Discovering Instance Spanning Exceptions from Process Execution Logs

1st Florian Stertz
University of Vienna
Faculty of Computer Science
Vienna, Austria
florian.stertz@univie.ac.at

2nd Karolin Winter
Technical University of Munich
Department of Informatics
Munich, Germany
karolin.winter@tum.de

3rd Stefanie Rinderle-Ma
Technical University of Munich
Department of Informatics
Munich, Germany
stefanie.rinderle-ma@tum.de

***Abstract*—Exceptions in process execution occur frequently and require appropriate handling strategies in order to avoid undesired consequences. For quality control in manufacturing processes, for example, when a blade gets missing, the set of affected instances is put on hold until the blade is found. As it can be seen from this example, the exception affects multiple instances and is hence denoted as instance spanning exception. Exceptions leave footprints in the process execution logs of the affected instances. Hence, process execution logs provide a valuable data source for discovering and analyzing exceptions. However, the discovery of instance spanning exceptions is still an open challenge. Thus, this paper proposes i) a classification of instance spanning exceptions based on literature and a set of real-world examples, followed by ii) a description of how instance spanning exceptions manifest in process execution logs along with an elicitation of minimal requirements for enabling their discovery, and iii) five instance spanning exception discovery algorithms, one for each class. The discovery algorithms are implemented and evaluated on a set of synthetic process execution logs reflecting real-world instance spanning exceptions and on a real-world process execution log from the public transport domain demonstrating the feasibility as well as applicability of the presented algorithms.**

***Index Terms*—Instance Spanning Exceptions, Exception Discovery and Handling, Process Analysis and Improvement, Process Aware Information Systems**

I. Introduction

Exceptions during process execution occur frequently and require appropriate handling strategies in order to avoid undesired consequences [11], [20]. [6], for example, show that especially unexpected exceptions cause longer throughput times and suggest to invest into the analysis of exceptions. Exceptions might also affect multiple instances of one or several process types and are particularly critical due their inherent effects on multiple process instances. We refer to these exceptions as *instance spanning exceptions*. Consider, e.g., a manufacturing process in the food industry where the regular maintenance encounters issues in the production line such as a missing blade. In this case all instances produced since the last maintenance must be searched for the blade and running instances can, e.g., be set on hold during that search process. Discovering such instance spanning exception behavior based on process execution logs hence becomes crucial in order to understand and avoid consequences such as tremendous costs caused by production downtimes. Discovery of instance spanning exceptions comprises the discovery of the *exception trigger*, e.g., in which machine the blade got lost, as well as the *exception handling* part, e.g., setting running instances on hold and searching for the blade in previously produced instances. As instance spanning exceptions can be spawned based on instance spanning constraints, the discovery of instance spanning exception is closely related to the discovery of instance spanning constraints [24], [25] as well. Moreover, the detection of batch processing activities [15] and concept drifts [14], [21] are also related research areas. Yet, discovery of instance spanning exceptions from process execution logs has not been comprehensively investigated so far. This paper aims at bridging this gap via addressing the following research questions.

RQ1 How do instance spanning exceptions manifest in process execution logs?

RQ2 How to discover instance spanning exception triggers and handling from process execution logs?

Based on literature and real-world examples, five distinct instance spanning exception classes *wait*, *cancel*, *redo*, *change* and *rework* are elicited as well as minimal requirements on process execution logs enabling their discovery ($\mapsto$ *RQ1*, Sect. II). Five discovery algorithms, one for each instance spanning exception class are presented ($\mapsto$ *RQ2*, Sect. III). Each algorithm discovers exception triggers and their handling which contributes to enhancing explainability of discovered exceptions as well as enabling support of future exception prevention actions. The algorithms are prototypically implemented and evaluated based on synthetic data sets for each of the exception classes as well as on a real-world data set from the public transport domain. The evaluation, cf. Sect. IV, demonstrates the feasibility and applicability of the presented algorithms. Afterwards, a brief discussion (cf. Sect. V) as well as related work (cf. Sect. VI) is presented before the paper concludes in Sect. VII. To sum up, this paper provides the basis for systematically analyzing exceptions in process behavior for application scenarios with multiple process instances and their interdependencies spanning one or several process types.

II. Classification of Instance Spanning Exceptions

To address *RQ1*, a classification of instance spanning exceptions is provided, based on which, the manifestation of each

class in process execution logs is determined. Afterwards, minimal requirements enabling the discovery of instance spanning exception triggers and handling are determined.

A. Classification Based on Exception Handling

In order to discover instance spanning exceptions, we need to identify the trigger of the exception as well as the handling strategy. The handling strategy, in turn, is based on the exception type. In order to identify instance spanning exception types, we rely on i) literature, i.e., [7], [13] propose several exception handling strategies: ignoring, warning, retry, suspend/stop/resume, workflow recovery operations (e.g., backward recovery, forward recovery, alternative tasks, etc.), workflow modifications and evolution and ii) an extensive set of real-world instance spanning exception examples. In the following, we outline to which extent these strategies are reflected for the instance spanning setting based on [25] which presents a categorization of instance spanning constraints based on 114 real-world examples [19]. Around 14.1% of these examples deal with exception handling, i.e., these examples refer to instance spanning exceptions. From these examples, five distinct classes of instance spanning exception handling strategies can be identified: *wait*, *cancel*, *redo*, *change* and *rework*. This is inline with the exception handling strategies from literature as *wait* reflects ignoring, warning or suspend, *cancel* reflects stop, *redo* reflects retry, *change* reflects workflow modifications and partly workflow recovery operations and *rework* also the latter. Table I summarizes the exception classes denoted along their exception handling and characteristics in relation to the process execution behavior. A distinction is made between whether additional instances are spawned during exception handling, how instances subsequent to the triggering instance, i.e., the instance that caused the exception, are affected, whether new tasks, i.e., tasks that were not observed in the process execution log until the trigger, can show up, and whether data attributes could change, i.e., do not have the value expected based on previous observations. For illustration, real-world examples from [19] for each instance spanning exception class including their trigger and handling part are provided.

class	additional instances	subsequent instances	new tasks	data
wait	no	delay	no	no
cancel	no	sudden end	no	no
redo	no	iteration	no	no
change	no	concept/data drift	yes	yes
rework	yes	not defined	yes	yes

TABLE I
CLASSIFICATION OF INSTANCE SPANNING EXCEPTION HANDLING

Wait does not require to spawn new instances for handling the exception and instances subsequent to the trigger can have a delay in terms of time. Tasks that were not seen before and a change in data attributes cannot be involved in the trigger or handling part.

Example (wait) "Several planes are in the landing process. A plane that had a problem during the landing can affect the landing of the other planes."
Trigger one plane had a problem during the landing
Handling subsequent planes have to wait

Cancel reflects a cancellation procedure across multiple instances. No additional instances need to be spawned, but subsequent instances can show a sudden end. No new tasks and no changes in data elements are observable.

Example (cancel) "Several diagnosis for a car are running at the same time. If the problem is identified in one of the diagnosis, the others are cancelled."
Trigger problem identified in one of the diagnosis
Handling all other instances referring to the same car are cancelled immediately, i.e., a sudden end of instances related to the same car

Redo does not require additional instances to be spawned. Instances subsequent to the trigger contain iterations for one or multiple tasks. Neither any new tasks nor changes in data attributes can be observed.

Example (redo) "When one student (instance) has not understood a concept, the concept is taught to every student in class again."
Trigger one student has not understood a concept
Handling teach concept again, i.e., repeat tasks at least once for a batch of instances

Change does not require to spawn additional instances. Subsequent instances are affected resulting in either a concept drift, therefore new tasks can be observed, or a data drift, i.e., a change in one or multiple data attributes. The latter happens in the example (data element airport changes). Additionally, depending on the granularity of the process execution log, a concept drift might also be observable.

Example (change) "If weather conditions change, a transport plane might land at another airport leading to rebundling of cargo."
Trigger weather conditions change
Handling airport change and rebundling, i.e., instances contain unexpected value in data element reflecting the airport, new tasks reflecting the bundling can be observed

Rework requires to spawn additional instances and the behavior of subsequent instances is undefined.

Example (rework) "During maintenance the loss of a blade is detected. All instances produced since the last maintenance have to be searched until the blade is found."
Trigger a blade got lost
Handling spawning of search process for already finished instances; implicit wait/cancel for instances of the production process

B. Manifestation of Exceptions in Process Execution Logs

Table II summarizes requirements on process execution logs for discovering instance spanning exceptions along the classification presented in Tab. I ($\mapsto$ *RQ1*). Each of these requirements is part of the XES standard [23]. In particular, we require the concept and time extensions as well as lifecycle transitions, i.e., `start`, `complete`, `pi_abort`, and `reassign`. Lifecycle transition `pi_abort` indicates whether or not the execution is aborted for this case and `reassign` indicates an assignment after a withdrawal. Moreover, the BPAF lifecycle transactional model [16] is re-

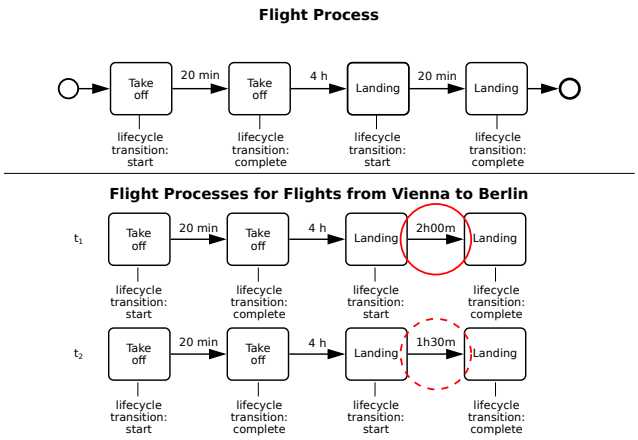

Fig. 1. **Wait:** A process reflecting a flight. A delay causing the exception occured in the second trace. A full circle reflects the trigger, a dashed circle the handling.

quired, since it distinguishes if a completed task has been executed successfully (`Completed.Success`) or failed (`Completed.Failed`).

class	concept name	instance uid	timestamp	lifecycle transition
wait	+	+	+	−
cancel	+	+	−	`pi_abort`, `complete`
redo	+	+	−	`Completed.Failed`, `Completed.Success`
change	+	+	−	`reassign`, `start`
rework	+	+	−	`pi_abort`

TABLE II
REQUIREMENTS ON PROCESS EXECUTION LOGS (+NECESSARY, −NOT NECESSARY)

General requirements. For every exception class, the process execution logs must contain a concept name and, in the case of process spanning settings, a unique identifier in order to be able to link related instances. How to identify and merge corresponding traces stemming from multiple process types is described in detail in Sect. III.

Wait. The duration of each task is analyzed. A task with a longer than expected duration can block an important resource, which is involved in other instances as well. In the example, several planes are using the same airport entry point. Hence, if one airplane takes longer to land, the following planes, i.e., other instances or processes, are affected and their landing takes longer than expected as well. Since delays in terms of time need to be discovered a process execution log must have timestamps whereas lifecycles do not play a crucial role, see Figure 1.

Cancel. The complete execution of a task is analyzed. To speed up a process, multiple sub-processes can be spawned, where the first sub-process that succeeds, cancels the others. In the example, several diagnostic processes are spawned to identify the problem of a car. When a diagnostic process completes successfully, the other diagnostic processes are canceled immediately. To discover cancel events, timestamps are not necessary, but a process execution log needs to have lifecycle transitions `pi_abort` and `complete`, see Figure 2.

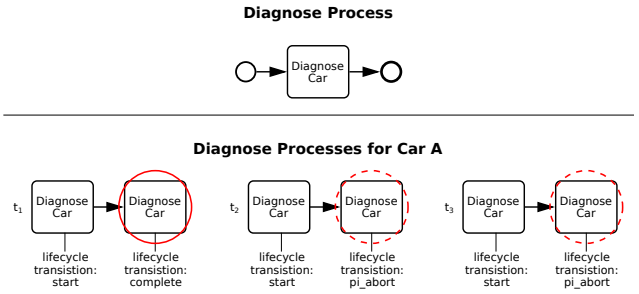

Fig. 2. **Cancel:** A process for diagnosing a car. Three instances are started, after the first diagnose process is successfully finished, the others are aborted. A full circle reflects the trigger, a dashed circle the handling.

Redo. The number of iterations of a task is analyzed. A requirement for the complete success of a process could be the successful completion of all related (sub)-processes or tasks. In the example, a course or seminar for students is only completed successfully if all participating students understood the concept of the course and passed it, but if at least one student fails the course, all students have to take the course again. For discovering iterations, we consequently require lifecycle transitions `Completed.Failed` as well as `Completed.Success`, see Figure 3.

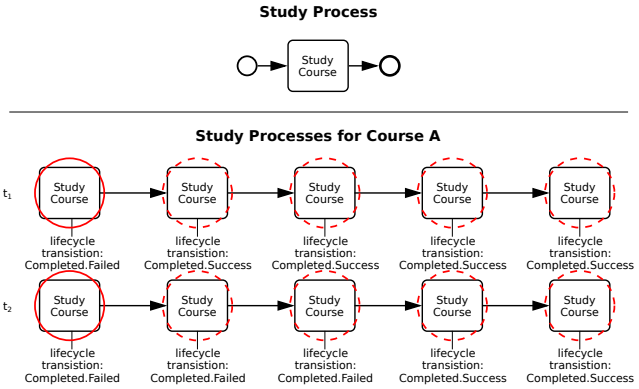

Fig. 3. **Redo:** A process for studying in class. Several instances are started, if one student fails the course, all others have to also repeat that course. A full circle reflects the trigger, a dashed circle the handling.

Change. We analyze the process tasks that occurred after the exception. If a resource that is shared between different processes and instances breaks down, future instances cannot use this shared resource and another one has to be used. In the example, due to weather conditions, an airplane cannot land at its destination, therefore another airport is selected. This can affect subsequent airplanes, until the airport, the original resource, is available again and can lead to rebundling of cargo. For discovering instance spanning exceptions of this class lifecycles `reassign` and `start` are necessary, see Figure 4.

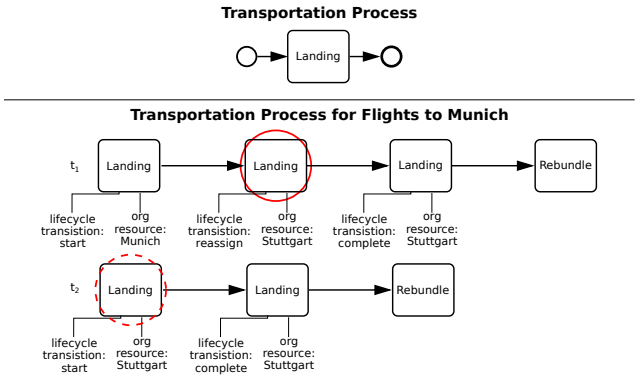

Fig. 4. **Change:** A process for transportation of goods. Several instances are started, if one airplane is redirected to another airport, subsequent airplanes can be affected and a rebundling of cargo can be necessary. A full circle reflects the trigger, a dashed circle the handling.

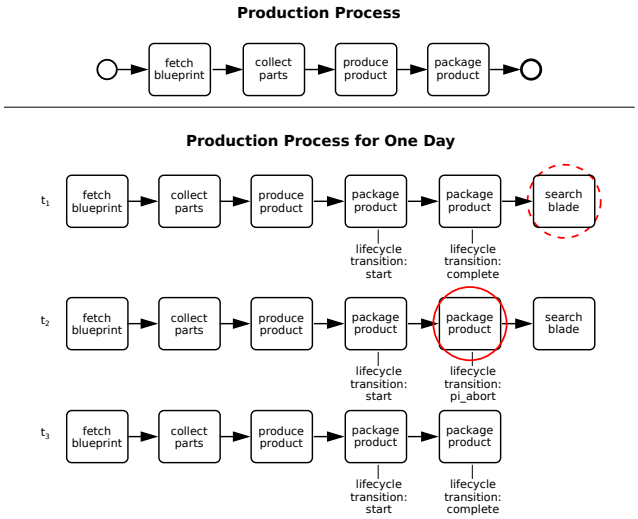

Fig. 5. **Rework:** A process for production of goods. A full circle reflects the trigger, a dashed circle the handling.

Rework. Tasks before an exception occurred are analyzed. If an error occurs in a process instance with an object or resource that is shared with different instances, the process instance with the error is stopped, the object or resource is repaired and previously executed instances from this batch have to be checked, see Figure 5.

III. Discovery Algorithms for Instance Spanning Exceptions

Based on the classification provided in Tab I and respecting the requirements set out in Tab. II, five algorithms for discovering instance spanning exceptions are provided in this section ($\mapsto$ *RQ2*). Each algorithm takes as input a set of process execution logs reflecting different process types. Consider a patient undergoing different treatments for which separate processes exist. As outlined in [25] in order to identify related instances, i.e., in the example those referring to the same patient, a data attribute which is ideally a unique identifier like a patient ID is required. This is reflected by function merge_traces(). This function is always called at the beginning of each algorithm. Based on a unique data attribute, related traces are merged into a single trace. As outlined in [25], if no unique attribute is available the merging can be carried out by more sophisticated techniques such as, e.g., [3]. The output for each algorithm consist of a list of trigger combined with handling events for each instance spanning exception that was discovered within the process execution log.

Wait. The exception trigger is identified by Alg. 1 based on events taking longer than expected. The handling part is represented by instances directly following the instances containing the trigger event, that also take longer than expected.

Algorithm 1 Wait Discovery Algorithm

Input: $logs$ = Log of different processes, th = Threshold for outlier detection

Output: $result$ = List of potential trigger and handling events for Wait

```
1:  traces = merge_traces(logs)
2:  calculate_temporal_information(traces)
3:  t_res = list()
4:  for trace in traces do
5:      for event in trace.events do
6:          if outlier(event) then
7:              trigger << event
8:              next
9:          end if
10:         if trigger and outlier(th,event) then
11:             handling << event
12:         end if
13:         if trigger and not outlier(th,event) then
14:             if handling then
15:                 t_res << [trigger,handling]
16:             end if
17:             trigger.clear
18:             handling.clear
19:         end if
20:     end for
21:     if trigger then
22:         t_res << [trigger,handling]
23:         results << t_res
24:         trigger.clear
25:         handling.clear
26:         t_res.clear
27:     end if
28: end for
```

In addition to the set of process execution logs, this algorithm takes as further input a threshold, $th \in [0; \infty[$ for the time-related outlier detection. We use the well-established z-score [4] for outlier detection and if it is above th the event is marked as outlier. For applying the z-score the temporal information (mean duration and standard standard deviation of a task) for all events is calculated in line 2. For each merged trace the algorithm starts with an empty list of trigger and handling events. The first event, that is classified as an outlier, is saved as a potential trigger event, lines 6-8. If the following event is still an outlier, it is saved as an exception handling event, lines 10-11. When the first event that is not classified as outlier is discovered, the trigger and handling events that are already discovered, are saved, lists are cleared, and the search for a new trigger event continues, lines 13-18. If there is only a potential trigger, without corresponding handling events, the potential trigger list is cleared, i.e., no wait exception is discovered.

Cancel. The trigger discovered by Alg. 2 is the task that reports completes successfully, i.e., contains the `complete`

lifecycle transition, while the tasks that are cancelled are classified as the handling for this exception.

Algorithm 2 Cancel Discovery Algorithm

Input: $logs$ = Log of different processes

Output: $result$ = List of potential trigger and handling for Cancel

```
1:  traces = merge_traces(logs)
2:  for trace in traces do
3:      for event in trace.events do
4:          if event.lifecycle == "pi_abort" then
5:              for potential in collect_events(event.name) do
6:                  if potential.lifecycle == "complete" then
7:                      trigger << potential.name
8:                  else
9:                      if potential.lifecycle == "pi_abort" then
10:                         handling << potential.name
11:                     end if
12:                 end if
13:             end for
14:             if trigger and handling then
15:                 result << [trigger,handling]
16:             end if
17:         end if
18:     end for
19: end for
```

After merging related traces, for all events in all traces, the lifecycle status is checked. The first event containing lifecycle transition `pi_abort`, starts the exception handling discovery, line 4. Since this algorithm is searching for tasks operating on the same resources, only tasks with such a resource are merged, while the others are discarded, line 5. Then, every event related to the first event with `pi_abort` of this trace with a `complete` lifecycle is classified as a trigger, line 6, while events with `pi_abort` as a lifecycle are classified as the handling for this exception, line 9. Other events, not operating on the same resource, i.e., not the same car, are not considered. At the end of each trace, the discovered trigger and handling events are saved into the result, line 14.

Redo. Algorithm 3 discovers the trigger as the task that is completed, but still failed, while batches of tasks that are discovered afterwards are classified as handling events. For the discovery of the trigger and handling events, the algorithm takes only the events into account that contain the outcome of a specific task, i.e., if it has been completed successfully or not. This is done for identifying iterations of a task. If an event is discovered with a failed execution, identified by the lifecycle status `Completed.Failed`, the remaining events are split into iterations, i.e., one completing event per original process per iteration, lines 4-6. If only successful events are discovered in the last iteration, the exception class can be identified and the iterations after the first one are considered the handling of this exception, while the first failed execution is saved as the trigger, lines 7-8. The results are collected in line 11.

Change. Algorithm 4 discovers the exception trigger as the task that changes its assigned resource to another one, e.g., the designed airport is not available anymore, thus another airport is assigned in an event. The related handling tasks are the following instances with a task using the new resource in the beginning instead of the original resource.

If a resource is reassigned to another resource, a potential trigger is discovered, lines 6-8. For all following events,

Algorithm 3 Redo Discovery Algorithm

Input: $logs$ = Log of different processes

Output: $result$ = List of potential trigger and handling events for Redo

Require: BPAF lifecycle transactional model

```
1:  traces = merge_traces(logs)
2:  for trace in traces do
3:      for event in trace.events do
4:          if event.lifecycle == "Completed.Failed" then
5:              trigger = event
6:              iterations = collect_iterations(trigger)
7:              if (set(iterations.last.lifecycles) == set("Completed.Success") then
8:                  handling << iterations.events
9:              end if
10:             if handling then
11:                 result << [trigger,handling]
12:             end if
13:         end if
14:     end for
15: end for
```

that potentially use the same original resource, the assigned resource is checked for events at the start of a task execution. If it equals the new resource from the potential trigger event, the event is considered as a handling for this exception, lines 12-13. If a handling is discovered, the results are saved and appended in line 17.

Algorithm 4 Change Discovery Algorithm

Input: $logs$ = Log of different processes

Output: $result$ = List of potential trigger and handling events for Change

Require: org:resource for potential trigger

```
1:  traces = merge_traces(logs)
2:  for trace in traces do
3:      for event in trace.events do
4:          if event.lifecycle == "reassign" then
5:              o_res = event.resource
6:              n_res =find_start_resource(event)
7:              if o_res!=n_res then
8:                  trigger=event
9:                  next
10:             end if
11:         end if
12:         if trigger == event and event.lifecycle == "start" and event.resource==o_res then
13:             handling << event
14:         end if
15:     end for
16:     if handling.size then
17:         result << [trigger, handling]
18:     end if
19: end for
```

Rework. The trigger discovered by Alg. 5 is the task that is aborted and stops the instance, discovered by the `pi_abort` lifecycle transition. In the previously executed instances, additional tasks can then be discovered, reflecting the handling for this exception. For discovering this instance spanning exception class, the lifecycle status of the events is analyzed. If an event is discovered containing the lifecycle status `pi_abort`, a potential trigger event is saved, lines 4-5. Afterwards, the instances that have been previously executed are analyzed, line 6. If additional events are discovered in these instances, the events are saved as handling for this exception, lines 7-8. The additional events could be discovered using conformance checking algorithms, but this requires the discovery of a process model before. Since the order of the additional events

is not important, the number and content of events can be counted instead. If handling events are discovered, the results are saved in line 12.

Algorithm 5 Rework Discovery Algorithm

Input: $logs$ = Log of different processes

Output: $result$ = List of potential trigger and handling events for Rework

```
1:  traces = merge_traces(logs)
2:  for trace in traces do
3:      for event in trace.events do
4:          if event.lifecycle == "pi_abort" then
5:              trigger << event.instance
6:              for pi in previous_instances(trigger) do
7:                  if additional_events(pi,trigger) then
8:                      handling << additional_events(pi,trigger)
9:                  end if
10:             end for
11:             if handling.size then
12:                 result << [trigger,handling]
13:             end if
14:         end if
15:     end for
16: end for
```

IV. Evaluation

Algorithms 1 to 5 have been prototypically implemented and evaluated on synthetic logs and on a real-world log from the public transport domain. Data sets and implementation are available at http://gruppe.wst.univie.ac.at/projects/crisp/data_sets/.

A. Evaluation on Synthetic Data Sets

Each of the synthetic data sets has been generated based on the corresponding example presented in Sect. II to show the feasibility of the algorithms.

Wait. For Alg. 1, the data set consists of 10 transportation processes for different airplanes using the same landing platform saved as data element `shipment` and reflecting the id of the air strip. Each process consists of 200 traces where at least one trace of each process shares the landing platform with at least 9 other traces. For a crisp outlier detection, the duration of correctly executed instances is set to exactly 20 minutes. After an outlier happened, i.e., an airplane has been delayed, the subsequently arriving airplanes are delayed by an ever smaller margin until the planes fly correctly again. For the outlier detection we use a threshold th of 2 for the z-score, since the delay is rather small, but an impact on the following instances has been discovered. The algorithm discovers a trigger event, taking 20 minutes longer than expected, yielding a z-score of 3.4. The algorithm discovers the following events as handling until z-score < 2 holds.

Cancel. For Alg. 2, the data set consists of 4 diagnostic processes each reflecting different diagnostic systems of a car. All of them can be executed in parallel. Hence there are 4 traces for one car. The traces for each car, are identified by the data element `car_number`. In this data set, all diagnostic processes for one car stop immediately after one diagnostic process reports successfully back. There are process instances for 500 cars, i.e., 2000 process instances in total. The algorithm discovers a handling event first, by discovering the abortion of an event. Afterwards the trigger event is discovered as the only completed event, while the others are considered handling events.

Redo. For Alg. 3, the data set consists of 5 different processes, where each process reflects a student taking part in a course. Students taking the same course can be identified using the data element `course_number`. A process instance ends, if all participating students complete the course successfully. In this data set the number of failing students varies per iteration of attending the course, e.g., Student A completes the course successfully in the first iteration and Student B fails and the other way round in the second iteration. To discover the iterations, it is important that the `BPAF` lifecycle transactional model is used. Only the end event of a study process is required for this algorithm, e.g., the event with lifecycle transition `Completed.Success` and `Completed.Failed` to support the discovery of iterations. The data set consists of 200 instances per process.

Change. For Alg. 4, the data set consists of 10 processes. Instances of these processes reflect parts of a shipment to a specific airport. 150 instances of all processes contain no exceptions. In 50 instances, the airport is not available and the trigger instance and all following instances of this shipment are redirected to another airport. The algorithm identifies the reassignment of the airport resource and discovers the handling events, until the original resource is used again.

Rework. For Alg. 5, the data set consists of 10 processes, each process containing 200 traces. All processes are performed on the same machine. Process instances, combined by the data element `product_number` reflect a production batch on one machine, i.e., one process instance of each process. The first 150 instances of each process are performed without any exceptions to collect information on the processes. The remaining 50 instances of each process contain one instance reflecting the discovery of a missing blade of the machine. The machine is then repaired and all instances for this product that have been executed prior to the trigger instance, contain additional events, e.g., `Retrieve Product`.

B. Real-world Data Set – Public Transport Domain

Data Collection and Preparation. A real-world data set based on the Vienna open data platform was gathered by monitoring the public transportation by the Wiener Linien[1] between 8th of May and 10th of May 2021. The data gathering focused on the Tram lines 37, 38, 40, 41 and 42 since these have several stations in common, e.g., "Schottentor" which is their start as well as end point. The shared station of different lines should increase the chance of detecting an exception class. The collected data consists of two types of JSON response files and a request is sent every 60 seconds.

The first JSON response file contains general traffic infos on the selected lines (37,38,40,41,42), i.e., if an interruption occurs this is logged within a JSON file.

[1] https://www.data.gv.at/katalog/dataset/wiener-linien-echtzeitdaten-via-datendrehscheibe-wien

Within the responses there are several relevant fields: i) `trafficInfos.time.start` containing the start time of the interruption, ii) `trafficInfos.time.resume` containing the resume of the transport during an interruption, iii) `trafficInfos.time.end` containing the end time of the interruption, iv) `trafficInfos.title` and `trafficInfos.description` containing a (detailed) description of the interruption, e.g., a track damage.

The second type of JSON response files contains data on real time monitoring of specific stations. The data set contains information on the planned departure time and the real departure time of vehicles from a tram line from a station. This data set can be transformed into a suitable data set in two ways. The first one, compares the departure times with the planned departure times to recreate the route different vehicles have taken and view each vehicle as a process instance. The second way, interprets dispatching vehicles from a station as an activity and creates a process instance for each departure with the next station as a data element attached to it. The latter option is chosen, since unfortunately some lines are stopping at specific stations multiple times during one round and without a vehicle id, it cannot be guaranteed that the vehicles are transformed correctly all the time. Since the line number is available for each departure, the second option for the transformation yields a suitable data set.

Results. This data set is not using the BPAF lifecycle transactional model and no departures are performed as long as all departures complete successfully based on the setting, thus the *redo* exception handling class is very unlikely. Vehicles in a tram line are not stopping if other vehicles reach their destination earlier, hence, the exception class *cancel* is not available in this data set. Traffic jams or other obstructions on the track are typically affecting subsequent departures of vehicles and not previous departures. Thus the exception class *rework* is unlikely, but *change* should be detectable. A late vehicle or a delay in a departure in one station, can affect the departure of later vehicles from the the same of different lines, thus the exception class *wait* seems likely.

The destination of a dispatched vehicle is transformed to the resource of a dispatching event to detect reassignments of the target station. Data entries from the first data set, containing the keyword "umgeleitet" have been transformed to a reassignment of a resource to the lines mentioned in the data entry. This allowed us, to discover a *change* exception for line 40. The trigger event is discovered on the 9th of May at 13:32, a reassignment to "Michelbeuern U AKH", a stop not planned in this line. The following instance is using the same resource, at 13:42, therefore discovered as the handling for this exception. Instances following this, are already using the original resource again.

Since a time plan for a public transportation route does not have the same intervals during the whole day, we have opted to use the planned departure time as the starting point of the dispatching task and the real departure time as the end point. Thus the algorithm for discovering the *wait* class can easily be applied without any adjustments, as well. The used data sets are process instances of the station "Schwarzspanierstraße", where 5 different lines are stopping and consists of 3098 dispatched vehicles. With a threshold of 3 standard deviations of the mean dispatch time, a trigger event is discovered for Line 41 at 15:20 on the 9th of May, with a delay in the following vehicle as a handling event to this trigger, as well for Line 41 at 15:21. By lowering the threshold th to 1.5 for the z-score, additional wait class exceptions are discovered for line 38.

To sum up, Algorithms 1 to 5 correctly discover the exception trigger and handling events based on the synthetic log files. The real-world data set provided process logs where two out of five exception classes were present and also discovered.

V. Discussion

The evaluation demonstrated the feasibility and applicability of Algorithms 1 to 5 for instance spanning exception discovery as presented in Sect. III. The algorithms require neither any additional information on process models nor domain knowledge, only that the data set fulfills the requirements set out in Sect. II-B. These data set requirements concern the existence of certain lifecycle transition events which correspond to definitions from the XES standard [23] combined with the BPAF lifecycle transaction model. Note that the BPAF lifecycle is only required for the *redo* exception class.

However, the requirements can be altered if sufficient domain knowledge is present, e.g., in a production process a task taking only a few milliseconds instead of hours, can be identified as faulty execution reflecting an abortion of a process, even though the lifecycle status data elements are not reflecting this. For Alg. 3 for example, a simple data element providing the result of a course can be used as well instead of the `BPAF` lifecycle transactional model. In future work, an adaption of the requirements for each algorithm by exploiting domain knowledge will be investigated in more detail. The goal is to analyze options for relaxing the requirements on process execution logs.

VI. Related Work

In a broader sense, this work can be positioned in the area of process exception handling and process flexibility [17]. "*Instance-specific changes* are often applied in an ad-hoc manner and become necessary in conjunction with real-world exceptions" [18]. This means, that process changes are an instrument to deal with *unexpected* exceptions [1], i.e., exceptions that occur during runtime and were not foreseen when designing the process. *Expected* exceptions, in turn, can be treated in the process model by possibly infrequent paths of an alternative branch (decision) [5] or by a predefined exception handling pattern [1], [20]. The work at hand abstracts from the distinction into expected or unexpected exceptions by looking at the process execution log, i.e., at what has actually happened, in an ex post way. Cleary, if exception handling classes *change* and possibly also *rework* are detected, one can conclude that an unexpected exception has occurred

where classes *wait*, *cancel*, and *redo* might hint towards an expected exception. We will investigate this in future work, but at this point it becomes already clear that the detection of the instance-spanning exceptions supports the definition of handling strategies for previously unexpected exceptions.

In the following, we discuss related approaches in more detail. In [20] a framework for exception handling based on workflow patterns is presented and five types of exceptions *work item failure*, *deadline expiry*, *resource unavailability*, *external trigger* and *constraint violation* were identified. However, their aim is not at defining or discovering ISC exceptions from process execution logs as it is outlined in this paper. Approaches on exception and deviation mining, e.g., [9], [22] have not considered instance spanning exceptions yet. Anomalies and concept drifts can be caused by exceptions and detected from logs, but this requires root cause analysis which is rarely provided. Anomaly detection, e.g., [2], [8], and concept drift detection [14], [21] does moreover not include a distinction in trigger and handling part none of those approaches explicitly considers instance spanning anomalies and drifts. Approaches such as [10], [12] support class *cancel*, but the instance spanning behavior is still not covered. Instance spanning behavior in process mining has been considered by discovering constraints [25] and batch processing behavior [15] from process execution logs. Yet, none of these works provides full support for the detection of instance spanning exceptions.

VII. Conclusion

Exceptions in process execution can occur frequently and exceptions spanning multiple instances are of particular concern. Within this paper we presented a classification of instance spanning exceptions based on literature as well as an extensive set of real-world instance spanning exception examples resulting in five distinct instance spanning exception classes, *wait*, *cancel*, *redo*, *change* and *rework*. For each class we captured how it manifests in process execution logs and elicited minimal requirements enabling their detection from process execution logs. Based on these findings one algorithm per class was developed and evaluated on synthetic as well as real-world data sets. Evaluation results on synthetic log files demonstrated the feasibility and the real-world data set from the public transport domain the applicability of each algorithm. As future work, an adaption of the requirements on log files in order to extend the algorithms to process execution logs containing less information on lifecycles is investigated. Moreover, the exploitation of domain knowledge to relax the requirements will be investigated.

References

[1] Adams, M., ter Hofstede, A.H.M., van der Aalst, W.M.P., Edmond, D.: Dynamic, extensible and context-aware exception handling for workflows. In: On the Move to Meaningful Internet Systems. pp. 95–112 (2007). https://doi.org/10.1007/978-3-540-76848-7_8

[2] Böhmer, K., Rinderle-Ma, S.: Mining association rules for anomaly detection in dynamic process runtime behavior and explaining the root cause to users. Inf. Syst. **90**, 101438 (2020). https://doi.org/10.1016/j.is.2019.101438

[3] Claes, J., Poels, G.: Merging event logs for process mining: A rule based merging method and rule suggestion algorithm. Expert Syst. Appl. **41**(16), 7291–7306 (2014)

[4] Crocker, L., Algina, J.: Introduction to classical and modern test theory. ERIC (1986)

[5] Curbera, F., Khalaf, R., Leymann, F., Weerawarana, S.: Exception handling in the BPEL4WS language. In: Business Process Management. pp. 276–290 (2003). https://doi.org/10.1007/3-540-44895-0_19

[6] Dijkman, R.M., Türetken, O., van IJzendoorn, G., de Vries, M.: Business processes exceptions in relation to operational performance. Bus. Process. Manag. J. **25**(5), 908–922 (2019). https://doi.org/10.1108/BPMJ-07-2017-0184

[7] Eder, J., Liebhart, W.: Contributions to exception handling in workflow management. In: EDBT Workshops. pp. 3–10 (1998)

[8] Genga, L., Alizadeh, M., Potena, D., Diamantini, C., Zannone, N.: Discovering anomalous frequent patterns from partially ordered event logs. J. Intell. Inf. Syst. **51**(2), 257–300 (2018). https://doi.org/10.1007/s10844-018-0501-z

[9] Grigori, D., Casati, F., Dayal, U., Shan, M.: Improving business process quality through exception understanding, prediction, and prevention. In: VLDB 2001. pp. 159–168 (2001)

[10] Kalenkova, A.A., Lomazova, I.A.: Discovery of cancellation regions within process mining techniques. Fundam. Informaticae **133**(2-3), 197–209 (2014). https://doi.org/10.3233/FI-2014-1071

[11] Laznik, J., Juric, M.B.: Context aware exception handling in business process execution language. Inf. Softw. Technol. **55**(10), 1751–1766 (2013). https://doi.org/10.1016/j.infsof.2013.04.001

[12] Leemans, M., van der Aalst, W.M.P.: Modeling and discovering cancelation behavior. In: On the Move to Meaningful Internet Systems. pp. 93–113 (2017). https://doi.org/10.1007/978-3-319-69462-7_8

[13] Luo, Z., Sheth, A.P., Kochut, K.J., Miller, J.A.: Exception handling in workflow systems. Appl. Intell. **13**(2), 125–147 (2000)

[14] Maggi, F.M., Burattin, A., Cimitile, M., Sperduti, A.: Online process discovery to detect concept drifts in ltl-based declarative process models. In: On the Move to Meaningful Internet Systems. pp. 94–111 (2013). https://doi.org/10.1007/978-3-642-41030-7_7

[15] Martin, N., Pufahl, L., Mannhardt, F.: Detection of batch activities from event logs. Inf. Syst. **95**, 101642 (2021). https://doi.org/10.1016/j.is.2020.101642

[16] zur Muehlen, M., Swenson, K.D.: BPAF: A standard for the interchange of process analytics data. In: Business Process Management Workshops. pp. 170–181 (2010). https://doi.org/10.1007/978-3-642-20511-8_15

[17] Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012). https://doi.org/10.1007/978-3-642-30409-5, https://doi.org/10.1007/978-3-642-30409-5

[18] Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems - a survey. Data Knowl. Eng. **50**(1), 9–34 (2004). https://doi.org/10.1016/j.datak.2004.01.002

[19] Rinderle-Ma, S., Gall, M., Fdhila, W., Mangler, J., Indiono, C.: Collecting examples for instance-spanning constraints (2016), http://arxiv.org/abs/1603.01523

[20] Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow exception patterns. In: Advanced Information Systems Engineering. pp. 288–302 (2006). https://doi.org/10.1007/11767138_20

[21] Stertz, F., Rinderle-Ma, S., Mangler, J.: Analyzing process concept drifts based on sensor event streams during runtime. In: Business Process Management. pp. 202–219 (2020). https://doi.org/10.1007/978-3-030-58666-9_12

[22] Tsoury, A., Soffer, P., Reinhartz-Berger, I.: How well did it recover? impact-aware conformance checking. Computing **103**(1), 3–27 (2021). https://doi.org/10.1007/s00607-020-00857-y

[23] Verbeek, H.M.W., Buijs, J.C.A.M., Dongen, B.F., Aalst, W.M.P.: XES, XESame, and ProM 6. In: Information Systems Evolution, vol. 72, pp. 60–75 (2011)

[24] Winter, K., Rinderle-Ma, S.: Discovering instance-spanning constraints from process execution logs based on classification techniques. In: Enterprise Distributed Object Computing Conference. pp. 79–88 (2017)

[25] Winter, K., Stertz, F., Rinderle-Ma, S.: Discovering instance and process spanning constraints from process execution logs. Inf. Syst. **89**, 101484 (2020). https://doi.org/10.1016/j.is.2019.101484