_3\ t_7\ t_8\ t_9\ t_{10}\}$ in σ_2, $cyc_4 = \{t_3\ t_{12}\ t_6\}$ in σ_4, $cyc_5 = \{t_5\ t_3\ t_4\ t_6\ t_7\ t_3\ t_8\ t_9\ t_{10}\}$, $cyc_6 = \{t_3\ t_4\ t_6\ t_7\}$ in σ_5, $cyc_7 = \{t_3\ t_4\}$ in σ_6. We can see that cyc_7 is equal to cyc_1. Thus, we can merge them because the goal is to obtain a set of different cycs, and their frequency is not important in this case. In addition, we can see that there is another cyc in cyc_5. Furthermore, according to [28], cyc_5 is not the elementary cyc because $\exists t_i \in cyc$ such that $\#t_i > 1$, and we could derive a smaller nested elementary $cyc' = \{t_3\ t_4\ t_6\ t_7\}$ from it.

The next step of the Algorithm $\mathfrak{B}$ is to build the causality graph for each cyc found. In the process of building the causality graph, we use the relations between activities. These relations can be extracted from the input event log using any suitable process discovery algorithm. The set of relations between activities for the log L is presented in Table 1.

Now, having all relations, we can easily build the causality graph for each cyc. In [28], it is proposed that a strongly connected component of the causality graph for a cyc is also the support of a minimal t-invariant in the final model. The resulting graphs are shown in Fig. 8. Note that the graphs for cyc_1 and cyc_7 are equal because they contain the same transitions, and this is also true for cyc_2 and cyc_4. Therefore, we have depicted them only once (Fig. 8a and Fig. 8b, respectively).

Let us consider the case of cyc_5 separately because the cyc contains a nested cyc. In this case, first, we processed the nested (elementary) cyc' and built the causality graph for it. The final causality graph

Table 1. Relations between transitions in Example 1

T_i	Causal Relationship	Concurrent Relationship
t_1	t_2, t_5	
t_2	t_3, t_5	
t_3	t_{11}, t_{12}	t_5, t_6, t_7
t_4	t_6	
t_5	t_6	t_3
t_6	t_2, t_7, t_{11}	t_3, t_4, t_{12}
t_7	t_8	t_3
t_8	t_9	
t_9	t_{10}	
t_{10}	t_5	
t_{12}	t_7, t_8	t_6

for cyc' is equal to the causality graph for cyc_6 presented in Fig. 8e. The strongly connected component in this case is equal to cyc_1, and it is already in our set of t-invariant supports. The next step is to remove the detected t-invariant support of a nested cyc from cyc_5. Afterwards, we get the new sequence of transitions $cyc'_5 = \{t_5\ t_6\ t_7\ t_3\ t_8\ t_9\ t_{10}\}$, and we can build the causality graph for this new elementary cyc. The resulting graph is shown in Fig. 8d.

As a result of applying Algorithm $\mathfrak{B}$ to the log L, we obtain the following set of t-invariant supports: $Y_1 = \{t_3, t_4\},\ Y_2 = \{t_3, t_{12}\},\ Y_3 = \{t_5, t_6, t_7, t_8, t_9, t_{10}\}$.

Algorithm $\mathfrak{B}$ allows us to detect the bodies of elementary cycles as sets of their activities and process them recursively, starting with inner elementary cycles. Thus, at each iteration we are dealing with a loop body without internal loops. To obtain a sub-trace, corresponding to the loop body with a set of activities B from a log trace σ, we construct the projection $\sigma\restriction_B$. After filtering all current traces in this way, we get an event log for discovering a WF-net that simulates the behavior of the loop body using Algorithm $\mathfrak{A}_0$.

As mentioned in Section 3.2, partitioning of low-level events should not violate the log control flow from the point of view on the iterated behavior. Here we give a more precise representation of this requirement based on the results produced by Algorithm $\mathfrak{B}$ to detect cycle bodies, in terms of t-invariants.

Let $B = \{b_1, \ldots, b_n\}$ be the cycle bodies found by Algorithm $\mathfrak{B}$ and L be an event log over $A = A_1 \cup A_2 \cup \cdots \cup A_k$, where k is a number of high-level activities. Then, the partition of events $A = A_1 \cup A_2 \cup \cdots \cup A_k$ is *consistent* with B iff $\forall b_i \in B$ and $A_j \in A$ one of the following holds:

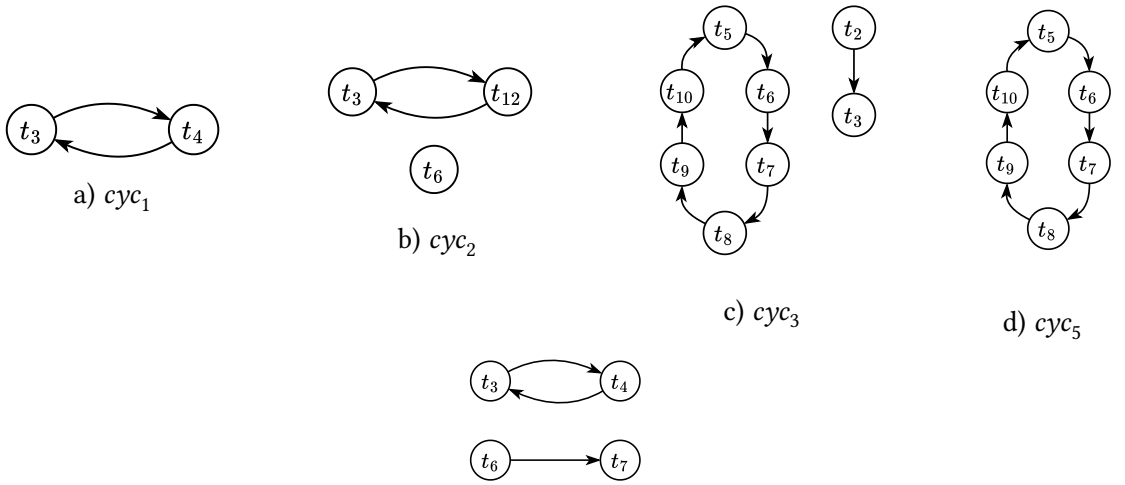
1. $b_i \cap A_j = \emptyset$;
2. $b_i \subseteq A_j$;
3. $A_j \subseteq b_i$.

The example in Fig. 4 discussed earlier in Section 3.2 shows that the inconsistency between the event partitioning and iterated behavior does not allow to construct an WF-net, since high-level events have common parts. Inconsistency can be corrected by revising the initial event partitioning.

The resulting high-level WF-net is then constructed recursively by replacing, the body of each detected loop with the name of its sub-process, starting with the inner loops. Note that if, after this step, in the WF-net there are transitions that are involved in more than one cycle, we need to merge all the same named transitions into a single one with that name. This also applies to places, because logically some of them should also be merged, depending on the activities relations. As the strategy of the merging algorithm, we also use the one proposed in [28]. We call the algorithm for merging transitions by activities correspondence Algorithm $\mathfrak{C}$.

4. Algorithm for discovering HWF-nets from low-level event logs

Here, we describe our main discovery algorithm in more detail.



a) cyc_1 b) cyc_2 c) cyc_3 d) cyc_5 e) cyc_6

Fig. 8. Causality graphs for every cyc from L

Let A be a set of activities and L — a log over A. Let then $A = A_1 \cup \cdots \cup A_k$ be a partition of A. Let $\tilde{A} = \{\alpha_1, \ldots, \alpha_k\}$ be a set of sub-process names. For $i \in [1, k]$, A_i is a set of activities of a sub-process α_i.

Then Algorithm $\mathfrak{A}(\mathfrak{D})$ constructs an HWF-net $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, \ell)$ with high-level WF-net $\tilde{N} = (\tilde{P}, \tilde{T}, \tilde{F}, \tilde{\lambda}, [\tilde{in}])$, where $\tilde{\lambda}\colon \tilde{T} \to \tilde{A}$ and for each $\alpha_i \in \tilde{A}$, $\ell(\alpha_i) = N_i$, i. e., sub-process named with α_i corresponds to low-level WF-net N_i in $\mathcal{N}$.

By $\tilde{B} = \{\beta_0, \beta_1, \ldots, \beta_m\}$ we denote a set of cycle names and by ℓ_B — a function which maps each name from $\tilde{B}$ to a WF-net that implements the cycle with this name. For a WF-net N, denote by $Loop(N)$ a WF-net that is a loop with body N.

Algorithm $\mathfrak{A}(\mathfrak{D})$:

Step 1. Apply Algorithm $\mathfrak{B}$ to L to find a set $B = \{b_1, \ldots, b_m\}$ of cycle bodies, where every $b \in B$ is a set of activities in some cycle and $m = |B|$. A cycle name for each $b \in B$ will be stored in a set $\tilde{B}$ where $|\tilde{B}| = |B|$. The correspondence between every $b \in B$ and its name $\beta \in \tilde{B}$ is defined by index, i. e., for each set of cycle activities $b_q \in B$ a cycle name will be $\beta_q \in \tilde{B}$, where $q = 1, 2, \ldots, m$ is the index.

Step 2. Construct the projection $L\restriction_{b_i}$ for each $b \in B$ and apply Algorithm $\mathfrak{A}_0(\mathfrak{D})$ to it (with respect to the partition $A = A_1 \cup \cdots \cup A_k$). Let $\tilde{N}$ be the resulting high-level WF-net over the set $\tilde{A}$ of sub-process names. Let $N_1, \ldots, N_j$ — resulting WF-nets for sub-processes with names $\alpha_1, \ldots, \alpha_j$. Let $N_{\beta_1}, \ldots, N_{\beta_m}$ — resulting WF-nets for sub-processes within the cycle.

Step 3. Let $\ell_B(\beta_1) = N_{\beta_1}, \ldots, \ell_B(\beta_m) = N_{\beta_j}$. Let also $\ell(A_1) = N_1, \ldots, \ell(A_j) = N_j$.

Step 4. For every $\sigma \in L$ such as $b_j \subseteq \sigma$ and $i = 1, 2, \ldots, |L|$ replace by β_j all occurrences of activities from b_j in σ_i and remove stuttering.

Step 5. Let A_{new} be a current set of activities such as $A_{new} = A \cup \tilde{B}$ and $\tilde{A}_{new}$ be a current partition of A_{new} such as $\tilde{A}_{new} = \tilde{A} \cup \tilde{B}$.

Step 6. Apply Algorithm $\mathfrak{A}_0(\mathfrak{D})$ to the log L obtained at *Step 4* with respect to the current partition of activities $\tilde{A}_{new}$. Let $\tilde{N}$ be a resulting high-level WF-net.

Step 7. For each $\beta \in \tilde{B}$, replace a transition labeled by β in $\tilde{N}$ with the sub-process $Loop(\ell_B(\beta))$.

Step 8. For each pair of transition $\tilde{t_i}, \tilde{t_j} \in \tilde{T}$ from $\tilde{N}$, which are corresponding to the same α sub-process name, apply the merging Algorithm $\mathfrak{C}$.

The resulting net $\tilde{N}$ is a high-level WF-net for the HWF-net constructed by Algorithm. Its low-level WF-nets, which are defined by function ℓ, are also built during Algorithm operation.

Correctness of Algorithm $\mathfrak{A}(\mathfrak{D})$ is justified by the following statement.

Theorem 1. *Let A be a set of activities and L — a log over A. Let also $A = A_1 \cup \cdots \cup A_k$ be a partition of A, and $\tilde{A} = \{\alpha_1, \ldots, \alpha_k\}$ — a set of sub-process names.*

If Algorithm $\mathfrak{D}$, given a log L', discovers a WF-net N' such that N' perfectly fits L', then Algorithm $\mathfrak{A}(\mathfrak{D})$ constructs an HWF-net $\mathcal{N} = (\tilde{N}, N_1, N_2, \ldots, N_k, \ell)$, such that $\mathcal{N}$ perfectly fits L according to the substitution.

Proof. To prove that an HWF-net built using Algorithm $\mathfrak{A}(\mathfrak{D})$ perfectly fits the input log, provided that Algorithm $\mathfrak{D}$ discovers models with perfect fitness, we use three previously proven assertions, namely:

1. The theorem proven in [3] states that when $\mathfrak{D}$ is an discovery algorithm with perfect fitness, Algorithm $\mathfrak{A}_0(\mathfrak{D})$ discovers a high-level WF-net, whose refinement perfectly fits the input log without repetitions (the log of an acyclic process).
2. In [28] it is proven that, given a log L, Algorithm $\mathfrak{B}$ correctly finds in L all repetitive components that correspond to supports of t-invariants in the Petri net model for L.
3. Proposition 1 in Subsection 3.1 justify correctness of refining a high-level WF-net by substituting sub-process modules for high-level transitions.
4. In [28] it is proven that the given merging strategy of Algorithm $\mathfrak{C}$ correctly merges all equally named transitions after substitutions of the processes corresponding to the components of repetitive behavior to the Petri net model for L.

□

Taking the above into consideration, proving the theorem is straightforward, though quite technical. Thus, we informally describe the logic of the proof here.

Let Algorithm $\mathfrak{D}$ be a discovery algorithm that discovers a perfectly fitting WF-net for a given event log.

From *Step 1* to *Step 5* of the algorithm, repetitive components, i. e., cycles are processed. At *Step 1*, all inner elementary repetitive components in the log are discovered using Algorithm $\mathfrak{B}$. The activities of every component are those of some inner loop body, which do not have repetitions. Then, a WF-net N for this loop body is correctly discovered using Algorithm $\mathfrak{A}_0(\mathfrak{D})$, and the loop itself is folded into a high-level activity β, and N is kept as the value $\ell_B(\beta)$. WF-nets for sub-processes within the body of this loop are also discovered by Algorithm $\mathfrak{A}_0(\mathfrak{D})$ and accumulated by ℓ. If a loop activity β is contained in another loop body, then with the one more iteration of Step 1, the upper loop N' is discovered, the transition labeled with β in it is replaced with N, and N' is itself folded into a new high-level activity.

After processing all loops, Algorithm proceeds to *Step 6*, where after reducing all loops to high-level activities, Algorithm $\mathfrak{A}_0(\mathfrak{D})$ is applied to a log without repetitions.

In *Step 7* all transitions labeled with loop activities in a high-level and low-level WF-nets are replaced by WF-nets for these loops, kept by ℓ_B.

Step 8 merges all equally named transitions that corresponds to the same activity in the event log.

That is why we can see that, while Algorithm $\mathfrak{A}_0(\mathfrak{D})$ ensures the perfect fitness between the acyclic fragments of the model (when loops are folded into transitions), Algorithm $\mathfrak{B}$ ensures correct processing of cyclic behavior, and Proposition 1 guarantees that replacing loop activities by the corresponding loop WF-nets does not violate perfect fitness, the main algorithm provides systematic log processing and model construction.

5. Experimental evaluation

In this section, we report the main outcomes from a series of experiments conducted to evaluate the algorithm for discovering two-level hierarchical process models from event logs.

To support the automated experimental evaluation, we implemented the hierarchical process discovery algorithm described in the previous section using the open source library PM4Py [32]. The source files for our implementation are published on the open GitHub repository [33]. We conducted experiments using two kinds of event logs:

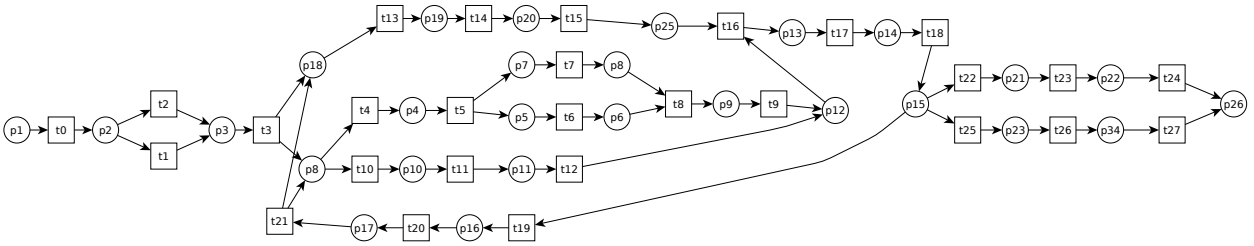


Fig. 9. A classical WF-net with generated by refining the WF-net in Fig. 1

1. *Artificial* event logs generated by manually prepared process models.
2. *Real-life* event logs provided by various information systems.

Event logs are encoded in a standard format as XML-based XES files.

To assess the quality of the algorithm quality, we will use several metrics of *conformance checking*. *Conformance checking* is an important part of process mining along with process discovery [34]. The main aim of conformance checking is to evaluate the quality of process discovery algorithm by estimating the corresponding quality of discovered process models. Conformance checking provides four main quality dimensions. *Fitness* estimates the extent to which a discovered process model can execute traces in an event log. A model perfectly fits an event log if it can execute all traces in an event log. According to Theorem 1, the hierarchical process discovery algorithm yields perfectly fitting process models. *Precision* evaluates the ratio between the behavior allowed by a process model and the one not recorded in an event log. A model with perfect precision can only execute traces in an event log. The perfect precision limits the use of a process model since an event log represents only a finite "snapshot" of all possible process executions. Generalization and precision are two dual metrics. The fourth metric, *simplicity*, captures the structural complexity of a discovered model. We improve simplicity by the two-level structure of a discovered process models.

Within the experimental evaluation, we estimated fitness and precision of process models discovered from artificially generated and real-life event logs. Fitness was estimated using alignments between a process model and an event log as defined in [35]. The precision was estimated using the complex ETC-align measures proposed in [36]. Both measures are values in the interval $[0, 1]$. As a discovery algorithm, we used Inductive Miner.

5.1. Discovering HWF-Nets from Artificial Event Logs

The high-level source for generating artificial low-level event logs was the Petri net shown earlier in Fig. 1. In this model, we refined its transitions with different sub-processes containing sequential, parallel and cyclic executions of low-level events. The example of refining the Petri net from Fig. 1 is shown in Fig. 9, where we show the corresponding classical representation of an HWF-net.

Generation of low-level event logs from the prepared model was implemented using the algorithm presented in [37]. Afterwards, we transform a low-level event log into a high-level event log by grouping low-level events into a single high-level event and by extracting information about cyclic behavior.

The corresponding high-level WF-net discovered from the artificial low-level event log that is generated from the WF-net shown in 9 is provided in Fig. 10. Intuitively, one can see that this high-level WF-net is rather similar to the original Petri net from Fig. 1.

As for the quality evaluation for the above presented high-level model, we have the following:

1. The discovered high-level WF-net perfectly fits the high-level event log obtained from a low-level log, where we identified cycles and grouped activities correspondingly.
2. The classical WF-net obtained by refining transitions in a discovered high-level WF-net by discovered sub-nets perfectly fits the low-level log.

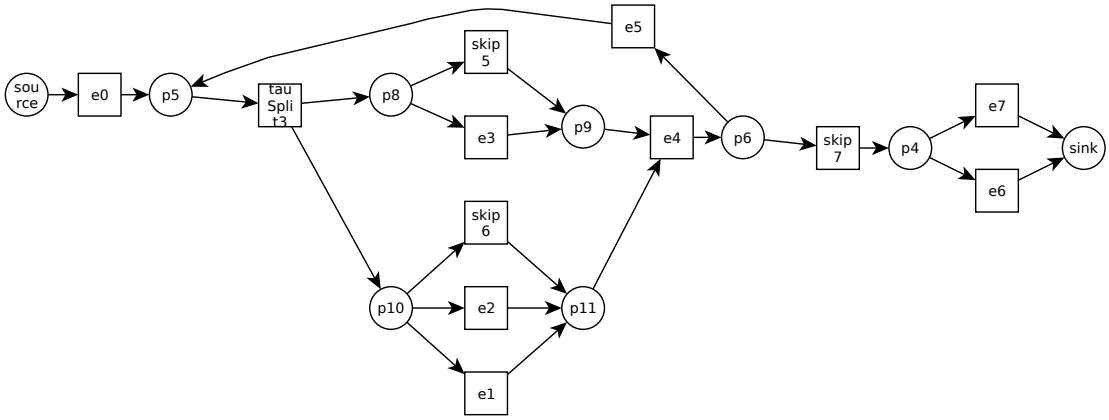


Fig. 10. A high-level WF-net discovered from an event log generated by refining the WF-net in Fig. 9

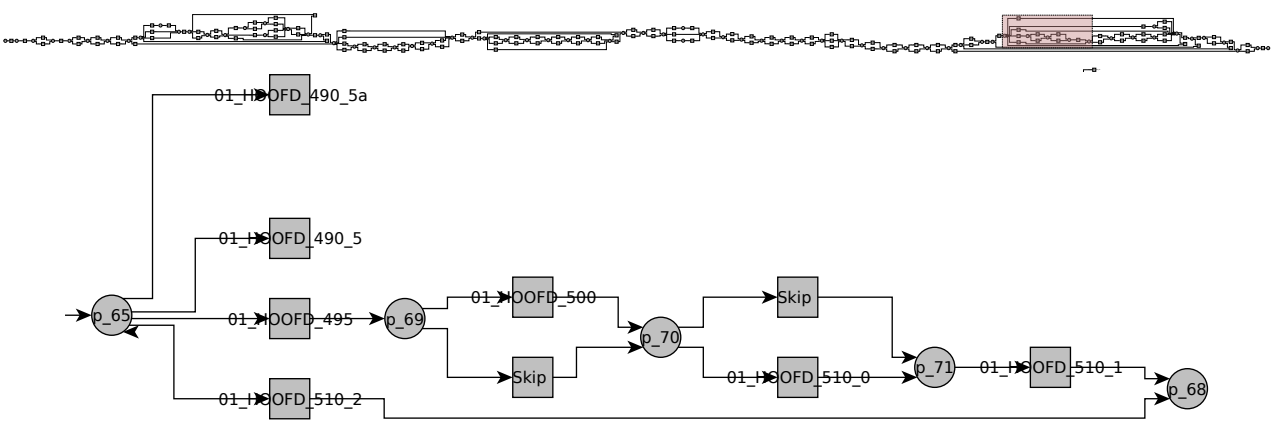


Fig. 11. A classical WF-net discovered from *BPI Challenge 2015* event log

Other examples of process models that were used for the artificial event log generation are also provided in the main repository [33].

5.2. Discovering HWF-nets from real-life event logs

We used two real-life event logs provided by *Business Process Intelligence Challenge (BPI Challenge)* 2015 and 2017 [38]. These event logs were also enriched with additional statistical information about classical process models.

The *BPI Challenge 2015* event log was provided by five Dutch municipalities. The cases in this event log contain information on the main application and objection procedures in various stages. A classical low-level WF-net for case $f1$ discovered using the Inductive miner is shown in Fig. 11. In addition, Fig. 11 shows an enlarged part of the process highlighted in the final model to clearly demonstrate the level of detail. It is easy to see that the resulting model is absolutely inappropriate for visual analysis.

The code of each event in the *BPI Challenge 2015* event log consists of three parts: two digits, a variable number of characters, and three more digits.

Using the event log description, we know that the first two digits and the characters indicate the sub-process the event belongs to, which allows us to assume an option of identifying the sub-processes.

We used the first two parts of the event name to create the mapping between low-level events and sub-process names. After applying our hierarchical process discovery algorithm in combination with the In-

ductive Miner, we obtained a high-level model presented in Fig. 12 that is far more comprehensible than the classical model mainly because of its size.

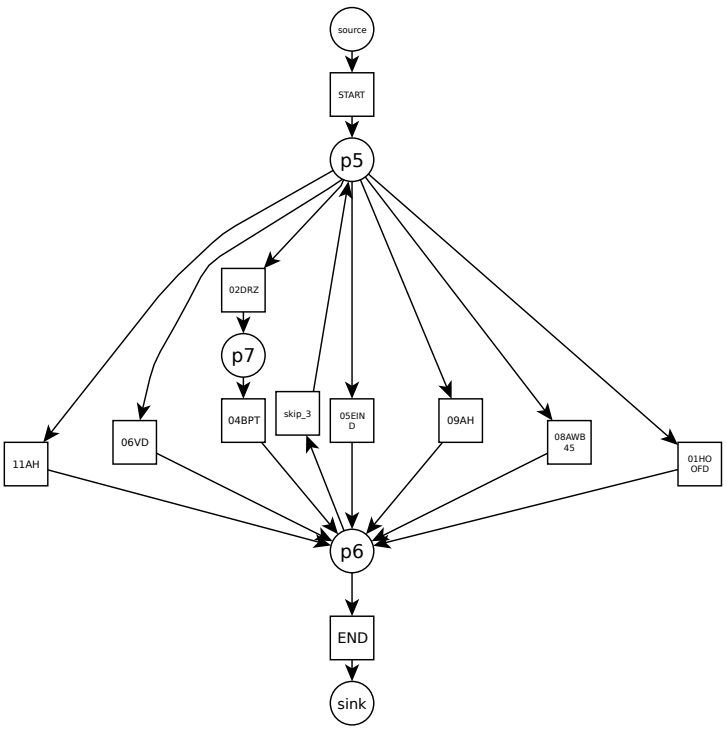


Fig. 12. A high-level WF-net discovered from the *BPI Challenge 2015* event log

The *BPI Challenge 2017* event log pertains to a loan application process of a Dutch financial institute. The data contains all applications filed trough an online system from 2016 till February of 2017. Here, as a base for mapping low-level events to sub-process names, we used the mark of the event type in its name — application, offer or workflow. Thus, a mapping could be based on various features of event data depending on the expert's needs. The classical model for these data is presented in Fig. 13, which is also difficult to read due to its purely sequential representation.

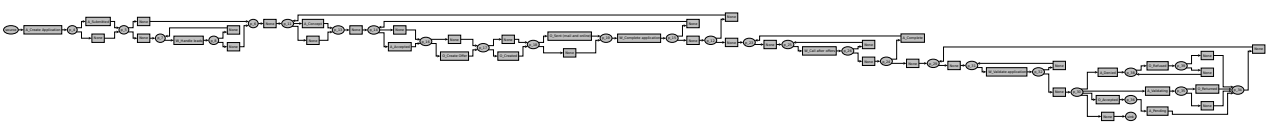


Fig. 13. A classical WF-net discovered from the *BPI Challenge 2017* event log

Applying the principle of mapping low-level events in the *BPI Challenge 2017* event log described above, we obtained the high-level WF-net shown in Fig. 14, which clearly demonstrates sub-processes (if necessary, they can be expanded) and their order.

Table 2 shows the fitness and precision evaluation of classical and high-level WF-nets discovered from real-life *BPI Challenge 2015* and *2017* event logs.

Fitness 1 shows the fitness evaluation between the classical WF-net constructed from the high-level WF-net by refining transitions with low-level sub-processes.

Fitness 2 shows the fitness evaluation between the high-level WF-net and an event log with low-level events grouped into sub-processes. This confirms the formal correctness results of the hierarchical pro-

cess discovery algorithm. Similar to the experimental results for artificial event logs, here we also observe a decrease in the precision for the identification of sub-processes, therefore, generalizing traces in an initial low-level event log.

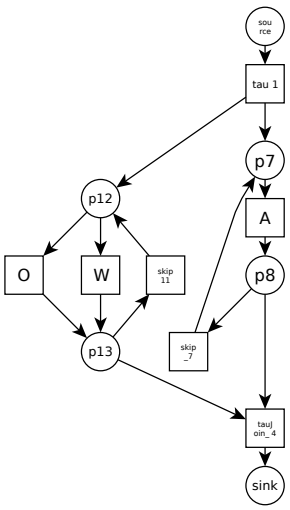


Fig. 14. A high-level WF-net discovered from the *BPI Challenge 2017* event log

Table 2. Comparing metrics for classical and high-level WF-nets discovered from *BPI Challenge* event logs

Event log	High-level WF-net			Classical WF-net	
	Fitness 1	Fitness 2	Precision	Fitness	Precision
BPI Challenge 2015	1	1	0.5835	1	0.5700
BPI Challenge 2017	1	1	0.3898	1	0.7000

6. Conclusion and Future Work

In this study, we propose a new process discovery technique for solving the problem of discovering a hierarchical WF-net model from a low-level event log, based on sub-processes abstraction into high-level transitions according to event partitioning. Unlike the previous solutions, we allow cycles and concurrency in process behavior.

We prove that the proposed technique makes it possible to obtain hierarchical models, which fit event logs perfectly. The technique was also evaluated in real and artificial event logs. Experiments show that fitness and precision of obtained hierarchical models are almost the same as for the standard "classical" case, while hierarchical models are much more compact, more readable and more visual.

To implement our algorithm and check it on real data we used Python and one of the most convenient instruments for process mining at the moment — the PM4Py [32]. The implementation is provided in the public GitHub repository [33].

In further research, we plan to develop and evaluate various event partitioning methods for automatic discovery of hierarchical models.

References

[1] A. Augusto *et al.*, "Automated discovery of process models from event logs: Review and benchmark", *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 4, pp. 686–705, 2018.

[2] W. van der Aalst, “Workflow verification: Finding control-flow errors using Petri-net-based techniques”, in *Business process management: models, techniques, and empirical studies*, Springer, 2002, pp. 161–183.

[3] A. K. Begicheva and I. A. Lomazova, “Discovering high-level process models from event logs”, *Modeling and Analysis of Information Systems*, vol. 24, no. 2, pp. 125–140, 2017.

[4] S. J. van Zelst, F. Mannhardt, M. de Leoni, and A. Koschmider, “Event abstraction in process mining: Literature review and taxonomy”, *Granular Computing*, vol. 6, no. 3, pp. 719–736, 2021.

[5] D. G. Maneschijn, R. H. Bemthuis, F. A. Bukhsh, and M.-E. Iacob, “A methodology for aligning process model abstraction levels and stakeholder needs”, in *Proceedings of the 24th International Conference on Enterprise Information Systems - Volume 1*, 2022, pp. 137–147.

[6] F. Mannhardt, M. de Leoni, H. Reijers, W. van der Aalst, and P. Toussaint, “From low-level events to activities – a pattern-based approach”, in *Business Process Management*, Springer, 2016, pp. 125–141.

[7] N. Tax, N. Sidorova, R. Haakma, and W. van der Aalst, “Event abstraction for process mining using supervised learning techniques”, in *Proceedings of SAI Intelligent Systems Conference 2016*, Springer, 2018, pp. 161–170.

[8] C.-Y. Li, S. J. van Zelst, and W. van der Aalst, “A framework for automated abstraction class detection for event abstraction”, in *Intelligent Systems Design and Applications*, Springer, 2023, pp. 126–136.

[9] G. van Houdt, M. de Leoni, N. Martin, and B. Depaire, “An empirical evaluation of unsupervised event log abstraction techniques in process mining”, *Information Systems*, vol. 121, p. 102 320, 2024.

[10] A. Rebmann, P. Pfeiffer, P. Fettke, and H. v. d. Aa, “Multi-perspective identification of event groups for event abstraction”, in *Process Mining Workshops*, Springer, 2023, pp. 31–43.

[11] 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 *Proceedings of the 2nd International Conference on Process Mining*, IEEE, 2020, pp. 137–144.

[12] A. Senderovich, A. Shleyfman, M. Weidlich, A. Gal, and A. Mandelbaum, “To aggregate or to eliminate? Optimal model simplification for improved process performance prediction”, *Information Systems*, vol. 78, pp. 96–111, 2018.

[13] S. Smirnov, H. Reijers, M. Weske, and T. Nugteren, “Business process model abstraction: A definition, catalog, and survey”, *Distributed and Parallel Databases*, vol. 30, pp. 63–99, 2012.

[14] C. W. Günther and W. M. Van Der Aalst, “Fuzzy mining–adaptive process simplification based on multi-perspective metrics”, in *International conference on business process management*, Springer, 2007, pp. 328–343.

[15] W. Reisig, *Understanding Petri nets: Modeling techniques, analysis methods, case studies*. Springer, 2013.

[16] K. Jensen and L. Kristensen, *Coloured Petri nets: modelling and validation of concurrent systems*. Springer, 2009.

[17] S. Leemans, D. Fahland, and W. van der Aalst, “Discovering block-structured process models from event logs – a constructive approach”, in *Application and Theory of Petri Nets and Concurrency*, Springer, 2013, pp. 311–329.

[18] G. Greco, A. Guzzo, and L. Pontieri, “Mining taxonomies of process models”, *Data & Knowledge Engineering*, vol. 67, no. 1, pp. 74–102, 2008.

[19] J. Li, R. Bose, and W. van der Aalst, “Mining context-dependent and interactive business process maps using execution patterns”, in *Business Process Management Workshops. BPM 2010*, Springer, 2010, pp. 109–121.

[20] X. Lu, A. Gal, and H. A. Reijers, “Discovering hierarchical processes using flexible activity trees for event abstraction”, in *Proceedings of the 2nd International Conference on Process Mining*, IEEE, 2020, pp. 145–152.

[21] W. van der Aalst and C. Gunther, "Finding structure in unstructured processes: The case for process mining", in *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, IEEE, 2007, pp. 3–12. DOI: 10.1109/ACSD.2007.50.

[22] J. De Smedt, J. De Weerdt, and J. Vanthienen, "Multi-paradigm process mining: Retrieving better models by combining rules and sequences", in *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, Springer, 2014, pp. 446–453. DOI: 10.1007/978-3-662-45563-0_26.

[23] J. de San Pedro and J. Cortadella, "Mining structured Petri nets for the visualization of process behavior", in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ACM, 2016, pp. 839–846. DOI: 10.1145/2851613.2851645.

[24] W. van der Aalst, A. Kalenkova, V. Rubin, and E. Verbeek, "Process discovery using localized events", in *Application and Theory of Petri Nets and Concurrency*, Springer, 2015, pp. 287–308. DOI: 10.1007/978-3-319-19488-2_15.

[25] A. Kalenkova and I. Lomazova, "Discovery of cancellation regions within process mining techniques", *Fundamenta Informaticae*, vol. 133, pp. 197–209, 2-3 2014. DOI: 10.3233/FI-2014-1071.

[26] A. Kalenkova, I. Lomazova, and W. van der Aalst, "Process model discovery: A method based on transition system decomposition", in *Application and Theory of Petri Nets and Concurrency*, Springer, 2014, pp. 71–90. DOI: 10.1007/978-3-319-07734-5_5.

[27] C.-Y. Li, S. J. van Zelst, and W. M. van der Aalst, "An activity instance based hierarchical framework for event abstraction", in *Proceedings of the 3rd International Conference on Process Mining*, 2021, pp. 160–167. DOI: 10.1109/ICPM53251.2021.9576868.

[28] T. Tapia-Flores, E. López-Mellado, A. P. Estrada-Vargas, and J.-J. Lesage, "Discovering Petri net models of discrete-event processes by computing t-invariants", *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 3, pp. 992–1003, 2017.

[29] T. Tapia-Flores and E. Lopez-Mellado, "Discovering workflow nets of concurrent iterative processes", *Acta Informatica*, vol. 61, Sep. 2023. DOI: 10.1007/s00236-023-00445-5.

[30] W. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Heidelberg, 2011.

[31] K. Lautenbach, "Linear algebraic techniques for place/transition nets", in *Petri Nets: Central Models and Their Properties. ACPN 1986*, Springer, Heidelberg, 1987, pp. 142–167.

[32] A. Berti, S. Van Zelst, and W. van der Aalst, "Process mining for Python (PM4Py): Bridging the gap between process- and data science", in *Proceedings of the ICPM Demo Track 2019*, CEUR-WS.org, 2019, pp. 13–16.

[33] A. Begicheva, *Hierarchical process model discovery – hldiscovery*, 2022. [Online]. Available: https://github.com/gingerabsurdity/hldiscovery.

[34] J. Carmona, B. van Dongen, A. Solti, and M. Weidlich, *Conformance Checking: Relating Processes and Models*. Springer, 2018.

[35] A. Adriansyah, B. F. van Dongen, and W. M. van der Aalst, "Conformance checking using cost-based fitness analysis", in *2011 ieee 15th international enterprise distributed object computing conference*, IEEE, 2011, pp. 55–64.

[36] J. Munoz-Gama and J. Carmona, "A fresh look at precision in process conformance", in *International Conference on Business Process Management*, Springer, 2010, pp. 211–226.

[37] I. Shugurov and A. Mitsyuk, "Generation of a set of event logs with noise", in *Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering*, 2014, pp. 88–95.

[38] A. Augusto *et al.*, *Data underlying the paper: Automated discovery of process models from event logs: Review and benchmark (version 1)*, Data set. 4TU.Centre for Research Data, 2019. DOI: 10.4121/uuid:adc42403-9a38-48dc-9f0a-a0a49bfb6371.