
Production Systems and Information Engineering
Volume 10 (2), pp. 52–63
doi: 10.32968/psaie.2022.2.5.

EVENT SEQUENCE SEGMENTATION FOR PARALLEL PROCESSES

László Kovács
University of Miskolc, Hungary
Institute of Information Technology
kovacs@iit.uni-miskolc.hu

Dávid Polonkai
University of Miskolc, Hungary
Institute of Information Technology
david.polonkai2@gmail.com

Abstract. The robotic process mining focuses on the analysis of historical process sequences in order to build up a process model for the investigated field. One of the main tasks in robotic process mining is the construction of process schema for the input sequences. Usual methods are able to generate models using only baseline graph structures. In order to support high level structures like parallelism, the input event sequence structure must support additional attributes on the events. This paper presents a novel approach on sequence segmentation providing an intermediate graph structure which can be used to mine complex graph patterns. The tested prototype system contains a Python-based implementation of the proposed algorithm. In the paper, some tests are shown to illustrate the suitability of the proposed model.

Keywords: robotic process mining, RPM, event sequences, even graph

1. Introduction

The automation of office workflows is a key area in the development of smart administration engines. Our investigation focuses on a specific problem domain, where the goal is to build up an accurate model of the underlying training processes. In this case, we assume that individual processes are given with event sequences and the integration, generalization of these event sequences generates an event graph.

Robotic Process Mining (RPM) [1] is a new research area aiming to explore automated processes with machine learning tools. Due to the large complexity

of event processes, the development of process mining methods is a largely unexplored issue. Several proposals have been made in the literature, but their effectiveness is still relatively low. The following methods can be highlighted in this area. One approach is the analysis of textual process specification. The problematic part of this method is the inaccuracy of the interpretation of natural language texts [2]. The second way is the form-based analysis in which employees fill out a data sheet about the workflows performed. The difficulty here stems from the partial non-automation of the process [3]. The third approach is the automatic event exploration [4]. This method is considered to be the most promising approach.

In the case of complex event graph structures, the appropriate segmentation of the incoming sequences can improve the efficiency of the schema mining algorithms. In this paper, we present a novel approach to perform process segmentation that can also be used also for event graphs containing parallel segments.

2. Background survey

Within the framework of automated process exploration, the following main steps can be distinguished [5]:

1. Data collection. The data used to describe and characterize the processes can come from several different and very different sources.
2. Data preparation. During data preparation, several processing steps may be required to ensure adequate data quality and quantity. The most common processing operations are: Data format transformation, Data structure conversion, . Data value conversion, . Data cleaning, Data reduction.
3. Data analysis. Based on the compiled teaching samples, the goal is to create the operating model that best fits the samples. The model is usually defined using statistical and machine learning methods. The most commonly used methods are: Pattern matching , Exploration of common and rare patterns, clusters, . Neural network based classification, . Process forecasting.

The most important source of data retrieval is the event log. Log-based process exploration is an area that has been slowly being explored for ten years. The main shortcomings of previous models [6] are: 1) unique, complex algorithms, 2) inadequately tuned models, or 3) under-learning or over-learning models. The development of effective models remains an important and active research goal.

The accuracy and efficiency of the different schema mining tools depends on the quality of the input event sequences. In practical cases, the investigated process includes several actors and several sub-processes. The event graph consists of the integration of these sub-processes where several dependency types can be discovered among the sub-processes. According to the YAWL standard [7], there are three main types of join nodes (see Fig. 1):

- OR-join
- XOR-join
- AND-join

In order to identify these control nodes in the schema, the input log sequences should contain appropriate information about these synchronization points.

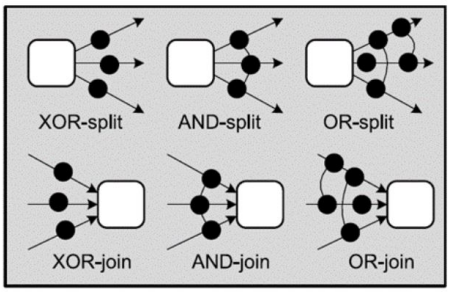

Figure 1. Control nodes in process graphs [1]

Regarding the event-log files, the key format is defined by the XES [8] standard. An XES document is actually an XML file that describes an event log, which can contain any number of traces, where a trace is a sequence of events. Both traces and events can have any number of attributes that can even be nested. There are five basic types defined by the standard: String, Date, Int, Float, and Boolean, which correspond to the following standard XML types: xs: string, xs: dateTime, xs: long, xs: double, and xs: boolean.

```
<trace>
        <date key="startDate" value="2021-04-14T00:00:00+02:00"/>
        <event>
                <string key="monitoringResource" value="7770681"/>
                <string key="concept:name" value="Purchase Item A"/>
                <string key="action_code" value="01_KIO8_011"/>
                <date key="time:timestamp" value="2021-04-14T00:00:00+02:00"/>
        </event>
```

Figure 2. Sample XES file

In addition, the model can be extended with any additional attributes (extensions). The XES standard does not make the use of any attribute mandatory. However, we can specify this in the global attribute list for events and traces at the beginning of the event log. When processing event logs, the following must be observed:

- Correlation and consistency. Ensuring consistency and correlation of events from different related systems [9].
- Chronology. When we combine data from different systems, we want to arrange the events based on the timestamp.
- Completeness. An event log may not contain a complete process
- Validity. Because a lot of data comes from different source systems, it is difficult to decide which of these are relevant and which are not; which should be included in the event log and which should not.
- Detail. In practice, the detail of event log events and user-relevant activities often do not match.

In the case of parallel subsequences, the control flow is split into two or more subsequences where each subsequent must have different actors. This split event is symbolized with an AND-split node in the event graph symbol systems. The parallel segments are merged into new segments at the AND-join nodes.

In the next sections, the possible errors in the input event logs are investigated and we present a method on how to locate the parallel and sequential segments in the incomplete event logs.

3. Event log for parallel sub-processes

In the standard event-log model, en event entry is a tuple of the following attributes [10]:

- tr: trace id
- id: event id
- tp: event type (class of the event)
- ti: timestamp
- ac: actor, agent
- co: cost

Considering only simple sequences, chain structure, we can use two parameters, the (timestamp, actor) tuple to segment the trace into sub-sequences. In this case, the following algorithm can be applied:

- sort the events by timestamp;

- segment the sequence by actors, every segment contains events belonging to the same actor.

It can be shown that in complex cases, this structure is not enough to determine the corresponding eventgraph. For example, the trace id will identify the whole process of several agents, thus the trace id is not suitable to separate the sub-processes. Taking the complex event graphs with parallel subprocesses, the following shortcomings can be identified:

1. In this case, the parallel sub-processes are belong to different actors, thus, these sub-processes overlap each others.

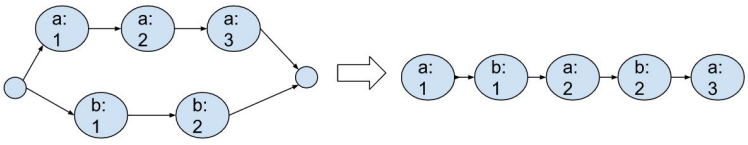

Figure 3. Example for false schema detection

2. There may exist false sequence detection. In this case, having two parallel sub-processes, if these processes have short execution times, it may happen that one process terminates before the other starts. Thus the log contains a sequence, but the real event graph contains a parallelism.

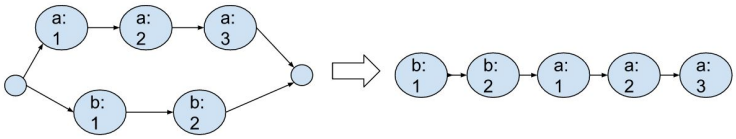

Figure 4. Example for false schema detection

3. Having a parallel (AND) split, it is hard to locate the end control node (AND join), as the sub-sequence after the AND-join may seem to be the next section of one of the parallel branches.
4. Having a nested parallelism, when one branch is split again into two parts, it is not possible to determine the correct control borders.

Based on the presented ambiguities, the presented event model alone is suitable for complex event mining. In order to upgrade the event log models for supporting the discovery of complex event graphs, we introduce some new information items into the event records.

4. Segmentation methods for parallel sub-processes

In order to determine the parallel sub-processes, we use a dependency relationship among the sub-processes based on the items (artifacts,products of the processing). An item here is the object, the target of the processing, like a mobile phone in the production line, or a document in the office records management. In this view, every event (processing step) is assigned to item sets (which may also empty):

- input items
- output items

We assume that each item is identified with an item type. Having these event properties, we can construct a dependency graph, where

- the nodes corresponds to the events
- there is a directed edge from event A to event, only if the intersection of the input itemset for B with the output itemset of A is not empty.

According to this dependency graph, if events e1 and e2 are not connected then e2 can not be an adjacent member of the event e1 in the schema graph.

The application of item-level dependency requires the extension of the feature attributes with the two new item sets:

- tr: trace id
- id: event id
- tp: event type (class of the event)
- ti: timestamp
- tl: duration, time length
- ac: actor, agent
- co: cost
- input items
- output items

We argue that this kind of extension is a natural requirement which is based on information usually available in the production systems. In the real application cases, the properties of the artifact items are important elements of the technical requirements.

In the event graph description, we introduce a segment unit, which denotes the sequence of events where

- the actor is the same
- the output of an event is equal to the input of the next event
- has a maximal length

Based on the proposed dependency graph, the corresponding segmentation mining algorithm consists of the following elements.

1. split every event into two events, a opening and a closing part; the set of input items is assigned to the opening part, the output set is related to the closing part
2. ordering the events by the related timestamp
3. the first event is the start event, the last one is an stop event
4. process the events in the timestamp order
5. initially we take the start event and we take an empty set of active items, and a set of idle items containing all items
6. at every processing step:
   (a) take the first open event where all of the input items are in the idle set
   (b) move these items from the idle set to the set of active items
   (c) connect this event with those events using an adjacency link where the output contains some items from the current input. Thus we assign for input item a unique sender event
   (d) at closing event we move the output items into the set of idle items
   (e) if two connected events have different actors, a new synchronization node is inserted between the events. The related synchronization nodes are merged into a single synchronization node.

It follows from the mentioned properties that every segment connects two synchronization nodes in the event schema graph.

5. Segmentation implementation

In the test framework, we used the Python programming language for implementation of the proposed segmentation algorithm. The main benefits of the selected programming language is the rich library pool on data management and data analysis. The implemented framework includes the following modules:

- core segmentation module
- input sequence generation
- graph visualization module

The sequence generation unit provides a description language which is based on the following building blocks:

- event_types
- event sequences

The eventtype class has four key parameters:

- type id
- duration time
- input artifacts, objects
- output artifacts, objects

Example for sample sequence:

```
e = EventSequence()

et1 = EventType(1,2,Ain=[],Aout=[1])
et2 = EventType(2,4,Ain=[1],Aout=[1])
et3 = EventType(3,2,Ain=[],Aout=[2])
...
e.add_event(Event(1,"a",et1))
e.add_event(Event(3,"b",et3))
e.add_event(Event(2,"a",et2))
e.add_event(Event(4,"b",et4))
```

The sequence discovery unit performs a processing loop on the sequence items. For each item, it will perform an adjacency matching using more aspects like actors, objects and time factors. The simplified code is presented in the following list.

```
def discover (Act, Log):

    jnodes_c = []
    for i in range(len(Log)):  # loop on the vents in the sequence
        ac = Act[ev.action]     # get event type
        out = ac.outputs.copy()       # get output objects (E1)
        for j in range(i+1,len(Log)): # loop on remaining events
            ev2 = Log[j]
            ac2 = Act[ev2.action] # candidate adjacent event
            in2 = ac2.inputs           # get input objects of E2
            z = in2.intersection(out)  # list of shared objects
            if len(z) > 0:   # if there are shared objects
                if ev.agent !=  ev2.agent:  # if different agents,
                            # insert a new synchronization event
                out = out - in2       # adjust the list of free
    while (len(jnodes_c) > 0):  # merging the atomic sync events
        xd = []
        X1 = set([jnodes_c[0][0]]) # process current sync event
        X2 = set([jnodes_c[0][2]])
```

```
        t1 = jnodes_c[0][5]
        t2 = jnodes_c[0][6]
        xd.append(0)
        for j in range(1,len(jnodes_c)): # compare other events
            fnd = 0
            if jnodes_c[j][0] in X1:
                X2.add (jnodes_c[j][2])
                fnd = 1
            if jnodes_c[j][2] in X2:
                X1.add (jnodes_c[j][0])
                fnd = 1
            if fnd ==1: # if they are matching process them
                xd.append(j)
                if t1 < jnodes_c[j][5]:
                    t1 = jnodes_c[j][5]
                if t2 > jnodes_c[j][6]:
                    t2 = jnodes_c[j][6]
                            # determine the merged sync events
        XX1 = [(x,Act[Log[x].action].name) for x in X1]
        XX2 = [(x,Act[Log[x].action].name) for x in X2]
```

The merged schema graph can be visualized to show and analyze the resulting output structure. Fig.5 presents a sample output schema showing four sync events and four segments with three actors (a,b,c,).

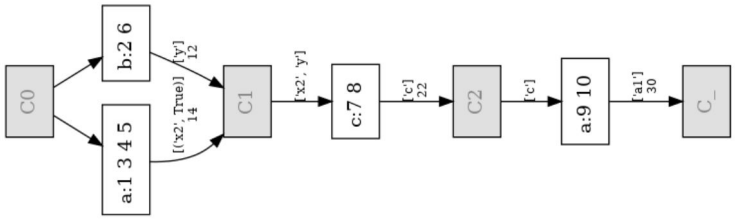

Figure 5. Generated base schema graph

Example 1 We construct here an event sequence containing collaboration of more actors and having parallel execution segments. The training set is defined with the following commands:

```
e=EventSequence()
et1 = EventType(1,2,Ain=[],Aout=[1])
et2 = EventType(2,4,Ain=[1],Aout=[1])
```

```
et3 = EventType(3,2,Ain=[],Aout=[2])
et4 = EventType(4,5,Ain=[2],Aout=[2])
et5 = EventType(5,4,Ain=[1,2],Aout=[3])
et6 = EventType(6,3,Ain=[1,2],Aout=[4])
et7 = EventType(7,1,Ain=[3],Aout=[5])
et8 = EventType(8,1,Ain=[4],Aout=[6])
et9 = EventType(9,4,Ain=[5,6],Aout=[7])
et10 = EventType(10,3,Ain=[7],Aout=[])

e.add_event(Event(1,"a",et1))
e.add_event(Event(3,"b",et3))
e.add_event(Event(2,"a",et2))
e.add_event(Event(4,"b",et4))
e.add_event(Event(5,"c",et5))
e.add_event(Event(6,"a",et6))
e.add_event(Event(7,"c",et7))
e.add_event(Event(8,"a",et8))
e.add_event(Event(9,"b",et9))
e.add_event(Event(10,"b",et10))
```

After processing the constructed complex event sequence , the segmentation algorithm will generate the model given in Fig. 6 for the sequence.

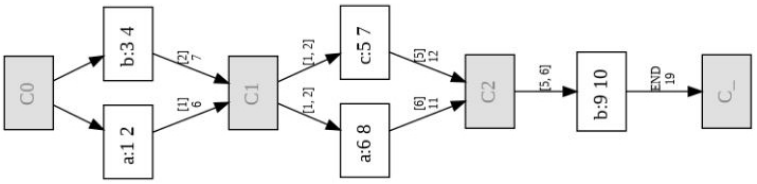

Figure 6. Generated schema result for Example 1

As we can see the segmentation algorithm discovered the parallel segments and the required synchronization nodes. Thus this compound sequence can be used as a member of a training set to construct complex event schema graphs in the next phase of the schema mining process.

6. Conclusion

This paper presents a novel approach on sequence segmentation providing an intermediate graph structure which can be used to mine complex graph patterns. The proposed model requires the availability of the object, artifact

level dependencies among the different event types. The detection of the parallel segments is based on this kind of relationship. Based on the discovered segments, the engine constructs a complex event schema graph, which may contain AND-join and AND-split synchronization nodes too. These schema graphs can be later merged into the next level schema graph where also XOR-nodes appear to show optional segments. The tested prototype system contains a Python-based implementation of the proposed algorithm. The performed tests show the suitability of the proposed model in mining of complex event schema graphs.

7. Acknowledgement

The described article was carried out as part of the 2020-1.1.2- PIACI-KFI-2020-00165 "ERPA - Development of Robotic Process Automation solution for heavily overloaded customer services" project implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2020–1.1.2-PIACI KFI funding scheme.

References

[1] Aalst, W., Bichler, M., and Heinzl, A.: Robotic process automation. *Business Information Systems Engineering*, **60**, (2018), 269-272, URL https://dx.doi.org/10.1007/s12599-018-0542-4.

[2] Leopold, H., van der Aa, A. H., and Hajo, A. R.: Identifying candidate tasks for robotic process automation in textual process descriptions. *Enterprise, business-process and information systems modeling*, **1**, (2018), 67-80, URL https://dx.doi.org/10.1007/978-3-319-91704-7_5.

[3] Leno, V., Polyvyanyy, A., Dumas, M., Rosa, M. L., and Maggi, F.: Robotic process mining: vision and challenges. *Business Information Systems Engineering*, **63**/**3**, (2021), 301-314, URL https://dx.doi.org/10.1007/s12599-020-00641-4.

[4] Bose, R., Jagadeesh, C., and van der Aalst., W. M.: Discovering signature patterns from event logs. *IEEE Symposium on Computational Intelligence and Data Mining*, URL https://dx.doi.org/10.1109/CIDM.2013.6597225.

[5] Truong-Chi, T. and Fournier-Viger, P.: A survey of high utility sequential pattern mining. *High-Utility Pattern Mining: Theory, Algorithms and Applications*, pp. 97-129, URL https://dx.doi.org/10.1007/978-3-030-04921-8_4.

[6] Augusto, A. and et al.: Automated discovery of process models from event logs: review and benchmark. *IEEE transactions on knowledge and data engineering*, **31**/**4**, (2018), 686-705, URL https://dx.doi.org/10.1109/TKDE.2018.2841877.

[7] Aalst, V. D. and et al., W. M.: Design and implementation of the yawl system. *International Conference on Advanced Information Systems Engineering*, URL https://dx.doi.org/10.1007/978-3-540-25975-6_12.

[8] Gunther, C. W. and Verbeek, H. M. W.: Xes-standard definition. *IEEE Computational Intelligence Magazine*, pp. 4-8, URL https://dx.doi.org/10.1109/MCI.2017.2670420.

[9] Ferreira, D. R. and Gillblad, D.: Discovering process models from unlabelled event logs. *International Conference on Business Process Management*, URL https://dx.doi.org/10.1007/978-3-642-03848-8_11.

[10] Avigdor, G., Senderovich, A., and Weidlich., M.: Challenge paper: data quality issues in queue mining. *Journal of Data and Information Quality (JDIQ)*, **9**/**4**, (2018), 1-5, URL https://dx.doi.org/10.1145/3165712.