fCM Fragment Construction From SUBDUE-Based Pattern Extraction

Jost Goette[1] and Sandro Speh[1]

Hasso Plattner Institute, University of Potsdam, Germany
{jost.goette,sandro.speh}@student.hpi.uni-potsdam.de

Abstract. Mining event logs with high activity variance produces complicated models that are hard to grasp in conventional process models. Extracting behavioural patterns from the log using SUBDUE-based pattern extraction enables us to retrieve closely connected sets of activities. Fragment-based case management (fCM) allows us to visualize these patterns and show their interconnectedness. The presented semi-automated process uses SUBDUE to extract frequent occurring patterns and retrieves data objects that can be combined to an fCM model.

Keywords: Behavioural Patterns · fCM · SUBDUE

1 Introduction

Process Mining of logs with high variance can lead to complicated mined models. Such logs tend to have numerous different cases and high diversity in behaviour. As a result, the generated models tend to be difficult to understand because of the large number of paths between the activities. Gaining an overview of process steps becomes a challenge which reduces the understanding and therefore improvability of the process. To simplify the understanding of processes that result in these so-called spaghetti models [14] using conventional mining algorithms, it could help to split the model up into repeating fragments consisting of a subset of activities and to display the interconnection between these fragments. This eases the readability and understandability of the process.

Case management is one possibility to do this as it is used for modelling variable processes performed by knowledge workers [9]. Using fragment-based case management (fCM), one could derive fragments from complex event logs and display the relation between the fragments as an fCM model to gain an overview of the individual process steps. In this paper, we present an approach that retrieves fragments of often occurring activity sets, so-called *behavioural patterns*, that are combined with extracted data objects to aggregate them to an fCM model. The approach relies on SUBDUE-based pattern extraction and was tested on an event log that was extracted from the MIMIC-IV[11] database.

The remainder of this paper is structured as follows. In Section 2 we describe preliminary notions to understand the paper. Section 3 presents related work to our approach. In Section 4 we present the approach with its individual process steps. We discuss the approach and our result using an example data set in Section 5, while Section 6 concludes the paper.

2 Background

In this Section, we introduce the important preliminaries process mining, fCM, instance graphs, behavioural patterns and the SUBDUE algorithm to help understand the further content of the paper.

2.1 Process Mining

One part of process mining is the process discovery, where process models can be extracted from event logs using process discovery algorithms. These process models depict the underlying process and can be used to analyse the process and in the end enhance it. Event logs store records of events, that are characterized by a timestamp, a related activity and optionally attributes. Furthermore, each event belongs to a process instance (trace), which is one possible process execution or case. In order to identify the event's trace, each event holds a case id.

2.2 Fragment-based Case Management

In Fragment-based Case Management, business processes are captured in case models, consisting of four parts, the domain model, object lifecycles, process fragments and a goal state [9]. The domain model defines the business objects relevant for the scenario as a set of data classes and their associations in a UML diagram. Each data class in the domain model comes with an associated object lifecycle that specifies valid behaviour of its instances, i.e. data objects. Instead of being combined in one full process model, BPMN ([7]) process model components are split into fragments, describing a structured part of a business scenario. They can be dynamically executed and combined in runtime, via data objects state changes, enabling and disabling fragment activity execution. At runtime, the scenario is represented by the instantiated case model in form of a case. It has a state that changes when knowledge workers perform activities. While being similar to process instances in traditional workflow systems, cases consist of several fragment instances, as well as data objects. A case can be terminated when a specific goal is reached, that is when certain data objects are in a desired state [9].

2.3 Instance Graph

An instance graph represents the flow of the activities for a single process instance. The nodes of this graph are the events, the edges between two nodes represent the ordering relation between corresponding events. If an instance graph has branches, the events of the branches are executed in parallel. By definition there cannot be exclusive behaviour (XOR joins/splits) in an instance graph, because only one of the possible paths can be chosen. Instance graphs are acyclic as each loop execution is represented by the repetition of its corresponding events in the instance graph [8].

2.4 Behavioural Pattern

In the context of process mining, behavioural patterns represent activities of the process that often occur together in a case, for example triggered by circumstances like attributes having certain values. In this work we build on the determination approach by Diamantini et. al. [8], where a behavioural pattern is a subgraph of an instance graph that is considered relevant. This means their compression index is higher than a threshold k. The compression index considers the size of the subgraph and its frequency over all instance graphs. Hence, the most relevant subgraph is the one that compresses the instance graphs the most [8].

2.5 SUBDUE

SUBDUE is a structural knowledge discovery algorithm that discovers substructures in graphs. The goal of the search is to find a substructure that leads to the highest compression[1] in the input graph [10]. The algorithm can be run iteratively, as the graph with the contraction of the found substructure can be used as input for the next iteration. The input graph does not necessarily need to be connected, therefore a set of instance graphs can be the input as well. This allows to find substructures that lead to a high compression over all instance graphs. To lead to a high compression, these substructures consequently have to either occur very frequently or must include many elements in the set of instance graphs.

3 Related Work

In this Section we present two approaches that already dealt with behavioural pattern extraction from event logs and compare them to our solution.

Acheli et al. introduce the COBPAM algorithm [4]. Here, potential behavioural patterns are obtained by combining simpler behavioural patterns using algebraic operations on process trees. The algorithm exploits a partial order on potential behavioural patterns to discover only those that are maximal while respecting a compactness property. Moreover, COBPAM considers that complex behavioural patterns can be characterized as combinations of simpler behavioural patterns, which enables pruning of the behavioural pattern search space. In later work, Acheli et al. present the Contextutal COBPAM algorithm (CCOBPAM) [5]. Contextual behavioural patterns may be frequent solely in a certain partition of the event log, which enables fine-granular insights into the aspects that influence the conduct of a process. By including analysis of causal relations between behavioural patterns and the context information, behavioural patterns are included in the result that are frequent only in their specific context and not in the whole log.

[1] Compressing the substructure means to contract the edges between the nodes in the substructure and therefore merging multiple nodes but maintaining the adjacency of the resulting node to other nodes in the graph.

In their contribution, Diamantini et al. [8] extract behavioural patterns from instance graphs using the SUBDUE algorithm (Section 2.5). To this end, they exploit the causal relationships found by the inductive miner [12] to build instance graphs, with events having edges to their closest causal successor and predecessor. Subsequently, they repair irregular, noiseful traces. They then apply hierarchical clustering to mine behavioural patterns. For this, they use the instance graphs as input for SUBDUE (2.5) At each iteration, the algorithm compresses the subgraph, which is the most relevant behavioural pattern, found by SUDBUE in the input graphs and presents the result again to SUDBUE.

Our approach compares to the two solutions in the sense that it is similar to the Diamantini approach. We also apply SUBDUE on prior generated instance graphs to extract patterns. But as discussed in Section 2.3 we have a different strategy for instance graph generation. Additionally we leave out the repair of irregular traces. We do not rely on the notion of a context or the COBPAM algorithm and therefore consider patterns in the entire log. We append the Diamantini approach by the construction of fCM (Section 2.2) fragments based on the extracted patterns.

4 fCM Fragment Construction From SUBDUE-Based Pattern Extraction

The approach, depicted in Figure 1, consists of five stages. First, we preprocess our input data to retrieve a directed acyclic graph from it. Next, we use SUBDUE to iteratively retrieve patterns from our graph and compress each pattern similar to the approach presented in the work by Diamantini et al. [8]. As soon as SUBDUE cannot find patterns any more, we post process the found patterns and then continue with the creation of the data objects. Finally, we construct the fragments for the fCM model. In this chapter, we describe and explain the individual steps of the process and briefly describe the implementation in the end.

4.1 Preprocessing

The preprocessing retrieves the event log and constructs a directed acyclic graph to build the input for the SUBDUE algorithm. To retrieve the data, we load the event log and mine a Petri net using the inductive visual miner. This miner enables us to replay all traces from the log, since the mined model is complete and therefore each trace can be found in the model [13]. The Petri net provides relevant information regarding the process behaviour, especially which activities are parallel or sequential to each other. This will be useful when we build the instance graphs later on. From the Petri net, we construct an adjacency graph, where two transitions are adjacent if there is a path in the Petri net between the transitions using only places or silent transitions[2]. Next, because we consider

[2] A silent transition is a transition that has no relation to the trace activities

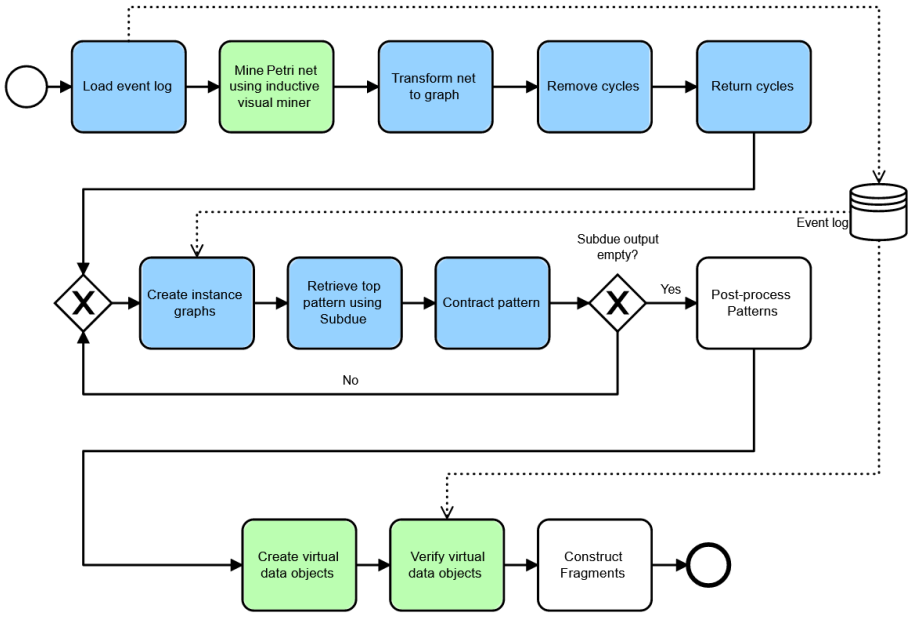

Fig. 1: Process of approach and its fully automated (blue), semi-automated (green) and manual (white) tasks.

activity loops as a single fragment since they can be executed iteratively in cycles, we remove backward edges and therefore cycles from our graph. Doing this, we also remove multi activity cycles because instance graphs do not contain cycles. Instead, they contain repeating activities multiple times. We store these cycles because they will be incorporated into the final model when post-processing the patterns. By doing this, we include the cyclic information in the log in our final model. After the removal of the cycles, we receive a directed acyclic graph representing the process model as output of the preprocessing phase. Another output is the collection of cycle edges, as we do store them to build fragments later in the process.

4.2 Pattern Mining

After we retrieved the input data and preprocessed it, we continue with the pattern mining. Here we iteratively run SUBDUE and then contract the found pattern in the graph and our instance graphs. As soon as SUBDUE returns no more patterns, we post process the patterns to continue with the next phase, the retrieval of the data objects.

To execute SUBDUE, we first construct instance graphs (cf. Section 2.3) from our directed acyclic graph. To construct the set of instance graphs, we iterate over all traces. For each trace, we collect the activities and remove the

duplicates from it. Next, we retrieve the subgraph of our activities from our adjacency graph. This subgraph contains only the activities in the trace as nodes and the edges between them. As we have mined our graph from the Petri net, each instance graph contains the information regarding sequential or parallel execution of its activities. Because instance graphs show the path an instance has taken, the instance graph also correctly represents exclusive behaviour of the original process, as we only display the branch of the exclusive behaviour taken. We can also remove the duplicates from the trace, as we do not support loops in instance graphs and we do not need duplicates to retrieve subgraphs.

The collection of instance graphs are the input for the SUBDUE execution. Subdue (cf. Section 2.5) retrieves patterns in the graph and ranks them based on their contraction grade. We select the pattern with the highest contraction grade and retrieve its edges and nodes. Using the pattern graph, we contract the pattern to a single node in our adjacency graph. The pattern node is adjacent to all nodes that each node of the pattern has been adjacent to. We also remove the edges between the pattern nodes because we want to avoid self loops[3] in the graph. Our contracted adjacency graph is the starting point for the next iteration. Again, we construct the instance graphs as the first step. The only difference is, that whenever we traverse an activity of an already contracted node in the trace, we replace this activity with the pattern that it is inside of. Apart from this, there are no other changes to the instance graph creation. Then we can again execute SUBDUE, retrieve the highest ranked pattern and then contract it.

The process iterates until SUBDUE does not find any patterns anymore. This does not necessarily mean that our contracted graph is only a single node. SUBDUE only detects a pattern if it occurs in more than one instance graph. Therefore, we might end up with single nodes in our adjacency graph that do not belong to a pattern. We include these nodes when we post process our patterns.

4.3 Post Processing

After the iterative SUBDUE pattern extraction terminates, we enter a post-processing phase. Here, we prepare the found patterns and remaining activities for the data object retrieval and fragment construction. To this end, we perform several modification steps. In order to prepare the execution dependencies between patterns, we memorize for each pattern, which of its activities have edges to which outside activities, that do not belong to the pattern, and call these connections *pattern edges* (PE). We split patterns that contain concurrency, since part of our modelling strategy is to model concurrency and exclusivity through fragments and data objects. We therefore refrain from gateways inside the fragments. That way, we can more efficiently handle the case when later found patterns contain paths of concurrency/exclusivity that would extend prior found structures of that behaviour. When doing so, we model predecessor

[3] A self loop is a loop that starts and ends in a single node and has only one edge

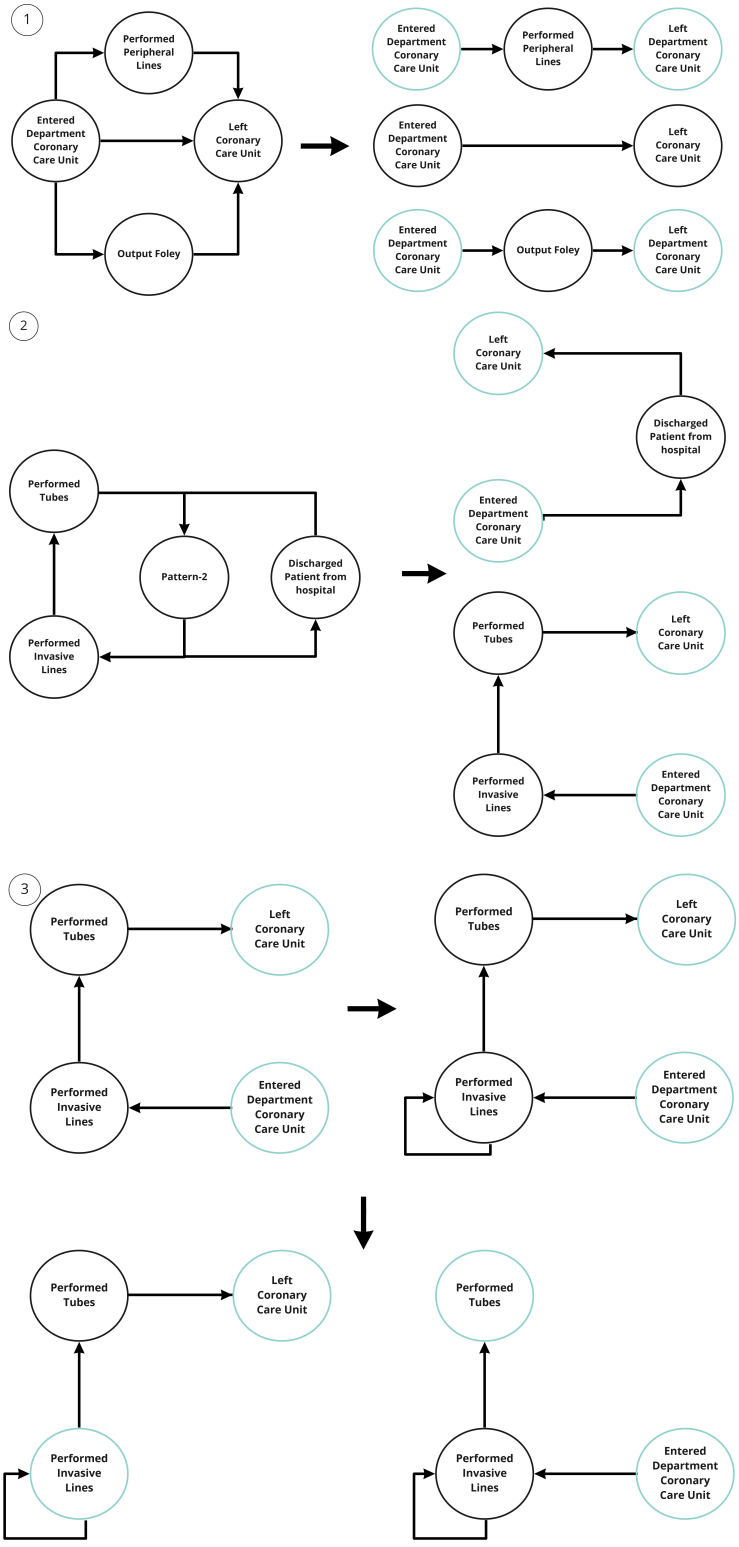

Fig. 2: Post Processing Steps (Before/After) with turquoise modelling execution dependency references : (1) Handling of Parallelism, (2) transitive pattern edges (3) loop insertion

and successor as one *post-processed pattern* (PPP) and each path of the concurrency/exclusivity as a separate PPP. Additionally, we memorize the execution dependency of the predecessor and successor for the PPP, for the same reason we construct the PEs. In the case that we split a pattern that includes another pattern node, we can now exploit the patterns PEs to replace the pattern node with the corresponding PE as an execution dependency. This can work transitively through multiple pattern nodes, for example if the investigated connection belongs to an activity inside a pattern which is embedded in another pattern, that is connected to the second concernd activity. Next, we reinsert the loops memorized in the preprocessing and, if necessary, extract them into separate PPPs in the same fashion as in the other steps, by keeping the neighbours in their mined pattern as references. The second to last step is to handle the remaining activities, that were not included in any pattern. By looking at the DAG after the last iteration, we can infer for each remaining activity its connections by inspecting its PEs. As we cannot treat the final DAG as another pattern, each of the remaining activities gets extracted into a separate PPP. We cannot assume, that any of the remaining activities occur together in a case.

Finally, if and only if, all traces start with the same activity, we mark the PPP with that activity as the initial fragment. Other cases are currently not supported by our approach.

4.4 Data Objects Retrieval

As a first step, from the PPPs execution dependency references, we can derive *virtual data objects* (VO). They defer from "real" *data objects* (DO) in the sense that they do not necessarily depict data behaviour. If the state of a VO changes, this does not stem from an attribute change in the underlying data model. Instead, we use VOs to model fragment dependencies, like concurrent behaviour and to disable and enable activities or fragments. Still, VOs can also be DOs, which is why the data object retrieval includes a confirmation check of the VOs.

Virtual Data Object Derivation We start the derivation with the base pattern. Using the mined Petri net as a source to look up behaviour, we create VOs from the execution dependency references. For the VO construction, we propose a set of rules, that depend on the behaviour the VOs have to depict. However, they are limited to a range of supported cases, as especially multiple combinations of behaviour can lead to complex implications for the VOs. Nevertheless, some requirements can be articulated:

1. In order to depict concurrency, the last, non-optional activity in the Petri net before the parallel split has to enable the activities on the path. The successor has to disable them.
2. Optional activities that are executed zero to one times should be disabled by their direct successors.
3. Exclusivity works similar to concurrency with the addition that each path disables the other paths via a state change to a path dedicated state.

4. Input and output sets can be used to depict different cases that can occur if optional events, exclusivity and loops are combined.
5. convert execution dependencies between patterns, prior found in PEs, to VOs.

Virtual Data Object Confirmation After the VO derivation, we check if any of the constructed objects are DOs. For this, we examine whether consistent attribute changes in the log exist for those activities that are connected via virtual data objects. Hence, we group the log case wise and compare the attribute values for each activity pair which are related through a VO. If any changes in each trace can be found for, we convert that VO into a DO that represents the so found attribute(s) and their changes as state changes. Otherwise, we simply keep the VOs as they were before.

4.5 Fragment Construction

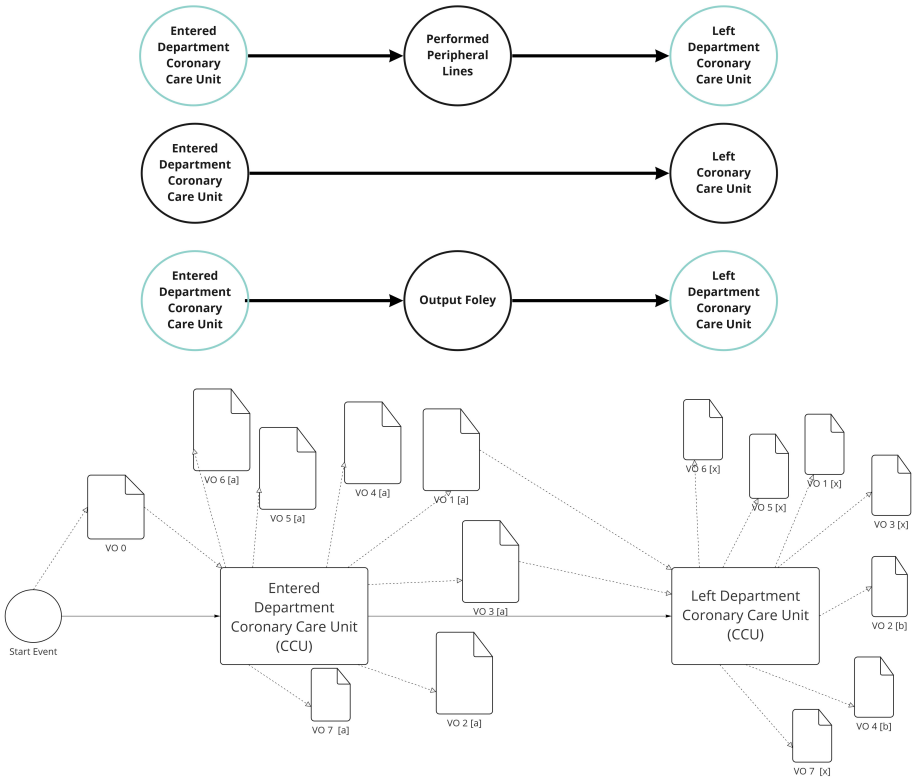

Fig. 3: Example of post processed pattern and its corresponding fragment

Finally, the fragments can be constructed combining the PPPs from Section 4.3 and the VOs and DOs from Section 4.4. We use the nodes from the PPPs as activities and switch their execution dependency references with the corresponding VOs and DOs, in accordance of the rules defined in Section 4.4. While, the predestined intitial fragment's starting activity from Section 4.3 is connected with a starting event.

4.6 Implementation

We automated most steps of the process. The results are available in repository on GitHub[4]. The verification of the virtual data objects is semi-automated using a jupyter notebook as well as the Petri net mining which has to be done using ProM 6 [3] and the inductive visual miner by Leemans [12]. Post-processing the patterns and constructing the fCM fragments remain a manual task, except for the confirmation of the virtual objects. The automation is a python script which uses SUBDUE as submodule. It supports the preprocessing, iterative pattern extraction and virtual data object extraction. The Petri net processing and the log interaction relies on the PM4PY [6] module. A user only has to state the locations of the Petri net and the log in the *Extensible Event Stream* format (XES) [1] and will receive the found patterns, the unverified virtual data objects and cycle edges. From then on, the user can post process the patterns, verify the data objects and finally construct the fragments.

5 Evaluation

In this Section we analyse the result of our approach achieved with an exemplary dataset and reflect underlying assumptions as well as its limitations. But first, we introduce the dataset we use.

MIMIC-IV [11] is a publicly available database of patients admitted to the Beth Israel Deaconess Medical Center in Boston, USA. It contains data of patients admitted to an intensive care unit or the emergency department between 2008 and 2019. For each patient, it contains comprehensive information while they were in the hospital, for example laboratory measurements, medications administered or vital signs. The data is de-identified and split up in multiple modules.

We focus on a subset that stems from a combination of the *core* and *icu* module. including patient tracking data and data sourced from MetaVision, the clinical information system of the hospital.

Our exemplary data stems from the MIMIC-IV dataset and consists of 11 coronary care unit traces. Each trace represents a patient during their stay at the care unit. Each activity of trace describes various activities during the stay, ranging from entering or leaving the unit over machinery output from catheters to simple procedures being done on the patient. Overall, it contains 20 distinct activities and 328 trace events.

[4] `https://github.com/SanJSp/FCMFromSudbueBasedPatternExtractor`

We evaluate our approach by qualitatively analysing our results. To that end, we compare our results to the guidelines presented in [2] First of all, our approach does not produce a complete fCM model, as object life cycles, domain model as well as goal state are missing and data objects may be incomplete. We therefore only check the alignment with the fragment guidelines. While we generally concord to them, the produced fragments tend to consist of few activities and rely heavily on data objects. Note, that this is not entirely uncharacteristic for fCM. Therefore, the approach can be enhanced to produce an even more readable process representation. Opposed to the guidelines, we do not model automatic decisions with XOR-gates, as we model all decisions as different fragments enabled through data dependencies. We also do not consider all data dependencies as we only support VOs and some DOs with our VO confirmation. As pointed out in Section 4.6, the approach is only semi-automatic. Nevertheless, we are able to reliably mine patterns and create valid fragments even from larger logs with 100 traces that contain 100 activities each, while supporting different process behaviour.

5.1 Discussion

In this subsection we reflect our assumptions and limitations and discuss their consequences.

Although, fCM allows XOR and AND gateways in fragments, we model paths in concurrency as well as exclusions as separate fragments and use data objects to model the corresponding control flow. The question arises how complete and absolute these rules should be as modelling always trusts on the ability of the person to construct a good model beyond the rules in place to create a valid model.

Assumptions In order to enable the confirmation of the virtual data objects, we assume that knowledge about the data attributes of the domain our approach is used in is provided. For the same cause: when an attribute value changes for a record in the log in comparison to its case's predecessor/successor, we assume that this is due to the corresponding activity execution opposed to access from the outside. Furthermore, we assume that if all traces of the log start with the same activity, said activity can be considered the starting activity of the underlying process.

Limitations Our approach does not support loops where the path between exit point and entry point contains activities. This means their exit point lies in the middle of the loop, where the loop can be exited before each activity has been executed in an iteration. This is due to the loop extraction temporarily deleting the backwards edge of a loop. Another limitation of our approach is the construction of the virtual data objects that is currently not formally defined, rather consists of a vague set of rules and trusts the modellers modelling intuition. An aspect of this is that these rules can also change with the modelling strategy

that is being used. We only support logs where all traces start with the same activity.

Finally, it is again noteworthy, that the approach is limited by the circumstance that it is not able to produce a full fCM model.

6 Conclusion

In this paper, we proposed a pipeline to generate fCM process fragments from event logs of unstructured processes with a high variability of behaviour. To this end, we adapted and implemented the SUBDUE-based pattern extraction introduced by Diamantini et. al. [8] in Python and appended it by a post-processing, virtual data object retrieval and fragment construction. Future research could concentrate on integration of the remaining components of fCM. General techniques to construct for instance a domain model based on a event log could incorporated as this is not coherent with our current approach. Furthermore, the requirements for the virtual data object derivation are currently rudimental and the derived rules are not fully formalized. Therefore, we identify the need of completion and formalization for this part of the approach. Support for loops with activities on the path from exit point to entry point needs to be added. The implementation would benefit from the replacement of PROM [3] by PM4PY [6] log mining. Complete approaches for choosing start events need to be inspected and integrated. Future work could include the integration of differentiation between automated and human made decisions. Finally, comprehensive analysis and evaluation regarding the clarity of the produced results needs to be taken care of. This could be done in form of user interviews or questionnaires.

References

1. IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams. IEEE Std 1849-2016 pp. 1–50 (2016). https://doi.org/10.1109/IEEESTD.2016.7740858
2. fCM design support guidelines (2022), `https://github.com/bptlab/fCM-design-support/wiki/Guidelines`
3. Aalst, W., Dongen, B., Günther, C., Rozinat, A., Verbeek, E., Weijters, A.: Prom: The process mining toolkit. (01 2009)
4. Acheli, M., Grigori, D., Weidlich, M.: Efficient discovery of compact maximal behavioral patterns from event logs. In: Giorgini, P., Weber, B. (eds.) Advanced Information Systems Engineering. pp. 579–594. Springer International Publishing, Cham (2019)
5. Acheli, M., Grigori, D., Weidlich, M.: Discovering and analyzing contextual behavioral patterns from event logs. IEEE Transactions on Knowledge and Data Engineering pp. 1–1 (2021). https://doi.org/10.1109/TKDE.2021.3077653
6. Berti, A., van Zelst, S.J., van der Aalst, W.M.P.: Process mining for python (pm4py): Bridging the gap between process- and data science. CoRR **abs/1905.06169** (2019), `http://arxiv.org/abs/1905.06169`

7. Chinosi, M., Trombetta, A.: BPMN: An introduction to the standard. Computer Standards Interfaces **34**(1), 124–134 (2012). https://doi.org/https://doi.org/10.1016/j.csi.2011.06.002, https://www.sciencedirect.com/science/article/pii/S0920548911000766
8. Diamantini, C., Genga, L., Potena, D.: Behavioral process mining for unstructured processes. Journal of Intelligent Information Systems **47** (08 2016). https://doi.org/10.1007/s10844-016-0394-7
9. Hewelt, M., Weske, M.: A hybrid approach for flexible case modeling and execution. vol. 260, pp. 38–54 (09 2016). https://doi.org/10.1007/978-3-319-45468-9$_3$
10. Istvan Jonyer: Graph-based hierarchical conceptual clustering. The Journal of Machine Learning
11. Johnson, A., Bulgarelli, L., Pollard, T., Horng, S., Celi, L.A., Mark, R.: Mimic-iv (2021). https://doi.org/10.13026/S6N6-XD98, https://physionet.org/content/mimiciv/1.0/
12. Leemans, S., Fahland, D., Aalst, W.: Discovering block-structured process models from event logs containing infrequent behaviour. vol. 171, pp. 66–78 (05 2014). https://doi.org/10.1007/978-3-319-06257-0$_6$
13. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Exploring processes and deviations. In: Fournier, F., Mendling, J. (eds.) Business Process Management Workshops, Lecture Notes in Business Information Processing, vol. 202, pp. 304–316. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-15895-2_26
14. Veiga, G.M., Ferreira, D.R.: Understanding spaghetti models with sequence clustering for prom. In: van der Aalst, W., Mylopoulos, J., Sadeh, N.M., Shaw, M.J., Szyperski, C., Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) Business Process Management Workshops, Lecture Notes in Business Information Processing, vol. 43, pp. 92–103. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12186-9_10