CDLG: Generation of Event Logs with Concept Drifts

Bachelor Thesis

presented by
Justus Matthias Grimm


submitted to the
Data and Web Science Group
Prof. Dr. Han van der Aa
University of Mannheim

January 2022

Abstract

Concept drift detection, a research discipline of process mining, focuses on detecting unexpected changes occurring in the execution of business processes through the analysis of recorded event data. Several real-life event logs with concept drifts are required for the evaluation of these techniques. Due to the limited number of companies providing their data for research purposes, it is difficult nowadays to access real-life event logs with its reference model. This bachelor thesis presents an approach for generating event logs suitable to be used for the evaluation of different process mining methods, mainly concept drift detection techniques. Special features of this application, called 'Concept Drift Log Generator' (CDLG), include the generation of the four concept drift scenarios (i.e. sudden-, gradual-, recurring- and incremental drift) with its required configurations in terms of change point and change localization (i.e. change in the control-flow). Furthermore, this framework provides the possibility to create several drifts and to introduce noise into an event log. Through an evaluation using the most suitable concept drift detection technique, called Visual Drift Detection (VDD), the efficiency of CDLG is demonstrated by verifying to what extent this method can detect the user-controlled drift configurations in the generated event logs. CDLG is implemented as an interactive Python application usable via the terminal or via text files containing the required parameters.

Keywords. Process mining · Concept drift · Event log

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Process mining as a growing research field aims at discovering, monitoring and improving business processes by extracting useful insights from event data, recorded in event logs [1]. An event log consists of a set of traces, whereby a trace represents a sequence of events generated by an execution of an operational process [1]. The execution of the business processes can change unexpectedly over a period of time, e.g. in the control-flow, resulting in the change of the process execution order [2]. These changes, so-called concept drifts, must be analysed using techniques from the research discipline of 'concept drift detection' established in process mining to enable the support and improvement of business processes and to gain important knowledge about their execution [2]. The concept drifts in event logs can be distinguished on the basis of three characteristics, namely the change point, change characterization and change localization [2]. Since concept drifts can differ in these features, different approaches have been and are being developed to detect them.

To assess the quality of concept drift methods, it is necessary to evaluate them using real-life event logs. The problem, however, is that companies rarely publish their data for research purposes and classify the execution information of their business process as corporate secrets, thus keeping the reference model private [3]. As a result, the number of useful real-life event logs with the corresponding reference model is limited and a reasonable evaluation of process mining algorithms becomes challenging. It is even more difficult to obtain event logs with concept drifts, where the change point, change localization and change characterization of the drift are known in advance, as this is essential for a proper evaluation. Consequently, synthetically generated event logs are required to satisfy the researchers' specifications. Several approaches for creating logs already exist, all of them having limitations regarding concept drifts. Hence, this calls for an approach capable of creating event logs with concept drift scenarios that work effectively for the

evaluation of concept drift techniques.

With this in mind, the research question of *how to implement the conceptual challenges of different drift scenarios to generate event logs with desired concept drifts* is targeted in this bachelor thesis. It is addressed with the Concept Drift Log Generator (CDLG), which is a new approach, elaborated and proposed in this thesis. The focus is first on identifying the challenges in generating a desired drift in an event log and subsequently on devising appropriate implementations. It is shown that these algorithms enable the generation of event logs with the four types of drift scenarios (i.e. sudden, gradual, recurrent, and incremental) while providing user-controlled drift configurations mainly on the change point and change localization (i.e. change in control-flow) of the drifts. Furthermore, noise and several drifts can be introduced into one event log. For the implementation with Python, packages of the open source process mining platform 'Process Mining for Python' (PM4Py) [4] are used.

In the core part of this thesis, the above mentioned implementations and three options to generate the desired event logs are presented. These options consist of a detailed run of the application via the terminal, a more rapid way via the text files containing the parameters and the simplest way via gold standard generation. Subsequently, an evaluation of this approach is conducted using the VDD tool [5]. The analysis shows that CDLG enables event logs to be generated with a wide variety of scenarios and that it is suitable for evaluating concept drift detection techniques.

The thesis at hand is structured as follows. Chapter 2 introduces background information and describes the challenges to be considered for event log generation with concept drifts. Chapter 3 explains the implementation solutions of CDLG for these challenges and an overall overview of the usability is given in Chapter 4. Afterwards, the approach is evaluated in Chapter 5. Last, CDLG is reviewed in light of related work in Chapter 6, before 7 draws a conclusion.

Chapter 2

Theoretical Framework

2.1 Background

This section discusses the basic concepts of process mining that are essential to CDLG's generation of event logs with drift scenarios. The process tree as shown below is created in this format by CDLG.

Process tree. A process model aims to visualize the execution dependencies of an organization's activities (i.e. the control-flow perspective of processes), whereby a process tree is a way of representing a sound process model [1]. Furthermore, these models support the basic control-flow patterns and can be easily modified [6], [7].

Process trees are directed connected graphs without loops, containing branch nodes (i.e. operators) and leaf nodes (i.e. activities) [6], as shown in Figure 2.1. Each branch node can have so-called children that are either activities or operators connected by edges [6], where the number of edges from the root to a specific node represents the depth of that node. Leaf nodes can contain, in addition to normal activities, silent activities that are not observed during execution [1].

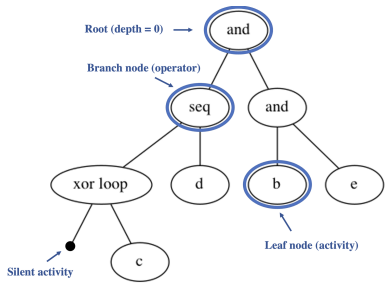


Figure 2.1: Process tree example.

The operators sequence ('seq'), exclusive choice ('xor'), parallel ('and'), loop ('xor loop') and OR ('or') can be found in the tree. In this context, the order of the child nodes only matters for the sequence and loop operators, since these execute the child nodes from left to right [6]. The process trees in CDLG adhere to the definition of the paper [7] published by Jouck and Depaire, where all operators can have multiple children. Only the loop operator must have at least two and at most three children.

Event log. CDLG uses process tree as a model for event log generation. Formally, an event log contains data from process execution consisting of a set of cases (i.e. traces), each of which has a sequence of ordered events, as shown in Table 2.1 [1]. These events refer to activities (e.g. 'register a request' or 'reject a request') executed during the process and can have attributes such as time and resource, among others [1]. CDLG generates event logs in the XES[1] standard, which is an advanced event log format used in the research discipline of process mining [8].

Case id	Activity	Timestamp	Duration
	b	2020-03-08 08:00:00	6879
	d	2020-03-08 09:54:39	10745
	e	2020-03-08 12:53:44	2717
1	e	2020-03-08 10:32:15	2734
	b	2020-03-08 11:17:49	6909
	c	2020-03-08 13:12:58	3365
	c	2020-03-08 14:09:03	3341
	d	2020-03-08 15:04:44	10816

Table 2.1: Fragment of an event log corresponding to Figure 2.1.

Researchers in this discipline have developed algorithms that use these event logs to extract useful knowledge about the process execution, such as discovering process models like the one depicted in Figure 2.1 [1].

Concept drift. Another information that event logs are analyzed for are concept drifts, which are defined as unexpected permanent changes in process execution over a period of time [2]. These drifts can differ in the main features called change point, change characterization and change localization [2], which are described with its respective challenges during event log generation in the following section.

[1] https://xes-standard.org

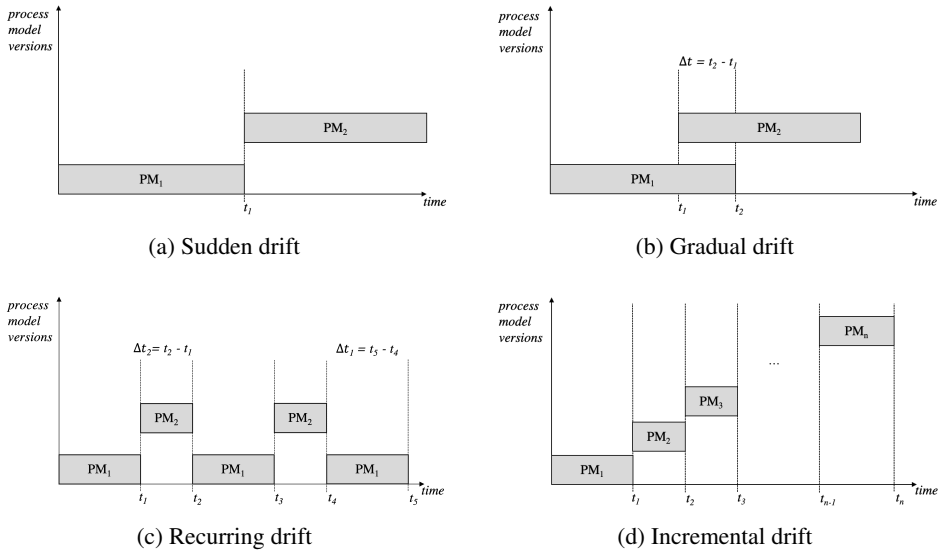


Figure 2.2: Concept drift types. Adapted from [2].

2.2 Drift Generation Challenges

In this section, the main challenges for generating suitable event logs with concept drifts are discussed. First, the main features are described, followed by an explanation of the challenges faced when generating the logs.

2.2.1 Change Point and Characterization

Change point refers to the point in time or the period of time at which the drift has taken place [2]. Change characterization goes one step forward and specifies the nature of change that has happened [2], whereby the paper [2] distinguishes between four concept drifts. These can be identified by their change point and their change dimension and are described below. Since the main challenge of CDLG is to create the different drift scenarios in event logs, the report explains the challenges of drifts by means of changing process model versions through which the synthetic process instances (traces) are generated.

Sudden Drift. Within this drift type, an existing process is replaced by a new process at an exact point in time [2]. For the creation of event logs, this means that the initial model version must abruptly change to a new version. This is illustrated in Figure 2.2a, in which the process model version PM_1 exists until time t_1 and is

then substituted by the evolved version PM_2. The challenge for creating a sudden drift in an event log is thus the abrupt change of two process model versions at a precise point in time.

Gradual Drift. During gradual drifts two versions of processes coexist for a certain period of time before the initial process is completely replaced by a new process [2]. This scenario is illustrated by two process model versions PM_1 and PM_2 in Figure 2.2b, which shows the trace generation period Δt of the versions coexisting. The evolved version PM_2 starts at t_1 and the initial model version PM_1 ends at t_2. In the time period Δt, the traces generated by the two versions of the process model can be distributed in such a way that, over time, an increasing number of traces from the evolved version and a decreasing amount of traces from the initial version appear in the event log until only traces from the evolved version remain. Theoretically, any mathematical distribution can be used for this purpose, e.g. a linear or exponential change in the two trace generation sources [9]. The main challenges to generate an event log with a gradual drift is therefore to precisely determine the start and end points of the drift and to distribute the traces in the period Δt in a reasonable way.

Recurring Drift. The recurring drift represents processes that occur again after a while [2]. Consequently, for the generation of event logs, recurrent seasonal changes of process model versions are required over a period of time. In Figure 2.2c, for instance, four seasonal interchanges of the process model versions PM_1 and PM_2 occur at time t_1, t_2, t_3 and t_4. Thereby, the execution period Δt_1 of PM_1 lasts longer than the period Δt_2 of PM_2. For these reasons, the generation of an event log with a recurring drift is challenged by the number of seasonal changes, the time period of the drift and the length of the seasonal execution periods of the model versions.

Incremental Drift. This drift corresponds to the replacement of a process by a new process through several smaller phased process changes [2]. Regarding event log generation, this means that several versions of a process model are replaced sequentially, differing in their evolutionary stage from the initial to the final version and in their execution periods. This is illustrated in Figure 2.2d, in which PM_1 is the initial model version and PM_n the final evolved version. Between the execution period of PM_1 and PM_n, the versions PM_2 to PM_{n-1} are successively used for the trace generation. These versions evolve one after the other increasingly towards the final version of the process model PM_n. Thus, the main challenge in creating an event log with an incremental drift is to determine the number of evolving versions of the process model and their respective execution periods. Likewise, the incremental evolution of the versions regarding change localization must be considered.

In summary, for a successful generation of event logs with the desired concept drifts, the various challenges of the four drift types, shown in Table 2.2, must be solved by suitable implementations.

Drift type	ID	Challenges
Sudden	S1	Abrupt change point of versions
Gradual	G1	Drift period
	G2	Distribution of traces
Recurring	R1	Drift period
	R2	Number of seasonal changes
	R3	Length of execution periods
Incremental	I1	Number of evolving versions
	I2	Length of execution periods
	I3	Incremental evolution of versions

Table 2.2: Challenges per drift types.

2.2.2 Change Localization

Concept drifts can occur in different perspectives during the execution of the processes. On the one hand, the control-flow of process execution can change and on the other hand, changes can occur in terms of resource, data and time [2]. In the scope of this thesis, only control-flow changes are addressed, as these changes have the most significant impact on the execution order of processes.

ID	Simple change pattern
CP1	Add/remove fragment
CP2	Make two fragments conditional/sequential
CP3	Make fragment loopable/non-loopable
CP4	Make two fragments parallel/ sequential
CP5	Make two fragment skippable/ non-skippable
CP6	Move fragment into/out of conditional branch
CP7	Synchronize two fragments
CP8	Duplicate fragment
CP9	Move fragment into/out of parallel branch
CP10	Substitute fragment
CP11	Swap two fragments
CP12	Change branching frequency

Table 2.3: Simple control-flow change patterns. Adapted from [10].

In the paper [10] published by Maaradji et. al., the control-flow change patterns proposed in [11] are summarised into twelve main process influencing patterns shown in Table 2.3, which are usually found in process models. These twelve

simple control-flow change patterns reflect the significant possible changes regarding localization during process execution (i.e. changes in the process model). For this reason, the main challenge for the generation of changes in the control-flow in event logs is to evolve a process model version in terms of the change patterns listed in Table 2.3 to a new version. In addition, since versions of a process model can differ in more than one simple change pattern, the challenge emerges to evolve a version multiple times to a new version.

2.2.3 Further Challenges

For a sound generation of event logs, the following further challenges need to be addressed.

Additional Drifts. The drifts explained in Section 2.2.1 can occur successively in an event log, as shown in Figure 2.3. In this example, a gradual drift taking place in the period Δt follows a sudden drift, which occurs at t_1. Therefore, the additional challenges emerge to determine the number of drifts per log and to address for each further drift the challenges explained in Section 2.2.1.

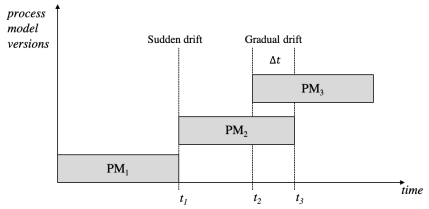


Figure 2.3: Successive drifts.

Noise. Noise often occurs in real-life event logs and is the result of data quality problems. It is manifested by the fact that irregular or incoherent process executions occur [12]. Two types of noise can occur in real-life event logs. First, noise can be completely chaotic, with no structure with respect to the current process model version. Second, it can have a similar structure to that of the underlying model versions [13]. Noise can be randomly present with a certain frequency in an event log. Thus, the generation of an event log with noise is challenged by the sector, the distribution, the type and the frequency of the noise.

Time Configuration. In an event log, traces can start at different times and each activity requires a certain duration to be completed, as shown for instance in Figure 2.1. Accordingly, the additional challenge exist of considering the starting point of the traces and the duration of the activities for generating event logs with a more realistic time configuration.

Chapter 3

CDLG Technical Details

The Concept Drift Log Generator proposed in this thesis provides the ability to generate event logs, with specific implementations addressing the challenges from the previous chapter. On the one hand, CDLG supports the creation of the four different drift types in event logs with user-controlled drift configurations in terms of the change point from either imported or randomly built process models. On the other hand, it supports full control concerning control-flow changes, by supporting a precise evolution of the process models. The random evolution of process models is ensured as well as introducing additional drifts and noise into the event log.

For the implementation in Python, packages of the open source process mining platform 'Process Mining for Python' (PM4Py) [4] are used. Since PM4Py supports the creation of event logs with process trees and enables random process tree generation, these are considered for the various implementation steps, as explained in the remainder of this chapter.

3.1 Change Point and Characterization

This section discusses by means of pseudo-codes how the challenges of change point and characterization (i.e. drift types) described in Section 2.2.1 for the drifts are implemented. In general, for the creation of the different drift scenarios, event logs with a certain number of traces are generated from the passed process tree versions and are subsequently merged depending on the drift type.

Sudden Drift. The challenge of the abrupt change is addressed by allowing the user to determine the exact time of the sudden drift. This change point is specified as a proportion of the total traces occurring in the event log. In Algorithm 1 below, the implementation for the sudden drift is reported. It expects as input the initial version, the evolved version, the change point and the number of traces to

be generated. The algorithm first calculates the number of traces required from each version of the process tree, depending on the change point t (lines 1 & 2), and then creates appropriate event logs for each model version (lines 3 & 4). The two generated logs log_1 and log_2 are then concatenated (line 5), resulting in the change point being exactly at the specified point t in the new returning event log.

Algorithm 1 Sudden drift.

Input: pt_1: Initial version of the process tree.
pt_2: Evolved version of the process tree.
n : Number of traces in the event log.
t : Change point as proportion of n.
Output: An event log with one sudden drift.

1: $n_1 \leftarrow \lceil t \cdot n \rfloor$ ▷ *see footnote*[2]
2: $n_2 \leftarrow n - n_1$
3: $log_1 \leftarrow generateLog(pt_1, n_1)$ ▷ *Generate log (PM4Py algorithm).*
4: $log_2 \leftarrow generateLog(pt_2, n_2)$
5: $result \leftarrow log_1 \cup log_2$ ▷ *Concatenate two logs.*
6: **return** result

Gradual Drift. To address the challenges of gradual drift, the start time, the end time of the drift and one of two trace distribution types can be specified.

Algorithm 2 Gradual drift.

Input: pt_1: Initial version of the process tree.
pt_2: Evolved version of the process tree.
n : Number of traces in the event log.
t_1 : Start change point as proportion of n.
t_2 : End change point as proportion of n.
d : Distribution type for the traces during the drift.
Output: An event log with one gradual drift.

1: $n_1 \leftarrow \lceil t_1 \cdot n \rfloor$ ▷ *see footnote*[2]
2: $n_2 \leftarrow (1 - t_2) \cdot n$
3: $n_3 \leftarrow n - n_1 - n_2$
4: $log_1 \leftarrow generateLog(pt_1, n_1)$
5: $log_2 \leftarrow generateLog(pt_2, n_2)$
6: $log_3 \leftarrow distributeTraces(pt_1, pt_2, d, n_3)$ ▷ *Algorithm 3 (d = 'linear').*
7: $result \leftarrow log_1 \cup log_3 \cup log_2$
8: **return** result

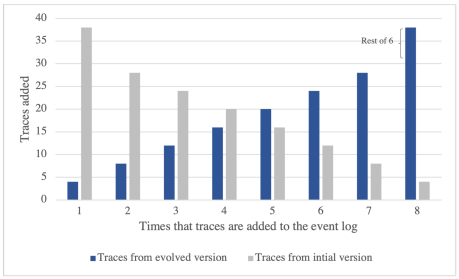
Algorithm 2 shows the required implementation, where the change point inputs t_1 and t_2 are a proportion of the set number of traces. The main part is the genera-

[2] $\lceil \rfloor$ means round up at 0.5 and round down at less than 0.5.

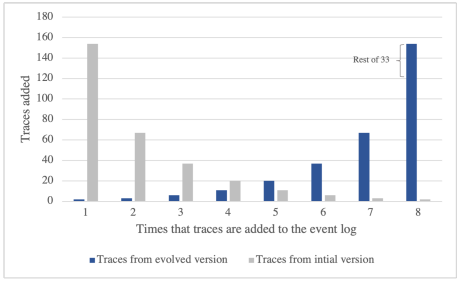
tion of three different logs with a specific number of traces (line 1-3) depending on t_1 and t_2. Log_1 contains the traces generated by the initial version for the part of the event log preceding the drift (line 4), log_2 contains the traces from the evolved version for the part after the drift (line 5), and log_3 contains the traces from both versions for the drift part (line 6). The drift part log_3 is generated by a specific algorithm (line 6) in which traces are generated from both the initial and the evolved process tree version and are merged with a certain distribution d to an event log. At the end, all logs are concatenated to one event log (line 7), ensuring that the start and end change points of the drift are accurate.

Two options are available for the distribution of the traces during the drift. First, the traces can be distributed linearly, meaning in this case that the traces of the initial version decrease and the traces of the new version increase continuously during the drift. In this instance, reference is made to the mathematical function $x \cdot b$, where the slope coefficient b varies between 2 and 5. Second, the more rapid exponential distribution exists, which follows an exponential rate of the trace increase and decrease. For this situation, the function $e^{x \cdot b}$ is considered, where the slope coefficient b ranges from 0.5 to 0.8.

The slope coefficient for the linear and exponential distribution is deliberately chosen in this range, as otherwise gradual drifts could occur far too quickly and might no longer be detected. In addition, the user is intentionally not allowed to determine the slope himself, as an algorithm automatically searches for the optimal coefficient, as explained below on the basis of Algorithm 3.



(a) Linear distribution $x \cdot 4$ of 300 traces.



(b) Exponential distribution $e^{x \cdot 0.6}$ of 600 traces.

Figure 3.1: Example distribution of traces during gradual drift.

These two types of distribution are demonstrated in Figures 3.1a and 3.1b, where the grey bars represent the added traces generated by the initial version and the blue bars are those created by the evolved version. The difference in the speed of the change from the initial version to the new version of the two options is shown in this example by the fact that the linear distribution distributed half as many traces

as the exponential distribution in the same time.

Algorithm 3 Linear distribution of traces.

Input: pt_1: Initial version of the process tree.
pt_2: Evolved version of the process tree.
n : Number of traces during the gradual drift.
Output: An event log only including the gradual drift part.

1: $result \leftarrow \emptyset$
2: $rest_1, rest_2, rounds, b \leftarrow calculateParameters(n)$ ▷ *see footnote*[3]
3: $x \leftarrow 1$
4: $most_1 \leftarrow rest_1 + rounds \cdot b$
5: $most_2 \leftarrow rest_2 + rounds \cdot b$
6: **while** $x \leq rounds$ **do**
7: **if** $x = rounds$ **then**
8: $log_2 \leftarrow generateLog(pt_2, most_2)$
9: **else**
10: $log_2 \leftarrow generateLog(pt_2, x \cdot b)$
11: **end if**
12: $result \leftarrow result \cup log_2$
13: **if** $x = 1$ **then**
14: $log_1 \leftarrow generateLog(pt_1, most_1)$
15: **else**
16: $log_1 \leftarrow generateLog(pt_1, (rounds - (x - 1)) \cdot b)$
17: **end if**
18: $result \leftarrow result \cup log_1$
19: $x \leftarrow x + 1$
20: **end while**
21: **return** result

In Algorithm 3 the implementation of the linear distribution is illustrated. In order to have as few inaccuracies ($rest_1$ & $rest_2$) as possible at the beginning and end of the drift the most suitable coefficient *b* ranging from 2 to 5 is automatically searched for by the algorithm in line 2. These remainders are sometimes unavoidable, as it must be ensured that the parameters set by the user are correct with regard to the duration of the drift. Nevertheless, these remainders do not affect the detection of the drift by process mining techniques, since an ascending natural change of the evolved model version is still guaranteed. Algorithm 3 generates an increasing number of $x \cdot b$ (line 10) traces from the evolved model version and a decreasing number of $(rounds - (x - 1)) \cdot b$ (line 16) traces from the initial version, which

[3]This function calculates the coefficient *b* ranging between 2 and 5, which brings the smallest remainder $rest_1$ and $rest_2$. It also calculates the number *rounds* of the trace sequence generations of one version.

are subsequently added to the resulting log per loop *x*. For the implementation of the exponential distribution only the multiplication with *b* is exchanged with $e^{x \cdot b}$ (line 4, 5, 10 & 15) with *b* allowed to range from 0.5 to 0.8.

In Figure 3.1a, the function in line 2 of Algorithm 3 has taken the coefficient 4, as this brings the smallest remainder of 6, which is additionally merged to the first generated set of traces by the initial version and to the last set of traces generated by the evolved version during the drift. For Figure 3.1b, 0.6 is chosen for the exponential distribution due to the same reason.

Algorithm 4 Simplified recurring drift.

Input: pt_1: Initial version of the process tree.
pt_2: Evolved version of the process tree.
n : Number of traces in the event log.
t_1 : Start change point as proportion of n.
t_2 : End change point as proportion of n.
s : Seasonal changes of the two versions (odd number).
p_1 : Proportion of the traces from the initial version during the drift.
Output: An event log with one recurring drift.

1: $log_1 \leftarrow generateLog(pt_1, \lceil n \cdot t_1 \rfloor)$
2: $log_2 \leftarrow generateLog(pt_2, \lceil n \cdot (1 - t_2) \rfloor)$
3: $log_3 \leftarrow generateLog(pt_1, \lceil n \cdot (t_2 - t_1) \cdot p_1 \rfloor)$
4: $log_4 \leftarrow generateLog(pt_2, \lceil n \cdot (t_2 - t_1) \cdot (1 - p_1) \rfloor)$
5: $s_2 = (s + 1)/2$
6: $s_1 = (s + 1) - s_2$
7: $logs_3 \leftarrow generatePartsLog(log_3, s_1)$
8: $logs_4 \leftarrow generatePartsLog(log_4, s_2)$
9: $result \leftarrow log_1 \cup logs_4[0] \cup logs_3[0]$
10: $i \leftarrow 1$
11: **while** $i < s_1$ **do**
12: $result \leftarrow result \cup logs_4[i] \cup logs_3[i]$
13: $i \leftarrow i + 1$
14: **end while**
15: **if** $s_2 \neq s_1$ **then**
16: $result \leftarrow result \cup logs_4[s_2 - 1]$
17: **end if**
18: $result \leftarrow result \cup log_2$
19: **return** result

Recurring Drift. The challenges to generate an event log with a desired recurring drift are addressed by allowing the user to specify the exact time period of the drift, the amount of seasonal changes of the model versions and the proportion of the total traces of the initial version during the drift. The corresponding implementation

is shown in Algorithm 4, which is presented in a slightly simplified version requiring the start and end change points to be greater than zero and less than one. The respective logs log_1 and log_2 represent the parts before and after the drift and their lengths depend on the change points t_1 and t_2 (lines 1 & 2). Thereafter, s_1 event logs are generated from the process tree version pt_1 and s_2 event logs from pt_2 for the drift part. The amount of traces occurring in these event logs depend on the change points and the proportion p_1 of the traces generated by the initial version during the drift period (line 3-8). s_1 and s_2 are the number of seasonal occurrences during the recurring drift of the two versions (line 5 & 6) and the generated logs are saved in the lists $logs_3$ and $logs_4$ (line 7 & 8). Each log in the respective list has the same number of traces and reflects a seasonal process period. Afterwards, all logs are successively concatenated, whereby log_1 occurs initially until t_1 and the logs from the lists $logs_3$ and $logs_4$ are alternately merged (line 9-17). At the end, log_2 is appended to the resulting log (line 18).

In the proposed approach, the conditions of the drift covering the entire log ($t_1 = 0, t_2 = 1$) or the start point being zero ($t_1 = 0$) are additionally considered, since in these cases the seasonal execution periods of the model versions must be specially arranged to ensure the accuracy of the parameters set by the user.

Algorithm 5 Framework of incremental drift.

Input: pt_0: Initial version of the process tree.
n_0 : Number of traces of the initial version of the process model.
n_1 : Number of traces of the final evolved version.
n_2 : Number of traces of the intermediate evolved versions.
m : Number of intermediate evolving model versions.

Output: An event log with one incremental drift.

1: $result \leftarrow generateLog(pt_0, n_0)$
2: $trees \leftarrow [pt_0]$
3: $i \leftarrow 1$
4: **while** $i < m$ **do**
5: $pt_i \leftarrow evolveTree(trees[i-1])$ ▷ *Section 3.2*
6: $trees \leftarrow trees \cup pt_i$
7: $log_i \leftarrow generateLog(pt_i, n_1)$
8: $result \leftarrow result \cup log_i$
9: $i \leftarrow i + 1$
10: **end while**
11: $pt_2 \leftarrow evolveTree(trees[i-1])$
12: $log_2 \leftarrow generateLog(pt_2, n_2)$
13: $result \leftarrow result \cup log$
14: **return** $result$

Incremental Drift. By determining the traces for each particular process model version and allowing multiple versions of the model to be evolved, the challenges of this drift are addressed. Algorithm 5 shows the framework for the incremental drift, which is used in a similar way for all possible incremental drift generations. First, the part of the desired event log occurring before the drift is generated with n_0 traces by the tree version pt_0 (line 1). Subsequently, the initial version is incrementally evolved m times, each time generating n_2 traces of each version and adding those to the resulting event log (line 4-10). Each evolution in line 5 is performed on the last version, all of which are stored in the list $trees$ (line 6). In the end, the final version is evolved using the last model version in $trees$ and n_1 traces are generated and appended to the resulting event log (line 11-13). If event logs are generated via the terminal (details later in Section 4.1), an individual number of traces for each evolved version can be determined.

In summary, Table 3.1 gives an overview of the solving implementations for the challenges from Section 3.2. The comparison of the tables shows that CDLG addresses all IDs from Table 2.2 and enables the generation of the required drifts in event logs by user controlled parameters.

Drift type	**ID**	**Parameters**
Sudden	S1	Change point as a proportion of total occurring traces
Gradual	G1	Start and end point as a proportion of total occurring traces
	G2	Linear or exponential distribution
Recurring	R1	Start and end point as a proportion of total occurring traces
	R2	Number of seasonal changes
	R3	Proportion of traces of the initial version during the drift
Incremental	I1	Number of evolving versions
	I2	Number traces for each version
	I3	Evolution of each version

Table 3.1: Solving parameters for challenges of Table 2.2.

3.2 Change Localization

This section explains the solutions to the change localization challenges described in Section 2.2.2. The process tree version can be evolved regarding control-flow in a controlled or random manner, both of which refer to the change localization types in Table 3.2.

Location	ID	Type
Activity	A1	Adding an activity
	A2	Deleting an activity
	A3	Replacing an activity
	A4	Swapping two activities
Operator	O1	Replacing an operator
	O2	Swapping two operators
Tree fragment	F1	Adding a tree fragment
	F2	Deleting a tree fragment
	F3	Swapping two tree fragments
	F4	Moving a tree fragment

Table 3.2: Implementations for process tree changes.

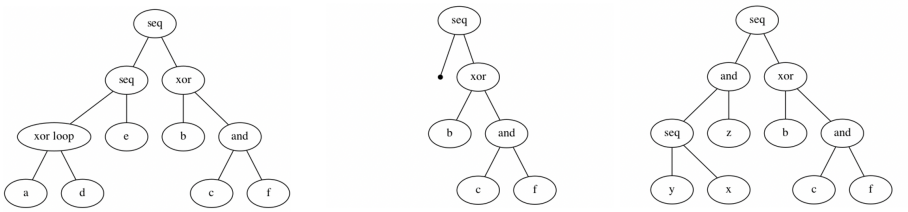
Controlled Evolution. By a controlled evolution of a process tree version, all listed change patterns of Table 2.3 except *CP12* are achievable. Table 3.3 lists the simple control-flow change pattern and an example of an efficient way with the implementations of Table 3.2 that can enable these pattern in this proposed approach. CP12 cannot be accomplished, as this would require a modification of the packages from PM4Py. Moreover, all listed change types in Table 3.2 can be used for a single evolution of a process tree version, as often as required, resulting in the capability to change the tree as desired.

The algorithms for controlled activity evolution (Table 3.2 activity types except *A1*) only require the activities to be changed as an input, if these are unique in the version of the tree. Otherwise, the parent nodes depth of the activity and its operator type are needed to clearly identify the activity in the process tree and to evolve it. The search algorithm proceeds in such a way that the process tree is split into all possible sub-trees at the given depth and searches for the specified operator type in the first node of the sub-trees. If several identical operators exist in this depth, the algorithm additionally requires an unique activity contained in one of the operator's descendants to find the parent node with certainty (example in Section 4.1). When adding a new activity (*A1*), the depth of the desired parent node and its operator type are always needed. In the case of *A2*, the activity to be deleted is replaced by a silent activity. The algorithms for *A3* and *A4* simply change the names of the leaf nodes.

This search algorithm described above is also used to precisely identify tree fragments and operators for the other evolution types (Table 3.2 operator and tree fragment types). For *O1* and *O2* an algorithm modifies the labels of the operator nodes to the required operator types. A new activity and a new operator must be specified for *F1's* algorithm in order to be added at the desired position in the

tree. No larger tree fragments can be added at once, as this would otherwise become too complex for sensible use. Instead, the desired whole tree fragment can be added in several rounds. The algorithm for *F2* replaces the identified tree fragment with a silent node. The same procedure is used in algorithm *F4*, except that the tree fragment is moved to another node and is not deleted. When swapping two tree fragments (*F3*), these fragments are not permitted to contain the same activities. This case would be unfeasible, since one tree fragment lies in the other tree fragment and are not interchangeable due to the tree structure. As unintentional silent activities may be present in the tree due to the deletion or movement of tree fragments, an extra algorithm can be used to remove such activities.

Figure 3.2 shows as an example the results of running the algorithms to achieve *CP10* from Table 3.3. First the tree fragment to be replaced at depth 1 with the operator '*seq*' is deleted (*F2*), as depicted in Figure 3.2b. Subsequently, the final version shown in Figure 3.2c is created by adding new fragments twice (*F1*) and replacing the silent activity with a new one (*A3*) at the end. The whole evolution could also have been achieved in this instance by replacing operators (*O1*) and activities (*A3*) in several rounds.



(a) Initial version. (b) *F2*. (c) 2 x *F1* & *A3*.

Figure 3.2: Example for a controlled evolution of a process tree version.

If the user enters wrong parameters during controlled evolution, the tree either fails to change or changes incorrectly. Similarly, if there are identical activities in the process tree, same operator types in the same depth and the user does not specify a unique activity, the search algorithm may identify the wrong activity, the wrong operator or the wrong tree fragment. An exact determination of an operator or a tree fragment would not be achievable in any other useful way, because the algorithms of PM4Py store child nodes of tree fragments not as sorted as shown in the illustrated trees.

ID	Simple change pattern	Implemented by
CP1	Add/remove fragment	A1/ A2/ F1/ F2
CP2	Make two fragments conditional/sequential	O1
CP3	Make fragment loopable/non-loopable	O1
CP4	Make two fragments parallel/ sequential	O1
CP5	Make two fragment skippable/ non-skippable	O1
CP6	Move fragment into/out of conditional branch	F4
CP7	Synchronize two fragments	F4, O1, A4
CP8	Duplicate fragment	F1
CP9	Move fragment into/out of parallel branch	F4
CP10	Substitute fragment	F2, F1
CP11	Swap two fragments	F3
CP12	Change branching frequency	x

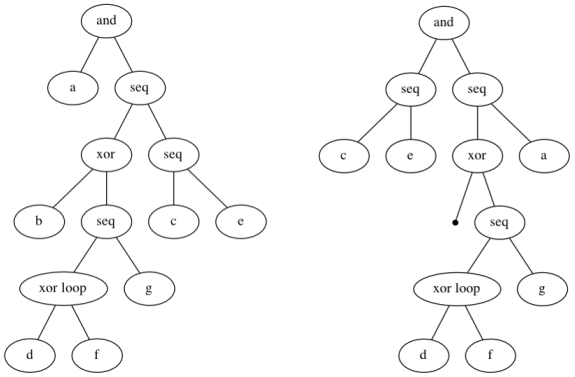
Table 3.3: Implementation examples of simple control-flow change patterns. Adapted from [10].

Random Evolution. In the case of random evolution, the proportion of the tree's activities that are to be influenced by the evolution is required as input. The algorithm randomly decides which implementation from Table 3.2 is used and how many activities determined by the user are to be affected with an evolving run, whereby silent activities are not counted as real activities. Affected activities include all activities that have been changed either directly or indirectly in the execution order. For instance, if an operator is replaced, all leaves (i.e. activities) under the operator node are counted as influenced activities. Furthermore, A3 of Table 3.2 is seen as affecting two activities because it is a shortened version of the sequence of A2 and A1.

Special adapted algorithms for the types of Table 3.2 are implemented, which divide the process tree into all possible sub-trees, where the changes are executed randomly. Before the change type is executed with the random number of activities, the algorithm checks whether this change is possible with the specific number of activities, whether the change adheres to the process tree structure (e.g. whether the change results in creation of empty traces) and whether an activity in the tree to be changed has not already been modified. If it is unfeasible, the algorithm randomly determines a different number of activities and a different change type from Table 3.2 and tries it again. If the change in the process tree is successful, the number of activities still to be affected is reduced by the amount of the modified activities. The algorithm repeats the process until the desired number of activities are affected by the changes.

Figure 3.3 shows a randomly evolved process tree, where the proportion of the set of activities to be affected is specified to 0.5, meaning in this example that

from seven activities four are affected by the evolution, as the algorithm rounds half up. In this case, the algorithm randomly chooses to swap the activity '*a*' with a tree fragment in depth 2, in which the parent node has the operator type '*seq*' and children activities '*c*' and '*e*' (*F3*). With this evolution run, three activities are influenced, which is why the algorithm performs another one with the modification of one activity. In this case, activity '*b*' is deleted (*A2*), resulting in exactly four activities being affected by this evolution.



(a) Initial version. (b) Randomly used *F3* & *A2*.

Figure 3.3: Example for a random evolution of a process tree version.

To sum up, CDLG solves not only the challenges from Section 4.2 (except for CP12), but additionally enables the possibility of random evolution of process trees. This saves a lot of effort and time during event log generation when no specific change is required in the control-flow.

3.3 Further Technical Details

In this section, the solution implementations for the additional challenges described in Section 2.2.3 are explained.

3.3.1 Additional Drifts

After generating the first drift with the algorithms from Section 3.1 and 3.2, more of the four concept drift scenarios can be introduced into the event logs when the guided path via the terminal is used (details later in Section 4.1). Initially, the algorithms from Section 3.2 are used to evolve the original or final evolved version of the process tree. This new version of the process tree is subsequently used

to generate traces for the additional drift, which can either be added at the end or inserted into the event log. If the drift is placed at the end of the event log, the same algorithms from Section 3.1 are used to generate the drift accordingly. However, if the drift is set in the event log, modified algorithms are used and the exact change points of the concept scenario in the event log can be specified. The algorithms then replace the traces in the existing event log with the newly generated ones using the updated version of the process tree.

The challenges of additional drifts are addressed, as the user can include as many drifts as desired in an event log. Likewise, complete control is given over the placement and configurations of the additional drifts and thus even drifts can be superimposed with correct procedure.

3.3.2 Noise

To produce more realistic synthetic generated data, a noise generation algorithm is introduced in this approach. On the one hand, the algorithm can introduce complete random noise by building a new process tree and generating traces from it that entirely differ from the actual ones. On the other hand, a certain proportion of the initial version of the process tree can be randomly modified, using the random evolution algorithm from Section 3.2. This model version is then used for generating the noise traces, which resemble the real ones closely. With these two types of noise, the user has the option of introducing simple or hard-to-identify noise into a desired event log.

Furthermore, the algorithm requires the start and end points as well as the frequency of the noise as input. To distribute the noise into the set sector, the algorithm generates the required number of noise traces and replaces actual traces with them in the event log. The procedure is such that the number of actual traces for which one noise trace needs to occur in order to guarantee a balanced distribution is calculated, and a defined number of sub-sectors with this quantity of traces is formed in the set noise sector. Afterwards, the algorithm replaces one randomly chosen real trace in these areas with a noise trace, as depicted in Figure 3.4. This illustrates a small noise sector with three sub-sectors, each containing a noise trace. To ensure that the frequency of the distributed noise is correct, the algorithm may sometimes insert two noise traces in one sector. Likewise, no more than 50% of noise can be introduced into the whole noise sector, as the noise would otherwise outweigh the actual traces and thus no longer be noise.

By specifying the type, sector, and frequency of noise, CDLG addresses the challenges identified in Section 2.2.3. The user has the possibility to distribute his required amount of noise in his set area, allowing him to disguise or even fake drifts.

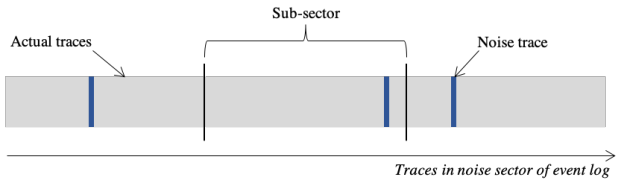


Figure 3.4: Random distribution of noise in the set sector of the event log.

3.3.3 Time Configuration

In the proposed approach, the start time of the first trace needs to be specified, as well as the minimum and maximum duration in seconds of the activities appearing in the process model. The algorithm for time configuration processes all traces and activities in the event log as soon as all actions (i.e. drifts and noise) have been completed. First, it sets the start time of the first activity in the first trace to the user's set timestamp and assigns a random duration between the minimum and maximum value to each activity in the trace, adjusting the timestamps of these activities accordingly. Afterwards, the start timestamp of each subsequent trace is modified by adding a random value to the start timestamp of the previous trace. This value ranges from the set minimum duration of an activity to the total duration of the last trace. In this way, the traces can start consecutively and run in parallel for different periods. As soon as a certain activity has been assigned a particular duration and this activity subsequently recurs in a trace, the same duration is reassigned to it, except that it can deviate by 1%. This is due to the fact that in reality, process activities are not executed for exactly the same duration every time.

Although the user does not have the ability to specify the start time for each trace and the duration for each activity, the challenges from Section 2.2.3 are tackled as CDLG provides a simple way by only requiring the start time of the event log and the range for the duration of the activities.

Chapter 4

CDLG Usability

This chapter explains the procedures of the three existing options to generate the desired event logs through CDLGs' implementations defined in the previous chapter. Firstly, the user can opt for the detailed run of the application via the terminal, in which all parameters are requested consecutively with an explanation (Section 4.1). In this way, all previously described implementations of the presented tool can be accessed. Secondly, the user can choose a more rapid way via the text files, in which the required parameters are stored (Section 4.2). With this option, however, controlled evolution and the insertion of additional drifts are not available, as these processes are complex and interactive. Lastly, it can be decided to use the swiftest way via gold standard generation (Section 4.3). This option, however, brings the most limitations regarding controlled configuration.

CDLG provides the user with the ability to import either process trees, Petri nets or BPMN models as process models. If the user intends to import his own BPMN models or Petri nets, the models must be block-structured, otherwise a conversion to a process tree with the used approach [14] published by van Zelst and Leemans is not feasible. If initial tree versions are used with silent activities, i.e. activities that generate empty events, it might happen that empty traces are generated in case of inappropriate tree structures.

4.1 Generation via Terminal

The event log generation via terminal can run in several ways depending on the set parameters, which is why the following instructions explain the process of this option using mostly one comprehensive path with illustrations of the terminal.

In general, the following information should be considered during use. Before each input there is a brief explanation of its purpose, and the brackets at the end

indicate the required input. A distinction is made between two types of brackets. Square brackets list all possible exact inputs, and round brackets indicate the type of input. Moreover, default values are provided for most parameters, which can be accessed with the Enter key and in most cases, the algorithm detects an incorrect input and allows the user to repeat his input instead of generating a system abort.

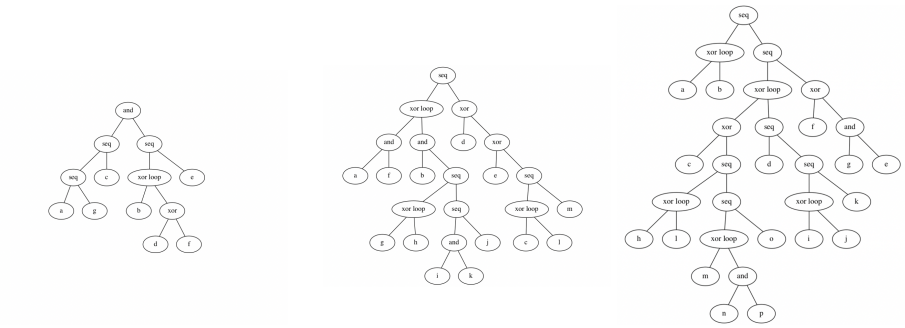
General Input. Figure 4.1 illustrates the process in the terminal for the general input. At the beginning, general information to be noted by the user is printed (lines 1-8), explaining the options for generating an event log.

```
1  --- GENERAL INPUT ---
2  The tool offers different paths to generate event logs with drifts.
3        - import models: one or two own models can be imported,
4          whereby the BPMN models and Petri nets need to be block-structured.
5        - random generated models: one or two process trees can be generated randomly.
6  When using one model, it can evolve randomly or in a controlled manner.
7  When using two models, the second model is/should be the already evolved version of the first model.
8  Several event logs can be generated with one run of the application, and the parameters can be changed for each event log.
9
10 Is an event log with a concept drift scenario required? [yes, no]: yes
11 Import models or use randomly generated models [import, random]: random
12 Evolution of one randomly generated model or use of two randomly generated models [one_model, two_models]: one_model
13 Do you want to adjust the various settings/parameters for the process tree, which will be used to generate the model randomly [yes, no]? no
14 Complexity of the process tree to be generated [simple, middle, complex]: middle
15 Number of event logs to be generated (int): 1
16
17 Random generated model is made visible as a process tree. Please do not close it, as it is needed for further actions.
18 Random generated model 'random_initial_model' is saved in the folder 'Data/result_data/generated_tress'.
```

Figure 4.1: General terminal input when using a random model.

The user's first input is the decision to create an event log with or without concept drifts (line 10). This leaves the possibility to create event logs without drifts and with noise only. Afterwards, the way to obtain the process model can be selected (line 11), whereby the creation of a random tree is chosen in the example from Figure 4.1. In both ways, the user can decide whether he wants to generate or import one or two models (line 12). If two models are chosen, the user cannot evolve a model with the controlled or random evolution, as the second model already represents the final evolved version of the process model. With random process tree generation, the user can opt to adjust the 15 parameters of the approach 'PTandLogGenerator' [7] that are responsible for the random creation (line 13). If it is decided not do so, as in the example in Figure 4.1, the complexity of the tree to be built can be selected (line 14). For the different complexities, the parameters of the approach [7] are adjusted accordingly. Thus, the higher the complexity, the more activities and complicated operators are generated in the process tree by the algorithm. Example process trees of the three complexity levels are illustrated in the Figure 4.2. When importing process models, the file path to the models must be specified instead of the parameters for tree generation (lines 13 & 14). Lastly, the number of event logs to be generated from the initial model version can be determined (line 15). Thereby, the user can set the parameters explained below for

each event log separately. Once the user has generated a random process tree, it is displayed as an image and saved (lines 17 & 18). The image should not be closed if controlled evolution is desired, as the process tree must be used as an orientation for this purpose.



(a) Simple. (b) Middle. (c) Complex.

Figure 4.2: Random generated process trees with different complexity.

Time Input. In Figure 4.3, the inputs addressing the timing challenges for each individual event log (line 1) are presented. The user can determine the start date of the first trace (line 2), which must follow the format *year/day/month hours:minutes:seconds*, like 20/8/3 8:0:0 for the date of 08 March 2020 and the time of 8 a.m.. Furthermore, the range for the duration of the activities in seconds (lines 3 & 4) need to be specified, as in the example of Figure 4.3, in which the activities of the event log last between one and five hours.

```
1 --- INPUT 1. EVENT LOG ---
2 Starting date of the first trace in the event log (y/d/m H:M:S like '20/23/8 8:0:0'): 20/8/3 8:0:0
3 Minimum for the duration of the activities in the event log in seconds (int): 3600
4 Maximum for the duration of the activities in the event log in seconds (int): 18000
```

Figure 4.3: Input for time configurations.

Drift Input. For the concept drift scenarios, different algorithms exist requiring various inputs, as explained in Section 3.1 and shown in Figure 4.4 and Figure 4.5. If the user selects sudden, gradual or recurring as drift type (line 1), the desired number of occurring traces in the event log can be specified (line 2). Thereby, the number must be greater than or equal to 100 in order to ensure that a reasonable event log with concept drift is generated.

For *sudden drift*, the only input required is the point in time of the drift as a proportion of the total number of traces in the event log, as shown in line 3 of Figure 4.4a. Thus, if the drift is to appear after 4,500 traces and the event log contains a total of 10,000 traces, the parameter must be set to 0.45.

During the input for *gradual drift* shown in Figure 4.4b, the start time and the end time of the drift must likewise be entered as a proportion (lines 3 & 4). Moreover, the type of distribution (i.e. linear or exponential) of the traces must be chosen (line 5), whereby the change to the new process happens significantly quicker with the exponential distribution than with the linear one.

```
1 Type of concept drift [sudden, gradual, recurring, incremental]: sudden
2 Number of traces in the event log (x >= 100): 10000
3 Starting point of the drift (0 < x < 1): 0.45
```

(a) Sudden drift.

```
1 Type of concept drift [sudden, gradual, recurring, incremental]: gradual
2 Number of traces in the event log (x >= 100): 10000
3 Starting point of the drift (0 < x < 1): 0.3
4 Ending point of the drift (0 < x < 1): 0.7
5 Method for distributing the traces during the gradual drift [linear, exponential]: linear
```

(b) Gradual drift.

```
1 Type of concept drift [sudden, gradual, recurring, incremental]: recurring
2 Number of traces in the event log (x >= 100): 10000
3 Do you want the recurring drift to persist throughout the event log [yes, no]? no
4 Starting point of the drift (0 < x < 1): 0.2
5 Ending point of the drift (0 < x < 1): 0.8
6 Number of seasonal changes (boundary changes excluded) of the versions, which needs to be an odd number (int): 5
7 Proportion of the first model version in the drift sector of the log (0 < x < 1): 0.4
```

(c) Recurring drift.

Figure 4.4: Inputs for the drift scenarios 'sudden', 'gradual' and 'recurring'.

Since a *recurring drift* often extends over the entire event log, it must first be specified whether this is intentional, as shown in Figure 4.4c in line 3. If the recurring drift is intended to be in a sector in the event log, the start and end time need to be specified (lines 4 & 5). Thereafter, it is required that the number of seasonal changes between the two model versions for the process execution is determined (line 6). The changes of the versions at the start and end points of the drift are not considered and if the recurring drift occurs in sectors, only an odd number can be given for the seasonal changes, since otherwise a change is lost. Therefore, if a total of six seasons are required between the start time and the end time, five seasonal changes must be entered. At the end, the user has to set the proportion of the total traces generated by the initial model version during the drift (line 7). This allows the seasonal sections generated by the two versions to be of different lengths. Thus,

each season generated by the initial model version from the example in Figure 4.4c has 800 traces and each season generated by the evolved version has 1200 traces.

During *incremental drift*, the user does not have to specify the drift duration as proportions, but determines the desired number of traces for each evolved model version. This allows a more precise and easier handling. Figure 4.5 shows the input of the incremental drift with the random evolution of the versions, which is explained in more detail below. First, the desired number of model versions to be evolved must be determined (line 2). Subsequently, the user can specify the number of traces for the initial version (line 3) and then determine the traces for the model version after each respective evolution (lines 10 & 16), which is done successively. This process takes place in a similar way in controlled evolution and in the use of imported models. During controlled evolution, the user changes the versions one by one according to his requirements, as explained below. When using two imported models, the new models must be imported one after the other and the respective number of traces must be specified. It should be noted that the model imported second at the beginning is used as the final model version and the imported models during the incremental drift are the intermediate evolving model versions.

```
1  Type of concept drift [sudden, gradual, recurring, incremental]: incremental
2  Number of evolving model versions (int): 2
3  Number of traces from initial model version in the log (x >= 100): 4000
4  Controlled or random evolution of the process tree version [controlled, random]: random
5  Proportion of the activities in the 1. evolving version to be affected by random evolution (0 < x < 1): 0.3
6
7  Control-flow changes: operator replaced;
8  Activities affected by the changes:[c, d, f, g]
9
10 Number of traces from 1. evolved model version in the log (int): 2000
11 Proportion of the activities in the 2. evolving version to be affected by random evolution (0 < x < 1): 0.3
12
13 Control-flow changes: activities swapped; activity added; activity deleted;
14 Activities affected by the changes:[c, h, Random activity 1, b]
15
16 Number of traces from 2. evolved model version in the log (int): 4000
```

Figure 4.5: Input for incremental drift with random evolution.

Evolution Input. Two ways of changing the process tree exist (line 4). First, the tree can be evolved randomly, as shown in Figure 4.5. Second, it can be evolved in a controlled way, as shown in Figure 4.6.

For *random evolution*, the user only needs to specify the proportion of the process model activities to be affected by the changes, as shown in line 5 and 12 of Figure 4.5. Afterwards, the process tree changes automatically and the new version is displayed to the user as an image. In addition, the random changes of the tree are output in the terminal, where the first line (lines 7 & 13) reports the types of

changes and the second line (lines 8 & 14) reports the affected activities.

```
1  Controlled or random evolution of the process tree version [controlled, random]: controlled
2
3  --- INFORMATION FOR THE EVOLUTION ---
4  The following main operators are available for the evolution of the process tree versions:
5      - change_activity: all procedures to change activities [add_ac, swap_acs, delete_ac, replace_ac]
6      - change_operator: all procedures to change operators [replace_op, swap_ops]
7      - change_tree_fragment: all procedures to change a tree fragment [add_fragment, delete_fragment, swap_fragments, move_fragment]
8      - delete_silent_ac: delete an undesired silent activity
9  The first process tree version opened as a picture provides orientation for evolution.
10 To achieve the desired version, multiple evolutions may need to be performed.
11 Please note that if the input is not correct, the tree version will not change (depth starts with 0 at the parent node).
12 Moreover, a node with a 'xor loop' operator can only have at least two and at most three children.
13 The following activities are contained in the tree: [d, a, c, b, e, h, f, g]
14
15 Operator for the 1. intermediate evolution of the process tree version
16 [change_activity, change_operator, change_tree_fragment, delete_silent_ac]: change_operator
17 Procedure for changing operator/s in the process tree version [replace_op, swap_ops]: replace_op
18 Existing operator to be replaced [xor, seq, or, xor loop, and]: xor loop
19 Depth of the node with the operator to be changed (int): 2
20 One activity present in the changing subtree [a, c, b, e]: b
21 New operator [xor, seq, or, xor loop, and]: and
22 Do you want to continue with the evolution of the version [yes, no]: no
```

Figure 4.6: Input for controlled evolution (O1).

In the *controlled evolution* of the process tree, depicted in Figure 4.6, general information on changing the initial version (lines 3-12) is first displayed. The 11 options for changing the process tree (lines 4-8) are explained. They are the same operators as in Section 3.2, only with specified names, described in Table 4.1. Moreover, it should be noted that the 'xor loop' operator may only have a minimum of two and a maximum of three children.

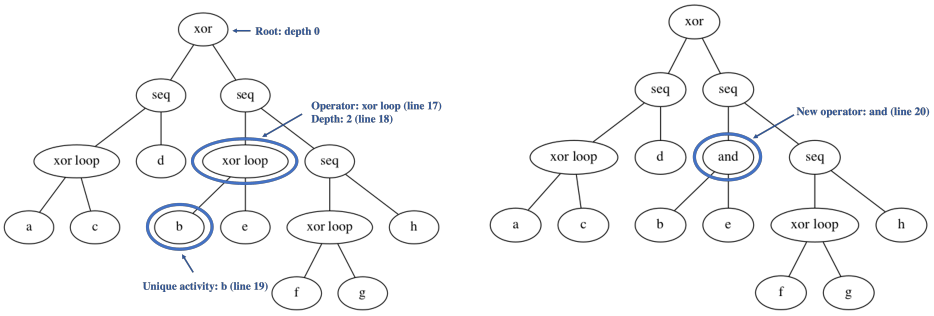
Location	ID	Name in CDLG
change_activity	A1	add_ac
	A2	delete_ac
	A3	replace_ac
	A4	swap_acs
change_operator	O1	replace_op
	O2	swap_ops
change_tree_ fragment	F1	add_fragment
	F2	delete_fragment
	F3	swap_fragments
	F4	move_fragment
delete_silent_ac	N/A	N/A

Table 4.1: Operator names in CDLG for the implementations of Table 3.2.

Since the inputs for the different evolution operators (Table 4.1) are quite similar, only the example illustrated in Figure 4.6 and Figure 4.7 is explained here.

First, the region (i.e. activity, operator, tree fragment or silent activity), in which the process tree is to be changed, must be determined (lines 14 & 15). In this illustrative example, an operator in the tree is modified (line 15), which is why a decision between O1 and O2 has to be made (line 16). For the other main change types, the respective configurations are requested instead (Table 4.1).

Subsequently, various parameters of the process tree (Figure 4.7) are required in order to change the tree as desired (lines 17-20). First, the user must specify the name of the operator to be replaced (line 17) and then its depth in the process tree (line 18). The first first node of the process tree starts with a depth of zero. If several operators of the same type exist in the same depth, a unique activity (line 19) located in the sub-tree of the operator to be changed must also be specified for a clear identification. Afterwards, only the new operator type is needed (line 20). At the end, it must be indicated whether the evolution of the process tree version has been completed (line 21). If this is not the case, it is possible to choose again between all change operators (Table 4.1) and continue with the modification of the version.



(a) Initial version. (b) Final version.

Figure 4.7: Evolving tree version from Figure 4.6.

Additional Drift Input. After the tree evolution is complete, an additional drift can be generated in the event log if desired, as shown in line 1 in Figure 4.8. First, general information for the procedure is described again (lines 3-9). Next, it can be decided which drift scenario should be added to the event log (line 11) and the user can specify the process model version to be evolved (line 12). Thereby he can decide between the initial version or the last evolved version. As described above, the selected tree version must then be evolved either randomly or in a controlled manner (lines 13-17). When using two imported or randomly generated model versions, the user can decide whether to import new models, have them randomly

generated or modify the original ones. Thereafter, the region of the drift can be decided (line 19 & 20) and the respective drift typical configurations can be determined. If it is selected to place the drift in the event log, the drift parameters must be entered exactly as described above. If the drift is to be added to the end (line 19), the number of additional traces can be specified (line 20) instead of the drift area. Lastly, it must be indicated whether a further drift scenario is to be introduced into the event log (line 21).

```
1  Do you want to add an additional drift to the event log [yes, no]? yes
2
3  --- INFORMATION FOR ADDITIONAL DRIFT IN THE EVENT LOG ---
4  Please note that by incautiously setting parameters the previously created drift can be destroyed.
5  It can be decided whether the additional drift is placed at the end or in the event log.
6  If the additional drift for the sudden, gradual and recurring type is added at the end of the log, the number of traces will change.
7  If incremental drift is selected, the number of traces in the resulting event log may change
8  regardless of whether it is set in or at the end of the event log.
9  For the evolution of the process model version, it can be decided between the initial version or the last evolved version.
10
11 Type of 2. concept drift in the event log [sudden, gradual, recurring, incremental]: sudden
12 Process model version for the evolution of the additional sudden drift [initial_version, evolved_version]: evolved_version
13 Controlled or random evolution of the process tree version [controlled, random]: random
14 Proportion of the activities in the process tree to be affected by random evolution (0 < x < 1): 0.2
15
16 Control-flow changes: activity deleted; activity deleted;
17 Activities affected by the changes:[e, b]
18
19 Adding the additional recurring drift at the end of the log or into the log [end, into]? end
20 Number of additional traces from the new model to be added at the end of the event log (int): 500
21 Do you want to add an additional drift to the event log [yes, no]? no
```

Figure 4.8: Input for additional drifts.

Noise Input. The last configuration of the user for his desired event log is the noise introduction (Figure 4.9). After the decision for noise generation (line 1), the user is shown the most important information (line 3-11). Two types of noise are available (line 12). On the one hand, completely random traces that have no structural similarities to the model versions can be used as noise. On the other hand, traces created by a modified version of the initial model can be selected. The exact sector (line 14 & 15) and the amount of noise (line 16) can be specified as a proportion of the total number of occurring traces in the event log. If the user decides to have noise from a changed process model version, he has to specify which proportion of the total activities should be affected by the change (line 17). The process tree for generating the noise is displayed to the user at the end. When all these steps have been processed in the terminal, CDLG exports and saves the generated event log and the generated process models.

```
1  Do you want to add noise to the event log [yes, no]? yes
2
3  --- INFORMATION FOR THE INTRODUCTION OF NOISE ---
4  The noise will be randomly distributed in the sector to be determined.
5  The proportion of noise for this sector can also be predefined.
6  This gives the possibility to either disguise the drift by placing the noise around/inside the drift
7  or to fake another drift by placing the noise away from the drift.
8  The following two types of noise exist:
9      changed_model: the initial model version is changed randomly, whereby the proportion of changes in the version can be specified.
10     random_model: a completely new model is used, which means that there is little or no similarity to the other model versions.
11 The created model, which will appear as an image, will be used to generate traces introduced in the event log as noise.
12
13 Type of model for generating the noise [changed_model, random_model]: changed_model
14 Start point of noise in the event log (0 <= x <= 1): 0.2
15 End point of noise in the event log (0 <= x <= 1): 0.8
16 Proportion of noise in the set sector of the event log (0 < x < 0.5): 0.1
17 Proportion of the changes in the initial tree version for creating the noise (0 < x < 1): 0.3
```

Figure 4.9: Input for noise configurations.

To put it concisely, event log generation via the terminal allows the user a wide range of possibilities. Through many different paths that can be taken several times in the terminal, a very specific desired event log can be generated. The user has exact influence on change point, change characterization and change localization. Furthermore, noise and additional drifts can be introduced into the event log.

4.2 Generation via Text Files

The second option, a quicker way to generate event logs, is accomplished via the respective text files of the different drift scenarios, which are shown in Figure 4.10.

In general, it should be noted that no algorithm checks the inputs and accordingly these must be correct for successful event log generation. The type of input for the different configurations is identical to those from Section 4.1. One space must always be left after the colons and no other spaces (except with timestamp) are allowed, otherwise the algorithm will not import the parameters from the text file correctly and errors will occur. If one or two own models are to be used for generation, their file path must additionally be specified as a parameter after the execution file when running in the terminal.

For all four parameter text files, the tree complexity (i.e. simple, middle or complex) must first be determined if a random tree is to be used for trace generation (Figure 4.10 line 1). In line 2 - 4 of the text files the user has to set the time configurations. This is followed by the drift-typical parameters for each drift scenario, as explained in the previous section. Since only random evolution is possible with this generation type, the proportion of total activities to be affected by the change must be specified in the fifth last line if only one imported or generated model is used (Figure 4.10a line 7).

```
1  Complexity_Random_Tree: complex
2  Timestamp_First_Trace: 20/8/3 8:0:0
3  Min_Time_Activity_Sec: 1200
4  Max_Time_Activity_Sec: 10800
5  Number_Traces: 200
6  Change_Point_Sudden_Drift: 0.7
7  Proportion_Random_Evolution: 0.1
8  Start_Sector_Noise: None
9  End_Sector_Noise: None
10 Proportion_Noise_In_Sector: None
11 Type_Noise: None
```

(a) Sudden drift input.

```
1  Complexity_Random_Tree: middle
2  Timestamp_First_Trace: 20/8/3 8:0:0
3  Min_Time_Activity_Sec: 1200
4  Max_Time_Activity_Sec: 10800
5  Number_Traces: 400
6  Start_Point_Gradual_Drift: 0.36
7  End_Point_Gradual_Drift: 0.62
8  Change_Type: exponential
9  Proportion_Random_Evolution: 0.4
10 Start_Sector_Noise: None
11 End_Sector_Noise: None
12 Proportion_Noise_In_Sector: None
13 Type_Noise: None
```

(b) Gradual drift input.

```
1  Complexity_Random_Tree: middle
2  Timestamp_First_Trace: 20/8/3 8:0:0
3  Min_Time_Activity_Sec: 1200
4  Max_Time_Activity_Sec: 10800
5  Number_Traces: 10000
6  Start_Point_Recurring_Drift: 0
7  End_Point_Recurring_Drift: 1
8  Seasonal_Changes: 5
9  Proportion_Initial_Model_During_Drift: 0.5
10 Proportion_Random_Evolution: 0.2
11 Start_Sector_Noise: None
12 End_Sector_Noise: None
13 Proportion_Noise_In_Sector: None
14 Type_Noise: None
```

(c) Recurring drift input.

```
1  Complexity_Random_Tree: middle
2  Timestamp_First_Trace: 20/8/3 8:0:0
3  Min_Time_Activity_Sec: 1200
4  Max_Time_Activity_Sec: 10800
5  Number_Traces_Initial_Model: 3000
6  Number_Traces_Drift_Models: 1000
7  Number_Traces_Evolved_Model: 3000
8  Number_Intermediate_Models: 4
9  Proportion_Random_Evolution: 0.25
10 Start_Sector_Noise: 0.1
11 End_Sector_Noise: 0.8
12 Proportion_Noise_In_Sector: 0.05
13 Type_Noise: random_model
```

(d) Incremental drift input.

Figure 4.10: Parameters in text files for drift types.

For incremental drift, a certain number of traces cannot be determined for each intermediate model version of the drift, but a constant number is maintained (Figure 4.10d line 6). Likewise, all intermediate evolved versions are modified with the same proportion (Figure 4.10d line 9). If two imported models are used, the first model version will evolve randomly with the set number of times (Figure 4.10d line 8) and the second imported model will be used as the final evolved version for the trace generation. Therefore, no intermediate drift model versions can be imported for the incremental drift generation.

In the last four lines the configurations for noise can be specified. If no noise is to be generated, 'None' must be entered at least in the first line of the noise parameters, as shown in Figure 4.10a (line 8-11). In the last line the different noise generation models (random_model or changed_model) can be chosen, where for 'changed_model' a fixed amount of 40% of the activities in the initial tree version are affected by the change. After running this path with the appropriate parameters set in the text file, CDLG exports the generated event log and its respective process trees.

Considering this, with the generation of the event logs through the specific text files, the user can swiftly create his desired event log with the most important drift configurations, when no controlled evolution of the model and no additional drifts are required.

4.3 Generation Gold Standard

The third option and the fastest way to generate several event logs is via the file for gold standard generation, shown in Figure 4.11.

In general, the input conditions from Section 4.2 apply here as well. In addition, ranges must be marked by '-' without spaces (Figure 4.11 line 5-7). With this event log creation, it is only possible to import one model, which is then modified for each event log generation differently by the random evolution.

Ten parameters are required for the generation, which are shown in Figure 4.11. First, the desired complexity (i.e. simple, middle or complex) is needed when using a random process tree (line 1) and subsequently the number of event logs to be generated (line 2) must be entered. In line 3 the number of traces to be created per event log and in line 4 the possible drift types to occur in an event log must be specified. Multiple entered drift types (i.e.sudden, gradual, recurring and incremental) must be separated by ';' without spaces, otherwise the algorithm will not import them correctly. In contrast, a individually entered drift type can be specified without ';'. The range where the start point and the end point of the drift should occur must be defined in line 5. The drift area must be large enough (> 0.1), as otherwise the random generated drifts will be too short or not present at all. In line 6 the user has to set the sector of the proportion for the random evolution of the initial process tree version. If noise is to be introduced into the event log, the range can be specified in line 7. Otherwise, '0' or 'None' must be inserted. In line 8 to 10 the time configurations for the event log must be entered.

```
1  Complexity_Random_Tree: middle
2  Number_Event_logs: 100
3  Number_Traces_Per_Event_Log: 400
4  Drifts: sudden;gradual;recurring;incremental
5  Drift_Area: 0.3-0.7
6  Proportion_Random_Evolution_Sector: 0.2-0.3
7  Proportion_Noise_Sector: 0-0.3
8  Timestamp_First_Trace: 20/8/3 8:0:0
9  Min_Time_Activity_Sec: 1200
10 Max_Time_Activity_Sec: 10800
```

Figure 4.11: Gold standard generation parameters in text file.

During this generation, CDLG randomly selects one of the specified drift types (line 4) with random start and end change points (line 5) and a random evolution proportion (line 6) for each event log generation. Additionally, random values for the drift-specific configurations are chosen from a fixed range for the different drift scenarios. With gradual drift, a decision is made between linear and exponential distribution. For the recurring drift, seasonal changes range from one to six and the proportion of the total number of traces generated from the initial version ranges from 0.3 to 0.7. With incremental drift, CDLG randomly selects between two to five intermediate evolving model versions and, for each, adjusts the proportion of total activities to be affected by the random evolution. Noise is distributed over the entire event log and a random decision is made between the noise types. These ranges are chosen to ensure that the generated drifts are recognizable even in small drift areas.

After execution, the event logs as well as the initial process tree version are exported and saved. A CSV file is also created, in which the data explained in Table 4.2 is stored for each event log. Moreover, all parameters used and the drift information of the CSV file are stored in a text file under the same identifier for each event log. Thus, the user has the possibility to reproduce certain drifts using the saved parameters via the generation path from Section 4.1 or 4.2.

In summary, gold standard generation can be used to generate a large number of event logs very quickly and thus save a significant amount of time during generation. However, via this way, the user has no influence on a controlled model evolution and is not able to insert additional drifts. Moreover, the drift configurations are specified randomly by CDLG and mainly random values from ranges are used. But, since all important randomly configured drift data is stored in a CSV file and a text file, these event logs can be efficiently used to evaluate process mining algorithms.

Name	**Meaning**
Event Log	Identifier of the event logs
Drift Perspective	Location of the drift (i.e. control-flow)
Drift Type	Type of drift
Drift Specific Information	For each drift type additional information
Drift Start Timestamp	Start change point of drift
Drift End Timestamp	End change point of drift
Noise Proportion	Proportion of noise in the whole event log
Activities Deleted	Activities deleted by A2, A3, F2 (Table 3.2)
Activities Added	Activities added by A1, A3, F1 (Table 3.2)
Activities Moved	Activities moved by A4, O1, O2, F1, F3, F4 (Table 3.2)

Table 4.2: Column headings of the gold standard CSV file.

4.4 Summary

As mentioned in the previous sections, each option for generating the event logs through CDLG has its advantages and disadvantages, which are summarized in Table 4.3. The generation via the terminal (Section 4.1) is the most advanced one and allows the most flexibility in the generation of event logs. The only drawback is that it can be time consuming and complex. The generation via text files (Section 4.2) is much faster, but has limitations regarding the IDs 1,2,3 and 4 of Table 4.3. Since gold standard generation (Section 4.3) can create as many event logs as desired with one run, it has by far the best efficiency in terms of effort. However, the user is limited in the event log generation regarding the IDs 1,2,3,4 and 5, and must rely mainly on random drift configurations. These three generation options allow the user to generate his required event logs with concept drifts with different effort and accuracy.

ID	Configurations	Terminal	Text Files	Gold Standard
1	Import models	1-2	1-2	1
2	Change point control	full	full	limited
3	Model evolution	con. & ran.	random	random
4	Additional drifts	∞	none	none
5	Noise control	full	full	limited
6	Efficiency	low	medium	high

Table 4.3: Overview of event log generation options.

Chapter 5

CDLG Evaluation

In this section, synthetically generated event logs by CDLG are evaluated using the Visual Drift Detection (VDD) technique [5] developed by Yeshchenko et. al.. This approach is deliberately chosen because in the papers [15] and [16], after evaluating many different concept drift detection techniques, it is verified that VDD is able to detect the four concept drift scenarios regarding their change points and their change in the control-flow. Moreover, as an offline analysis technique, it requires event logs and not event streams. All other reviewed approaches do not bring these necessary conditions for a successful evaluation of CDLG [15], [16].

In this case, the intention of using VDD is not only to show that CDLG generates the drifts with the parameters set, but also to show that CDLG is suitable for an evaluation of concept drift detection approaches. Accordingly, VDD is evaluated using CDLG generated data, in which the configurations are known in advance, to determine to what extent it detects these settings.

5.1 Visual Drift Detection Tool

The Visual Drift Detection Tool [5] clusters declarative process constraints, which are discovered in the event logs, according to their similarity and finds the possible concept drifts by applying change point detection to the identified clusters. By these constraints, the authors mean temporal rules that must be satisfied during the execution of activities. This approach allows users to clearly identify drifts in event logs through visualizations of drift maps, drift charts and DFGs, as shown for instance in Figure 5.2 below [5].

The *drift maps*, on the one hand, visualise all identified clusters over the entire period. The x-axis represents the time axis and the constraints can be found on the y-axis. The drifts are represented by the horizontal changes of the colors. Vertical

dashed lines are added to clearly indicate the change points and horizontal dashed lines are inserted to separate the identified clusters. *Drift charts*, on the other hand, represent only one cluster and have time on the x-axis and average confidence of the constraints are found on the y-axis. They are suitable for identifying the specific drift types. Likewise, VDD provides *DFG visualization* for each cluster showing the changes in the control-flow. This graph combines annotations from Declare constraints, which are explained in [17], and 'Directly Follows Graphs' [5]. Additionally, VDD gives for each cluster a probability for the presence of an incremental drift.

For the evaluation of the CDLG, the resulting visualizations are evaluated and interpreted accordingly. Only the most erratic drift chart will be illustrated for each event log, as it is closest to correctly identifying the drift. Since the DFG visualizations are large and would make this report confusing, they are placed in the Appendix B and only their interpreted solutions are explained.

5.2 VDD Results

The evaluation is conducted as follows. First, for each drift scenario an event log containing 5000 traces without noise is created by CDLG. For this purpose, the same process tree, depicted in Figure 5.1, is evolved for each event log individually with a change type from Table 3.2. Therefore, each event log differs in both drift type and change localization. These logs are evaluated by the Visual Drift Detection Tool with its set default parameters and it is checked to what extent it detects the set drift configurations (i.e. change point and control-flow change). If VDD successfully finds the drift configuration in one of the event logs, a new event log is generated with the same configurations, in which noise is introduced during the drift. Up to 10% noise will be distributed in the respective sectors, even if this is an uncommonly large amount, as the main change remains even at 90%. This event log is then analyzed again by VDD and the two drift charts and the respective DFGs, the first without noise and the second with noise, are compared. The drift maps associated with the event logs with noise are not included, as the most significant changes can be identified in the drift charts. At the end the respective results are interpreted.

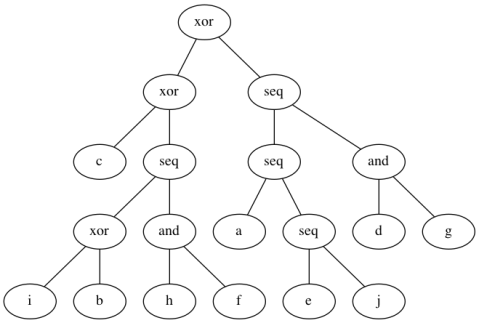


Figure 5.1: Initial process tree version used for the evaluation.

Based on this evaluation, it will be demonstrated that CDLG is capable of generating a variety of event logs with concept drifts that can be of different complexity for concept drift detection.

5.2.1 Sudden Drift

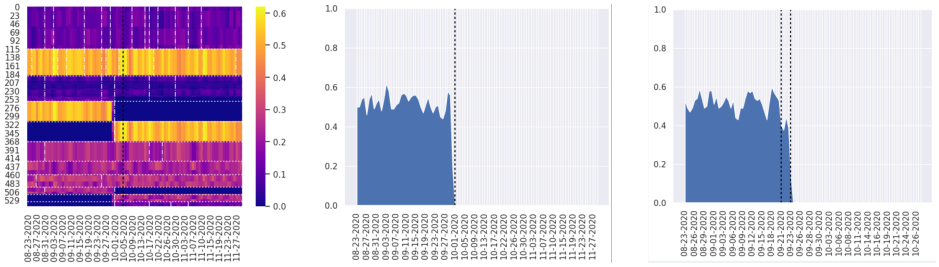
For the event log with a sudden drift, the drift specific configurations are specified in Table 5.1. During the controlled evolution of the process tree version (Figure 5.1), activity 'a' is replaced by 'z' (Figure B.1a).

Configuration	**Setting**
Evolution	Activities a, z affected by A3 (Table 3.2)
Change point	0.45

Table 5.1: CDLG configurations for sudden drift.

Without noise. The Figures 5.2a and 5.2b illustrate the results of VDD for the event log including a sudden drift without noise. In this case, clusters 4, 5, 9, 10, and 11 clearly indicate in Figure 5.2a the sudden drift at about 0.45. This is also evidenced by the abrupt change of the average constraint from about 0.6 to 0 in the drift chart (Figure 5.2b) from cluster 4. The corresponding DFG (Figure B.2a) indicates that activity 'a' only occurs directly before 'e' until the abrupt change.

Based on this observation, it can be concluded that VDD identifies all sudden drift configurations in the generated event log by CDLG. It detects the sudden change at 0.45 as defined during generation. While it does not exactly specify the replacement of activity 'a' with 'z' in one DFG, it does mark these two clearly as change localization individually in several DFGs and therefore finds the change in the control-flow.



(a) Drift map. (b) Drift chart without noise. (c) Drift chart with noise.

Figure 5.2: VDD visualizations for sudden drift.

With noise. Figure 5.2c is the resulting drift chart of VDD from an event log with the same drift configurations from Table 5.1, except that in this case noise was distributed during the drift. A total of 7% noise is introduced in the range of 0.4 to 0.5 in an attempt to disguise the drift. Furthermore, 40% of the initial process tree version is randomly changed for the generation of the noise traces.

Due to the noise, two vertical black lines are now displayed in the drift chart and it could be interpreted as a swiftly occurring gradual drift. However, based on the average constraint, a sudden drift is still clearly recognizable. The corresponding DFG (Figure B.2b) indicates the correct change localization, but it is less clear as it shows the execution paths of the noise.

From these observations, it is evident that VDD still recognizes the drift configurations from the sudden drift despite a 7% noise distribution. However, this assumes that the user correctly interprets VDD's various indications and clusters.

5.2.2 Gradual Drift

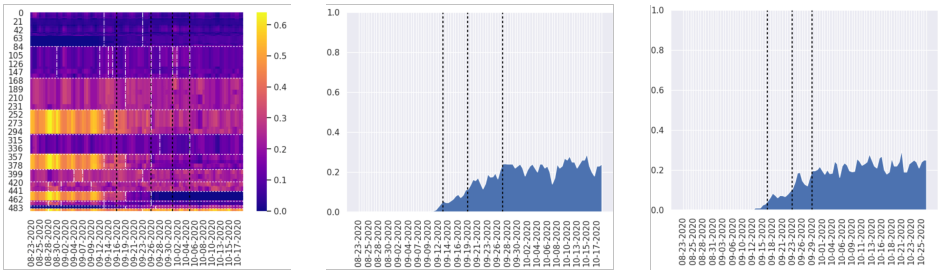
The set configurations for the event log with gradual drift are in Table 5.2. In this case, the operator 'seq' in depth 2 before activity 'a' is exchanged with the operator 'xor' during the evolution of the model version (Figure B.1b).

Configuration	Setting
Evolution	Activities a, e, j affected by O1 (Table 3.2)
Start point	0.3
End point	0.6
Distribution	linear

Table 5.2: CDLG configurations for gradual drift.

Without noise. The results of VDD shown in Figure 5.3 clearly illustrate an occurring gradual drift in the event log. In clusters 4, 6, 9, 10, and 11 (Figure 5.3a) the gradual drift is evident from the continuous color change. The drift chart (Figure 5.3b) shows this by the steady increase of the average constraint. Here the approximate period from about 0.3 to 0.6 of the drift can likewise be identified. In the DFG (Figure B.3a), the change in the new model version is specified in such a way that activity 'a' is never executed before 'e' and 'j'.

Through these visualizations, it can be inferred that the gradual drift configurations of the event log are discovered by VDD. It clearly shows the change points set by CDLG (Table 5.2) and due to the linear change of the color in the drift map and the linear change of the average constraint in the drift chart the linear distribution of the traces is identified. Moreover, the exact impact on the control-flow because of the change of the operator is detected, since the replacement of 'seq' with 'xor' results in the activity 'a' never occurring before 'e' and 'j'.



(a) Drift map. (b) Drift chart without noise. (c) Drift chart with noise.

Figure 5.3: VDD visualizations for gradual drift.

With noise. For this event log, 10% noise is distributed in the range 0.4 to 0.75, with traces differing only slightly by 10% from those generated by the initial model version. The reason for this is mainly an attempt to disguise the second change point.

At first glance, the two drift charts from Figure 5.3 look identical. The linear gradual drift can be clearly identified in Figure 5.3c. Likewise, the DFGs of the two charts are identical (Figure B.3) and thus, despite noises, the exact change location is detected. However, the drift observed in the chart with noise is shorter than that in Figure 5.3b, as it ranges from about 0.3 to 0.55.

Based on this analysis, it can be observed that VDD detects gradual drifts even if noise, quiet similar to real traces, is distributed in the event log. Although the change points may differ slightly, the approximate range is still correct and so is the change in the control-flow.

5.2.3 Recurring Drift

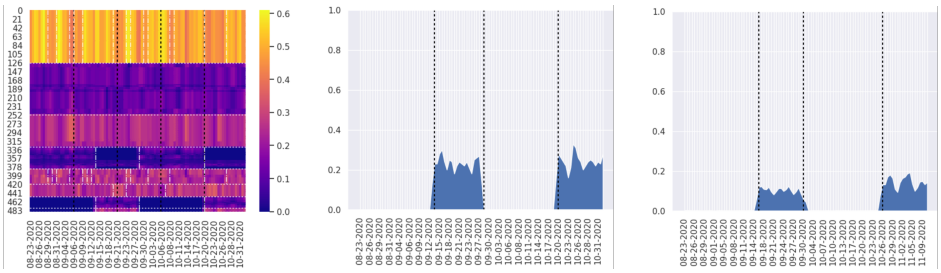
In this case, the recurring drift-specific configurations are specified in Table 5.3. During controlled evolution the tree fragment starting in depth three with the operator 'xor' is swapped with the tree fragment starting also in depth three with the operator 'and' (Figure B.1c).

Configuration	Setting
Evolution	Activities i, b, h, f affected by F3 (Table 3.2)
Start point	0
End point	1
Seasonal changes	3
Proportion initial version	0.6

Table 5.3: CDLG configurations for recurring drift.

Without noise. VDD detects the recurring drift in the event log, as shown in Figure 5.4. In the drift map in clusters 4, 7, and 8, as well as in the drift chart of cluster 8, three seasonal changes in process execution are indicated. The drift appears to be continuous in the event log, whereby the initial season of the process lasts longer than the subsequent season of the changed process. The corresponding DFG (Figure B.4a) shows that the activity 'b' and 'i' only occur when 'f' and 'h' are executed beforehand.

Based on these observations, it can be inferred that VDD detects the configurations of the recurring drift specified by CDLG. It identifies the three seasonal changes and the different duration of periods set by the proportion of the total traces generated by the initial version (Table 5.3). VDD highlights the four activities 'i', 'b', 'h' and 'f' affected by CDLGs evolution of the initial tree version correctly and thus detects the change in the control-flow.



(a) Drift map. (b) Drift chart without noise. (c) Drift chart with noise.

Figure 5.4: VDD visualizations for recurring drift.

With noise. The Figure 5.4c depicts the results of an event log with the same configurations of table 5.3 with the addition of noise. In this case, 10% of random noise (no similarity to any version) is distributed in the area 0.2 to 0.6. This is intended to mask the first seasonal execution period of the evolved process version.

In the two drift charts of Figure 5.4, a clear difference in the level of the average constraint can be observed, rendering the recurring drift in Figure 5.4c less noticeable. Moreover, the DFGs (Figure B.4) differ due to the noise, but the localization of the change is still reported correctly, although not as accurately as without noise. Thus, all activities that are affected by the evolution are indicated and the drift points visible in the chart are correct.

Consequently, VDD detects all important drift configurations generated by CDLG despite a 10% noise distribution and only becomes slightly less accurate.

5.2.4 Incremental Drift

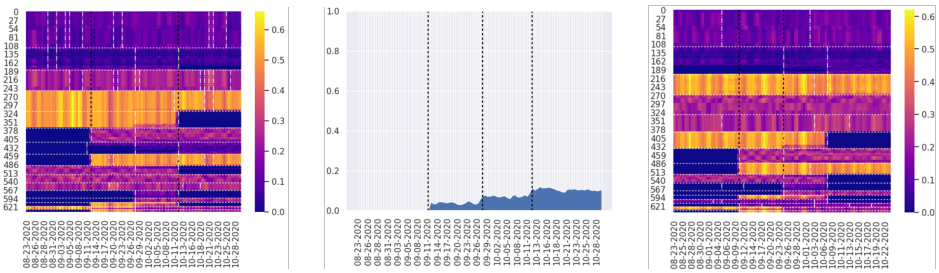
When generating the event log with an incremental drift, the initial version of the process tree is evolved three times in succession, as illustrated in Table 5.4. First, a tree fragment with operator 'and' and its activity 'z' is added to the activity 'a'. Second, the operator 'seq' in depth 2 in front of the new tree fragment is replaced by 'and'. Last, activity 'j' is deleted (Figure B.1d).

Configuration	Setting
Evolving versions	3
Traces initial version	1500
Evolution 1. version	Activities a, z affected by F1 (Table 3.2)
Traces 1. version	1000
Evolution 2. version	Activity a, z, e, j affected by O1 (Table 3.2)
Traces 2. version	1000
Evolution 3. version	Activity j affected by A2 (Table 3.2)
Traces 3. version	1500

Table 5.4: CDLG configurations for incremental drift.

Without noise. Figure 5.5 depicts the results of VDD, where two clear change points are recognizable in the drift map (Figure 5.5a) and even three small change points of the process execution in the drift chart of cluster 7 (Figure 5.5b) by vertical black lines. These changes occur after about 30%, 50% and 70% of the traces in the event log. Moreover, the probability reported for the presence of an incremental drift is about 82%. In order to analyze all these change points, several clusters have to be looked at. Many of these clusters, in addition to Figure 5.5b, have one of these changes reported in both the drift chart and DFG. For instance, the DFG of

cluster 8 indicates the appearance of a new activity 'z' for the first change point and DFG of cluster 5 the disappearance of activity 'j' for the last change point. Furthermore, the DFG of cluster 11 shows the new change that every time 'j' occurs, it is immediately followed by 'a' and 'z' for the second change point.



(a) Drift map without noise. (b) Drift chart. (c) Drift map with noise.

Figure 5.5: VDD visualizations for incremental drift.

Therefore it can be concluded that VDD determines all change points of the incremental drift. Additionally, the locations of the respective changes are detected relatively accurately, as a comparison of the observations with the configurations from Table 5.4 and the evolved version (Figure B.1d) shows. The probability for the incremental drift of cluster 7 gives an indication for the presence of the drift. Nevertheless, the user has to identify the incremental drift himself based on these observations, since the user cannot rely completely on the probability.

With noise. Over the entire event log with the same drift configurations (Table 5.4), only random noise of 3% is distributed to make an exact detection of the three drifts just slightly more challenging and to analyse the handling of VDD with a small amount of noise.

In this case, the drift map (Figure 5.5c) is referred to for comparison instead of the drift chart. This is because VDD does not generate or completely regenerate the drift chart from Figure 5.5b after the noise distribution and therefore no reasonable comparison is possible. Furthermore a solid drift chart with a high probability of the presence of an incremental drift is not given by VDD. In the drift map, however, some differences can be identified. For example, clusters, such as cluster 7, are gone or modified. The drift map in Figure 5.5c also appears more patchy in its color distribution. Nevertheless, the change points are recognizable in the drift map and the DFGs visualize the exact change localization for each change point as well. Moreover, all DFGs indicating the specific changes in the control-flow remain identical (example for DFG 5 in Figure B.5a and B.5b) .

Based on these observations, it is evident that VDD detects the incremental drift configurations in terms of change point and change localization despite a 3% noise distribution. Even when clusters change and the drift map becomes more vague.

5.3 Discussion

By referring to the evaluation of the Visual Drift Detection Tool in the Section 5.2 on CDLG synthetically generated data, it can be seen how accurate and advanced VDD actually is. If the drift map and all the different drift charts and DFGs are analyzed and interpreted correctly, it is possible to determine all set change points, change localization and change characterizations (i.e. drift types). Even with a high noise distribution of up to 10%, the drifts are still evident. Furthermore, in some cases VDD even assigns noise to its own cluster, as in the analysis of the event log with a recurring drift (Figure B.6). However, the respective drift maps, charts and DFGs may also become fuzzier, resulting in slightly inaccurate change points and changes in the control-flow. Hence, it can be generally concluded that VDD is an advanced tool that analyzes event logs extensively and outputs the analysis data by means of detailed visualizations. Nevertheless, these graphs only give information about the change point and the control-flow change. Noise is neither marked as noise nor ignored, but is included in clusters and DFGs. The user has to interpret the visualizations himself to infer the type of drift scenario and to understand the occurring noise. Moreover, a solid understanding of DFGs is required to interpret them correctly to determine the actual change in control-flow.

By evaluating the VDD tool using event logs generated by CDLG, it is shown that CDLG is well suited for evaluating concept drift detection tools. The visualizations of Section 5.2 clearly indicate the four concept drift types generated by CDLG with the set change points and control-flow changes and the effects when noise is distributed. Through this evaluation it is clear that CDLG generates event logs exactly according to the configurations set by the user. This brings the main advantage of this approach that all specified configurations are known in advance and therefore the response of the process mining technique to the given scenarios can be precisely evaluated. Thus, the conclusion can be drawn that CDLG enables event logs to be generated with a wide variety of scenarios, evaluating process mining approaches in a broad range of ways.

Chapter 6

Related Work

As access to real event logs has been a challenge since the inception of the research discipline process mining, several approaches to generate synthetic event logs have been developed and published. However, most of these tools [13, 18, 19, 20] generate event logs in which concept drifts cannot be introduced and therefore are not suitable for the evaluation of concept drift detection methods. They mainly specialise in creating event logs for the initial process mining techniques, such as control-flow detection techniques. To put CDLG in context, this chapter briefly reviews the two known approaches for generating synthetic event logs with concept drifts.

Manual generation of event logs for the evaluation of process mining algorithms is the most intuitive way and provides the greatest scope, as the user is completely in control of the generation process. The exact change point, change localization and the type of drift can be specified. However, the limitations of manual generation outweigh its advantage, as it consumes a significant amount of time and several data errors are inevitable [18].

Process Log Generator 2 [3], published by Andrea Burattin, as an extended version of the Process Log Generator, is so far the only research tool available that is able to consider concept drifts when generating event data. It provides the capabilities to build or import process models, to randomly evolve models, to create multiperspective (i.e. time and data perspective) event logs with noise, and to generate multiperspective streams, where concept drifts can occur [3]. The support of concept drifts in PLG2 is only possible during the simulation of online event streams. The user can change the source for generating event data during the stream run, which influences the change point directly and provides the user with control over the change localization (i.e control-flow change). In the case of changing the process model, the user has two options. First, a new model can be

randomly created or uploaded, meaning that the control-flow change can theoretically be influenced if the appropriate model is owned and uploaded by the user. Second, the existing model can be evolved randomly, giving the user no influence on change localization. In Burattin's approach, the different concept drift scenarios are not addressed and only one general concept drift is referred to [3].

These two approaches demonstrate clearly that an approach like CDLG for synthetic event log generation with all concept drift types is essential for the process mining community. Despite the fact that PLG2 is a sophisticated tool that is more advanced than CDLG in terms of multiperspective event logs, it does not provide a controlled evolution of the process models or control over the different drift types either. Therefore, no real approach exists yet that efficiently generates these event logs, where full control over the key configurations is provided. CDLG fills this gap with its various options for creating event logs and through the flexibility in configuring the drifts.

Chapter 7

Conclusion and Future Work

This bachelor thesis describes CDLG, which is a new approach to generate synthetic event logs with different drift scenarios. It provides the ability to generate event logs with user-controlled drift configurations via three ways (i.e. terminal, text files, gold standard). All four concept drift types with their specific configuration regarding change point and change localization (i.e. controlled or random evolution of the process tree) can be created into an event log. Furthermore, two types of noise can be specifically distributed and several drifts can be added.

This newly developed approach is intended to be a valid tool for the process mining community to efficiently obtain synthetic event logs with all required configurations. It is designed to help researchers better evaluate their developed concept drift detection techniques and more quickly and accurately address the emerging challenges, as CDLG provides simple generation of event logs with several configurations.

CDLG is the first step towards an approach that allows to generate synthetic event logs with concept drifts according to all the user's requirements. Consequently, it still has limitations that calls for future work. On the one hand, CDLG requires extensions that provide drifts in the time and data perspectives as well as in their respective noise introduction. On the other hand, control-flow paths should be able to be executed with different probabilities, so that control-flow changes are feasible even in this domain with suitable configurations. The possibility to create recurring drifts with several models and to use the gradual drift for the change of processes in the recurring and incremental drift should be additionally enabled. It also needs to be evaluated to what extent the CDLG is able to generate usable event logs with multiple drifts. Lastly, a further modification in the form of a graphical user interface is required to increase usability, as the generation via terminal can become complex (re controlled evolution).

Bibliography

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

[2] Jagadeesh Chandra Bose R.P., Wil Van Der Aalst, Indre Zliobaite, and Mykola Pechenizkiy. Dealing with concept drifts in process mining. *IEEE Transactions on Neural Networks and Learning Systems*, 25(1):154–171, May 2014.

[3] Andrea Burattin. PLG2: Multiperspective process randomization with online and offline simulations. In *CEUR Workshop Proceedings*, volume 1789, 2016.

[4] Alessandro Berti, Sebastiaan J. Van Zelst, Wil M.P. Van Der Aalst, and Fraunhofer Gesellschaf. Process mining for python (PM4py): Bridging the gap between process- And data science. In *CEUR Workshop Proceedings*, volume 2374, 2019.

[5] Anton Yeshchenko, Claudio Di Ciccio, Jan Mendling, and Artem Polyvyanyy. Comprehensive process drift analysis with the visual drift detection tool. In *CEUR Workshop Proceedings*, volume 2469, 2019.

[6] Wil van der Aalst, Joos Buijs, and Boudewijn van Dongen. Towards improving the representational bias of process mining. In Karl Aberer, Ernesto Damiani, and Tharam Dillon, editors, *Data-Driven Process Discovery and Analysis*, pages 39–54, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Toon Jouck and Benoît Depaire. Generating Artificial Data for Empirical Analysis of Control-flow Discovery Algorithms: A Process Tree and Log Generator. *Business and Information Systems Engineering*, 61(6), 2019.

[8] H. Verbeek, Joos Buijs, Boudewijn Dongen, and Wil Aalst. Xes, xesame, and prom 6. *Lecture Notes in Business Information Processing*, 72:60–75, 06 2010.

[9] J. Martjushev, R. P. Jagadeesh Chandra Bose, and Wil M.P. van der Aalst. Change point detection and dealing with gradual and multi-order dynamics in process mining. In *Lecture Notes in Business Information Processing*, volume 229, 2015.

[10] Abderrahmane Maaradji, Marlon Dumas, Marcello La Rosa, and Alireza Ostovar. Detecting sudden and gradual drifts in business processes from execution traces. *IEEE Transactions on Knowledge and Data Engineering*, 29(10), 2017.

[11] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features - Enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering*, 66, 2008.

[12] Raffaele Conforti, Marcello La Rosa, and Arthur H.M. ter Hofstede. Noise Filtering of Process Execution Logs based on Outliers Detection. *Institute for Future Environments; School of Information Systems; Science & Engineering Faculty*, 2015.

[13] Ivan Shugurov and Alexey Mitsyuk. Generation of a set of event logs with noise. In *Spring/Summer Young Researchers' Colloquium on Software Engineering*, 01 2014.

[14] Sebastiaan J. van Zelst and Sander J.J. Leemans. Translating workflow nets to process trees: An algorithmic approach. *Algorithms*, 13(11), 2020.

[15] Ghada Elkhawaga, Mervat Abuelkheir, Sherif I. Barakat, Alaa M. Riad, and Manfred Reichert. CONDA-PM—A Systematic Review and Framework for Concept Drift Analysis in Process Mining. *Algorithms*, 13(7), 2020.

[16] Denise Maria Vecino Sato, Sheila Cristiana De Freitas, Jean Paul Barddal, and Edson Emilio Scalabrin. A Survey on Concept Drift in Process Mining. *ACM Computing Surveys*, 54(9), 2022.

[17] Claudio Di Ciccio and Massimo Mecella. On the discovery of declarative control flows for artful processes. *ACM Transactions on Management Information Systems*, 5:1–37, 03 2015.

[18] Alexey A. Mitsyuk, Ivan S. Shugurov, Anna A. Kalenkova, and Wil M.P. van der Aalst. Generating event logs for high-level process models. *Simulation Modelling Practice and Theory*, 74, 2017.

[19] Ana Karla Alves de Medeiros and Christian W. Günther. Process mining: Using cpn tools to create test logs for mining algorithms. *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, 2005.

[20] Toon Jouck and Benoît Depaire. Generating artificial data for empirical analysis of control-flow discovery algorithms. *Business & Information Systems Engineering*, 61(6):695–712, 2018.

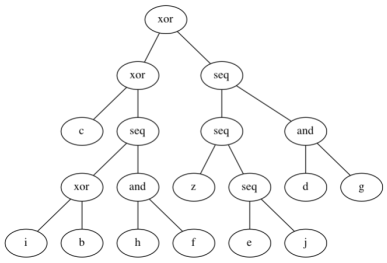
Appendix A

Program Code

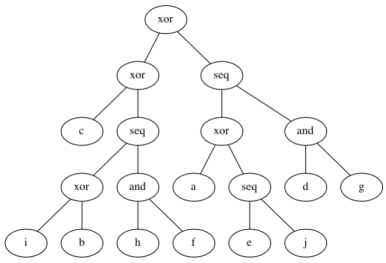
The source code, documentation on how to run CDLG, and the event logs used for evaluation are available at the GitHub repository 'ConceptDriftLogGenerator'.

Appendix B

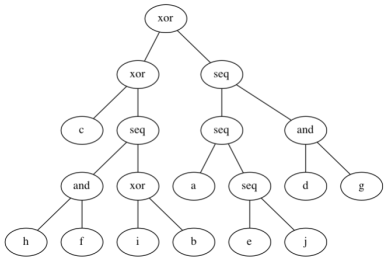
Further Experimental Results



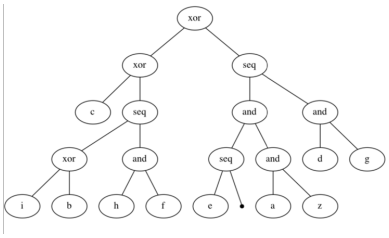
(a) Evolved version for sudden drift.



(b) Evolved version for gradual drift.

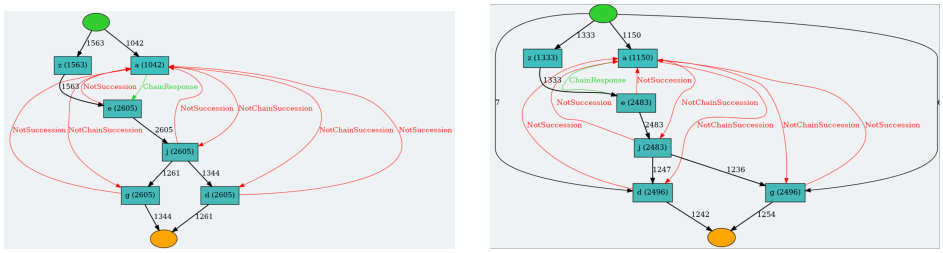


(c) Evolved version for recurring drift.



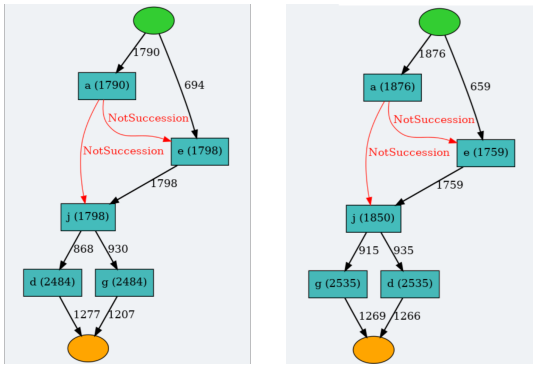
(d) Evolved version for incremental drift.

Figure B.1: Evolved process tree versions for the evaluation.



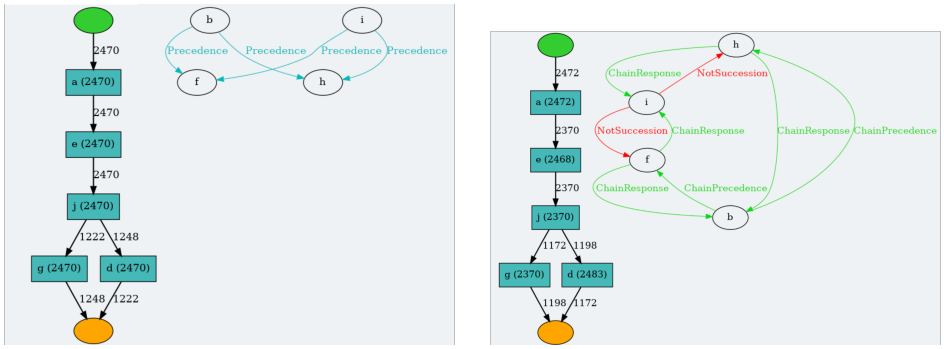
(a) DFG without noise. (b) DFG with noise.

Figure B.2: DFGs for sudden drift.



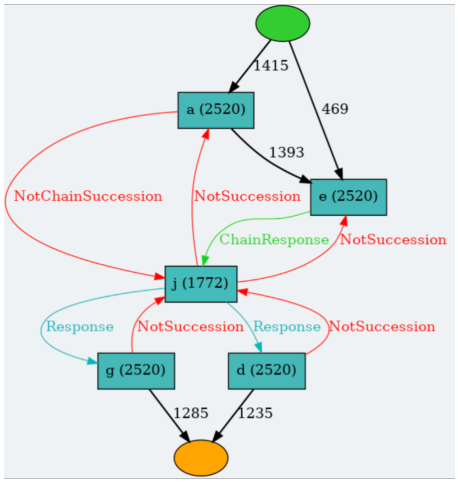
(a) DFG without noise. (b) DFG with noise.

Figure B.3: DFGs for gradual drift.

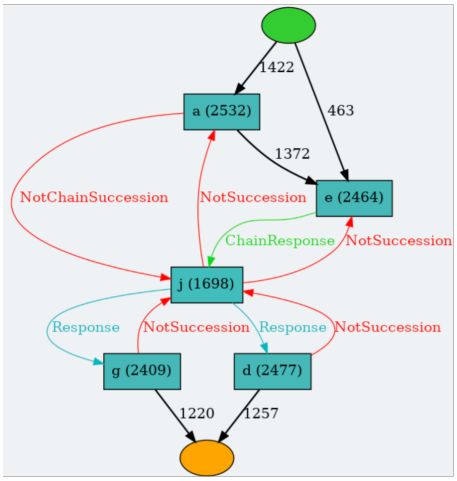


(a) DFG without noise. (b) DFG with noise.

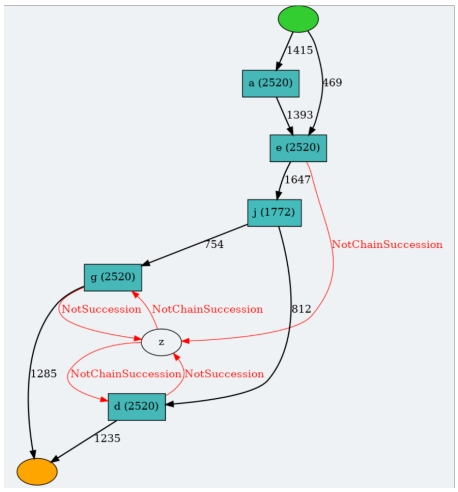
Figure B.4: DFGs for recurring drift.



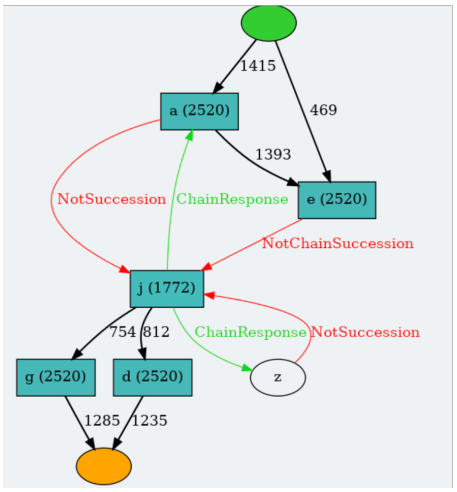
(a) DFG 5 without noise.



(b) DFG 5 with noise.



(c) DFG 8 without noise.



(d) DFG 11 without noise.

Figure B.5: DFGs for incremental drift.

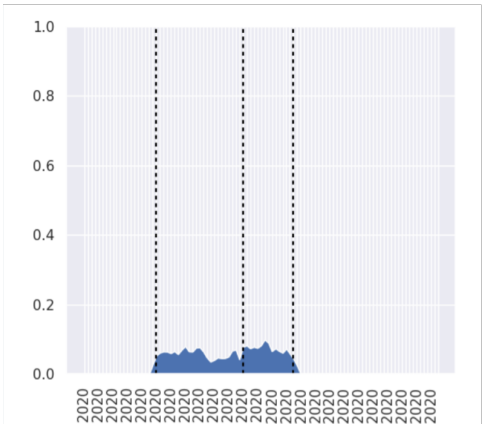


Figure B.6: Detected noise in the event log with a recurring drift.