Abstractions in Process Mining: A Taxonomy of Patterns

R.P. Jagadeesh Chandra Bose[1,2] and Wil M.P. van der Aalst[1]

[1] Department of Mathematics and Computer Science, University of Technology, Eindhoven, The Netherlands
[2] Philips Healthcare, Veenpluis 5-6, Best, The Netherlands

Abstract. Process mining refers to the extraction of process models from event logs. Real-life processes tend to be less structured and more flexible. Traditional process mining algorithms have problems dealing with such unstructured processes and generate spaghetti-like process models that are hard to comprehend. One reason for such a result can be attributed to constructing process models from raw traces without due pre-processing. In an event log, there can be instances where the system is subjected to similar execution patterns/behavior. *Discovery of common patterns* of invocation of activities in traces (beyond the immediate succession relation) can help in improving the discovery of process models and can assist in defining the conceptual relationship between the tasks/activities.
In this paper, we characterize and explore the manifestation of commonly used process model constructs in the event log and adopt pattern definitions that capture these manifestations, and propose a means to form abstractions over these patterns. We also propose an iterative method of transformation of traces which can be applied as a *pre-processing step for most of today's process mining techniques*. The proposed approaches are shown to identify promising patterns and conceptually-valid abstractions on a real-life log. The patterns discussed in this paper have multiple applications such as trace clustering, fault diagnosis/anomaly detection besides being an enabler for hierarchical process discovery.

1 Introduction

Process mining refers to the extraction of process models from event logs [1]. An event log corresponds to a bag of process instances of a business process. A process instance is manifested as a trace (a trace is defined as an ordered list of activities invoked by a process instance from the beginning of its execution to the end). Process mining techniques can deliver valuable, factual insights into how processes are being executed in real life.

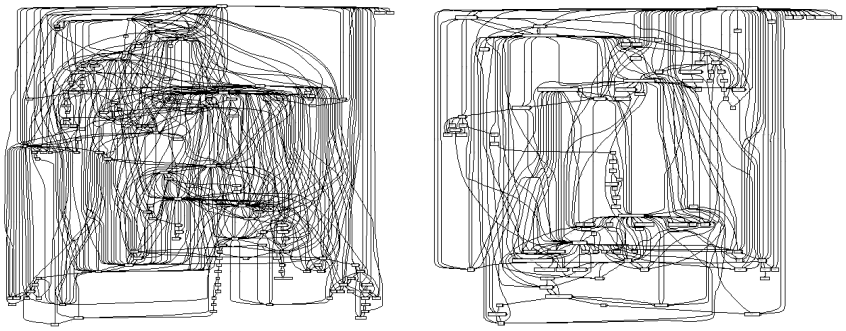
Real-life processes tend to be less structured than expected. Traditional process mining algorithms have problems dealing with such unstructured processes and generate spaghetti-like process models that are hard to comprehend. One reason for such a result can be attributed to constructing process models from raw traces without due pre-processing. A majority of process mining techniques

in the literature are purely syntactic in nature. From the viewpoint of existing process mining techniques all of the activities are different and unrelated. The activity names are treated simply as strings that typically do not have any semantics attached to them. However, in reality subsets of activities are related in that their usage is confined to certain contexts, and cater to some functionality. Recent efforts in process mining on Semantic MXML try to address this problem by incorporating semantics in the log specification [2]. It requires the domain expert to come up with the ontologies describing the domain concepts and relationships between them. Ontologies can assist in defining hierarchies of concepts over activities and there by provide abstractions. Asking a domain expert to build this from scratch would at times be too much to ask for, especially in real-life domains such as healthcare and finance where the complexity of the system/domain is too high.

The discovery of process models is based on the dependency relations that can be inferred among the activities in the log. More specifically, the dependency that is often explored is the succession relation (activity `B` succeeds activity `A`); process models are generated by assigning the control-flow links between tasks, i.e., activities based on the succession relation. Considering activities in isolation contributes to the "spaghettiness" of the discovered process models to a certain extent. Moreover, the context in which the activity is executed is not considered fully. In an event log, there can be instances where the system is subjected to similar execution patterns/behavior (where the pattern can manifest as a larger subsequence of tasks/activities), and instances where unrelated cases are executed. Discovery of common patterns of invocation of activities in traces (beyond the immediate succession relation) can help in improving the discovery of process models and can assist in defining the conceptual relationship between the tasks/activities.

In this paper, we first characterize and explore the manifestation of commonly used constructs (of building a process model) in the event log and propose pattern definitions that capture these manifestations. Some of these pattern definitions have been in existence in the string-processing and bioinformatics literature. We adopt these pattern definitions to the process mining domain and propose a means to form abstractions over these patterns. We propose a novel iterative method of transformation of traces which can be applied as a preprocessing step for most of the process mining analysis. The proposed approach first identifies the looping constructs in traces and replaces the repeated occurrences of the manifestation of the loop by an abstracted entity (activity) that encodes the notion of a loop. The second step involves the identification of sub-processes or common functionality in the traces and replacing the subprocesses/common functionality with abstract entities. We also present means to deal with complex process model constructs involving combination of choice, parallelism and loops. Fig 1 depicts two process models: one is obtained by mining the original log (Fig 1(a)), and the other is based on the log with abstractions (Fig 1(b)). We have used the heuristics miner plugin (with default settings) in

ProM[3] tool to mine the models. The model mined on the orignal log had 141 activities and 2901 arcs and had the fitness measures of 0.295 and -0.693 for the continuous semantics (cs) and improved continuous semantics (ics) metrics respectively. On the other hand, the model mined on the abstracted log had 99 activities and 537 arcs with fitness measures of 0.344 and 0.443 for the cs and ics metrics respectively. It is evident that the abstracted log is less spaghetti-like, more expressive, and more comprehensible.



(a) Process model mined from original log

(b) Process Model mined from log with abstractions over loop construct patterns

Fig. 1: Process models obtained by applying the heuristic miner to an event log of Philips Healthcare

We evaluate the goodness of the patterns proposed in this paper on a real-life log of Philips Healthcare. Philips Healthcare collates logs from their medical systems across the globe. These logs contain information about user actions, system events etc. The number of such log-recording systems in conjunction with the fine grained nature of logging makes the dataset available extremely large i.e., in the order of a few thousand logs per day. The patterns and abstractions presented in this paper are shown to be quite effective in that they are able to group activities pertaining to common functionality and also identify patterns of abnormal usage.

The rest of the paper is organized as follows. In Section 2, we introduce the notations used in the paper. Section 3 defines a few pattern definitions and correlates these signatures with the process model constructs. In Section 4, we propose one approach to form abstractions based on the patterns. Pattern definitions catering to the manifestation of complex process model constructs such as choice/parallelism within loops and sub-processes are discussed in Section 5. Approaches to discover these patterns from the event log are presented in Section 6. In Section 7, we propose an iterative approach of transforming traces which can be used as a pre-processing step for process mining analysis. In Section 8, we present and discuss the patterns uncovered in a case study of a real-life log

[3] ProM is an extensible framework that provides a comprehensive set of tools/plugins for the discovery and analysis of process models from event logs. See `http://www.processmining.org` for more information and to download ProM.

of Philips Healthcare. We discuss related work in Section 9. Finally, we conclude in Section 10.

2 Notations

Let $\mathcal{A}$ denote the set of activities. $|\mathcal{A}|$ is the number of activities. $\mathcal{A}^+$ is the set of all non-empty finite sequences of activities from $\mathcal{A}$. A trace, T is an element of $\mathcal{A}^+$. For $i \leq j$, $T(i,j)$ denotes the subsequence from the i^{th} position to the j^{th} position in the trace T. An event log, $\mathcal{L}$, corresponds to a multi-set (or bag) of traces from $\mathcal{A}^+$.
As an example, let $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ be the set of activities; $|\mathcal{A}| = 3$. $T =$`abcabb` is a trace of length 6. $T(2,5) =$`bcab` is a subsequence of T from positions 2 to 5. $\mathcal{L} = \{$`aba`, `aba`, `abba`, `baca`, `acc`, `cac`$\}$ represents an event log.

3 Taxonomy of Patterns

In this section, we will introduce various definitions of patterns and correlate them to the manifestation of process model constructs. Discovering such model constructs would help in answering questions such as *Are there loops within loops in my process model?*, *What are the most commonly used functionalities in my model?*, and also can assist in mining models bottom-up from primitive model constructs.

3.1 Loops as Tandem Arrays

Simple loops manifest as the repeated occurrence of an activity or subsequence of activities in the traces. In other words, an activity or a sequence of activities constituting a loop manifest themselves in a tandem fashion in a trace.

- *Tandem Array:* A tandem array in a trace T is a sub-sequence $T(i,j)$ of the form α^k with $k \geq 2$ where α is a sequence that is repeated k times. The *subsequence* α is called a tandem repeat type. We denote a tandem array by the triple (i, α, k) where the first element of the triple corresponds to the *starting position* of the tandem array, the second element corresponds to the *tandem repeat type*, and the third element corresponds to the *number of repetitions*.
- *Maximal Tandem Array:* A tandem array $T(i,j)$ of the form $\alpha^k (k \geq 2)$, is called a maximal tandem array if there are no additional copies of α before or after $T(i,j)$.
- *Primitive Tandem Repeat Type:* A tandem repeat type α is called a primitive tandem repeat type if and only if α is not a tandem array. i.e., $\alpha = \beta^p$, for some non-empty sequence β only if $p = 1$.
- *Primitive Tandem Array:* A tandem array $T(i,j)$ of the form α^k $(k \geq 2)$, is a primitive tandem array iff α is a primitive tandem repeat type.

For example, consider the trace T=`gdabcabcabcabcafica`. $(3, \mathtt{abc}, 4)$, $(3, \mathtt{abcabc}, 2)$, $(4, \mathtt{bca}, 4)$, $(4, \mathtt{bcabca}, 2)$, $(5, \mathtt{cab}, 3)$ are the tandem arrays in T. The corresponding tandem repeat types are `abc`, `abcabc`, `bca`, `bcabca`, `cab` respectively. The primitive tandem repeat types are `abc`, `bca`, `cab`.

3.2 Sub-processes as Conserved Regions

Finding similar regions (sequence of activities) common within a trace and/or across a set of traces in an event log signifies some set of common functionality accessed by the process. In other words, a region of high similarity shared within a process instance or by two or more process instances might be evidence of common functionality (often abstracted as a sub-process). In order to find these commonalities across the traces in the entire event log, we first construct a single sequence which is obtained by the concatenation of traces in the event log with a distinct delimiter between the traces. Let us denote this concatenated sequence by S. Multiple invocations of a sub-process within a trace can be detected by finding similar regions manifested within a trace.

- *Maximal Pair:* A maximal pair in a sequence, S is a pair of identical subsequences α and β such that the symbol to the immediate left (right) of α is different from the symbol to the immediate left (right) of β. In other words, extending α and β on either side would destroy the equality of the two strings. A maximal pair is denoted by the triple (i, j, α) where i and j correspond to the starting positions of α and β in S with $i \neq j$.
- *Maximal Repeat:* A maximal repeat in a sequence, S is defined as a subsequence α that occurs in a maximal pair in S.
- *Super Maximal Repeat:* A super maximal repeat in a sequence is defined as a maximal repeat that never occurs as a substring of any other maximal repeat.
- *Near Super Maximal Repeat:* A maximal repeat α is said to be a near super maximal repeat if and only if there exist at least one instance of α at some location in the sequence where it is not contained in another maximal repeat.

Table 1: Maximal, Super Maximal and Near Super Maximal Repeats in each trace of the event log $\mathcal{L}$

Id	Trace	Maximal Repeat Set	Super Maximal Repeat Set	Near Super Maximal Repeat Set
T_1	aabcdbbcda	{a, b, bcd}	{a, bcd}	{a, b, bcd}
T_2	dabcdabcbb	{b, dabc}	{dabc}	{b, dabc}
T_3	bbbcdbbbccaa	{a, b, c, bb, bbbc}	{a, bbbc}	{a, c, bbbc}
T_4	aaadabbccc	{a, b, c, aa, cc}	{b, aa, cc}	{a, b, aa, cc}
T_5	aaacdcdcbedbcc-badbdebdc	{a, b, c,d, e, aa, bd, cb, db, dc, cdc}	{e, aa, bd, cb, db, cdc}	{a, c, e, aa, bd, cb, db, dc, cdc}

Consider the event log, $\mathcal{L} = \{$aabcdbbcda, dabcdabcbb, bbbcdbbbccaa, aaadabbccc, aaacdcdcbedbccbadbdebdc$\}$ over the alphabet $\mathcal{A} = \{$a, b, c, d, e$\}$. Table 1 depicts the maximal, super maximal and near super maximal repeats present in each trace of the event log. For trace T_1, the set of maximal repeats = {a, b, bcd}. Since maximal repeat b, is subsumed in maximal repeat bcd, b does not qualify to be a super maximal repeat. The occurrence of maximal repeat b at position 6 in T_1 does not overlap with any other maximal repeat. Hence b qualifies to be a near super maximal repeat. Similarly for trace T_3, all occurrences of maximal repeats b and bb coincide with the maximal repeat bbbc. Hence neither

qualify for near super maximal repeat. The occurrence of maximal repeat `c` at position 10 in T_3 does not coincide with any other maximal repeat. Hence, `c` qualifies to be a near super maximal repeat.
Table 2 depicts the maximal/super maximal/near super maximal repeats present in the entire event log, $\mathcal{L}$. These are the repeats in the sequence obtained by concatenation of all traces in the event log. Near super maximal repeats are a hybrid

Table 2: Maximal, Super Maximal and Near Super Maximal Repeats in the Event Log $\mathcal{L}$

Maximal Repeat Set	{a, b, c, d, e, aa, ab, ad, bb, bc, bd, cb, cc, cd, da, db, dc, aaa, abc, bbc, bcc, bcd, cdc, dab, abcd, bbbc, bbcc, bbcd, bcda, dabc, bcdbb}
Super Maximal Repeat Set	{e, ad, bd, cb, aaa, cdc, abcd, bbbc, bbcc, bbcd, bcda, dabc, bcdbb}
Near Super Maximal Repeat Set	{e, aa, ad, bb, bd, cb, cc, db, dc, aaa, bcc, cdc, dab, abcd, bbbc, bbcc, bbcd, bcda, dabc, bcdbb}

between maximal repeats and super maximal repeats in that it contains all super maximal repeats and those maximal repeats that can occur in isolation in the sequence without being part of any other maximal repeat. Near super maximal repeats can assist in identifying *choice constructs* in the process model. Let us denote the set of maximal repeats, super maximal repeats and near super maximal repeats by M, SM and NSM respectively. The following relation holds between the three.

$$SM \subseteq NSM \subseteq M$$

The set $NSM \setminus SM$ (the set difference) depicts all maximal repeats that occur both in isolation and are also subsumed in some other maximal repeat. For any repeat $r \in NSM \setminus SM$, a super maximal repeat r^s which contains (subsumes) r can be either of the form αr or $r\beta$ or $\alpha r\beta$ (where α and β are subsequences of activities). This indicates that r can be a common functionality which might occur in conjunction with α and/or β. In other words, it indicates that α and β can potentially be optional (sequence of) activities in the context of r.

3.3 Mapping Primitive Tandem Repeats and Conserved Regions into Equivalence Classes

We consider both a *primitive tandem repeat type* and all variants of *maximal repeat* as a *repeat* in this section. For a repeat, r, let repeat alphabet $\Gamma(r)$, denote the set of symbols/activities that appear in the repeat. For example, for the repeats `abba`, `abdgh`, and `adgbh`, the repeat alphabets correspond to $\{\mathtt{a}, \mathtt{b}\}$, $\{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{g}, \mathtt{h}\}$, and $\{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{g}, \mathtt{h}\}$ respectively. Different repeats can share a common repeat alphabet. In the above example, the repeats `abdgh` and `adgbh` share the same repeat alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{g}, \mathtt{h}\}$. We can define equivalence classes on repeat alphabet.

$$[X] = \{r \mid r \text{ is a repeat and } \Gamma(r) = X\}$$

For the above example, $[\{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{g}, \mathtt{h}\}] = \{\mathtt{abdgh}, \mathtt{adgbh}\}$. Furthermore, the equivalence class under repeat alphabet will capture any variations in the manifestation of a process execution due to *parallelism*.

Reducing the number of features Large data sets and data sets with large alphabet might contain abundant repeats. But not all of them might be characteristically significant. For example, there might be repeats which occur only in a small fraction of traces. One way to tackle this is to filter the repeats. One can retain only those repeats that are contained in a large fraction of traces in the event log, i.e., repeats that have a high support in the event log. Other means of feature reduction can also be thought of.

4 Abstractions of Patterns

Subprocess abstractions can be discovered by considering a *partial ordering* on the repeat alphabet. Subsumption is used as the *cover relation*. A repeat alphabet ra_1 is set to cover another repeat alphabet ra_2 if $ra_2 \subset ra_1$. For example, consider the repeat types `abcd` and `abd`. It is most likely for activity `c` to represent a functionality similar to that of `a`, `b`, and `d`, since `c` occurs within the context of `a`, `b` and `d`. By defining a partial order on the repeat alphabets and generating a Hasse diagram on the partial ordering, one can form abstractions by considering the *maximal* elements in the poset. Fig 2 depicts the partial ordering on the repeat alphabets as a Hasse diagram. {`a,b,c`} and {`a,c,d`} are the maximal elements of the partial ordering. Maximal elements can be considered as abstractions of processes. Let us denote these two maximal elements with abstract activities `A` and `B` respectively. Repeat alphabets under a maximal element can all be represented with the abstraction of the maximal element. Repeat alphabets that contribute to more than one maximal element can either be put in one of the maximal elements or can define an abstraction in itself. It can be considered as a (sub-)functionality that is used in a larger functionality. In our example, let us assume that the repeat alphabet {`a,c`} is assigned to the maximal element {`a,b,c`}. With this abstraction, all repeats with repeat alphabets {`a,b`}, {`a,c`}, {`b,c`}, and {`a,b,c`} are represented by the abstract activity `A` while the repeats with repeat alphabets {`a,d`} and {`a,c,d`} are represented by the abstract activity `B` in all the traces. There can be instances where two

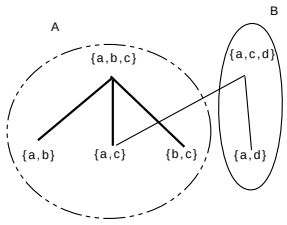


Fig. 2: Hasse diagram of the repeat alphabets

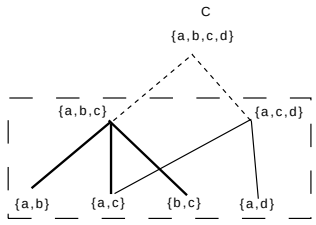


Fig. 3: Hasse diagram of the repeat alphabets with extended joins

maximals of the parital ordering on the repeat alphabets share a lot in common.

In order to reduce the total number of abstract activities introduced, one can define extended joins on the maximal elements. Fig 3 depicts the scenario where an extended join has been introduced on the maximal elements of Fig 2. The two maximal elements of Fig 2 viz., {a,b,c} and {a,c,d} are extended to join at {a,b,c,d}. The extended join now covers all repeat alphabets. Let us denote the extended join with abstract activity C. All repeats with repeat alphabets {a,b}, {a,c}, {b,c}, {a,d}, {a,b,c} and {a,c,d} can now be represented by a single abstraction viz., C in all the traces.

Let ra_1^m and ra_2^m be repeat alphabets corresponding to two *maximal* elements in the Hasse diagram. Different criteria for extending the maximal elements can be defined. For example, one can choose to extend two maximal elements provided they share a set of common elements above a particular threshold and also when the differences between them is less. In other words, extend the maximal elements only if $|ra_1^m \cap ra_2^m| \geq \delta_c$ and $|(ra_1^m \setminus ra_2^m) \cup (ra_2^m \setminus ra_1^m)| \leq \delta_d$. δ_c corresponds to the threshold on the number of common elements which can either be a fixed constant or a fraction of the cardinality of the participating maximal elements such as $0.6 \times \min(|ra_1^m|, |ra_2^m|)$. δ_d corresponds to the threshold on the number of differences between the two maximal elements.

5 Patterns in the Manifestation of Complex Process Model Constructs

The pattern definitions defined above (both tandem arrays, maximal repeats and its variants) capture some important manifestations of the process model constructs, but they are not sufficient enough to cater to complex model constructs where there is a parallelism or choice within other constructs. We call the above pattern definitions to be *exact*. In order to deal with complex constructs, the pattern definitions need to be more flexible and robust. In this section, we address some of these pattern definitions and call these *approximate*.

5.1 Approximate Tandem Arrays

In a trace T, an *approximate tandem array* is a concatenation of sequences $\alpha = s_1 s_2 s_3 \ldots s_k$ for which there exists a sequence s_c such that each s_i $(1 \leq i \leq k)$ is approximately similar to s_c. The notion of similarity can be defined in multiple ways (such as Hamming distance, string edit distance). For example, two sequences with string edit distance [3] less than δ (for some threshold, δ) can be considered to be similar. Here, s_c can be different from each and every s_i; alternatively, we may constrain that s_c be equal to at least one s_i $(1 \leq i \leq k)$. s_c is called as the *primitive approximate tandem repeat type*, and the approximate tandem array α is represented by the triple (j, s_c, k) where j signifies the starting position of α in T. Edit distance is defined as the minimum number of operations required to transform one sequence into the other (where the operations correspond to substitution, deletion or insertion of activities). Generic edit distance uses a cost function where different costs can be associated to the edit

operations. Levenshtein distance (LD) is a specific case of generic edit distance where all the symbols are treated equally and the cost of each edit operation is 1. Levenshtein distance might not be a good metric in most scenarios as it does not consider the context for edit operations. We have discussed some of the pitfalls of Levenshtein edit distance and proposed an automated approach to derive the costs of edit operations in [4]. One can use the generic edit distance framework [3] to define robust notions of similarity for approximate patterns.

– *Choice within Loops:* Approximate tandem arrays can be used to detect choices within loops. For example, consider the process model construct depicted in Figure 4(iii). In this example, we have a choice construct over the activities `b` and `c` inside the loop. $S =$ `abdacdacdabd` is one manifestation of the process model that constitutes an approximate tandem array with `abd` or `acd` as an approximate primitive tandem repeat type. The similarity criteria is Levenshtein distance, $LD(s_c, s_i) \leq 1$.

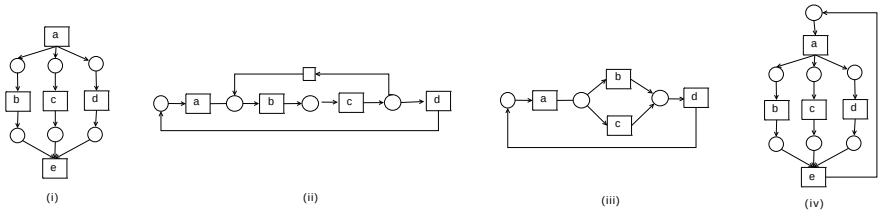


Fig. 4: Few Examples of Complex Process Model Constructs

– *Parallelism within Loops:* Parallelism within loops can also be handled in a similar fashion by approximate tandem arrays. However, defining an appropriate notion of similarity is crucial for the success of this approach. A too lenient notion might generate too many false positives while a stringent notion will miss certain constructs. This problem is compounded by the number of activities involved in the parallelism construct. A more practical way is to process the traces iteratively as would be discussed in Section 7.

5.2 Approximate Conserved Regions

Just like the approximate tandem arrays, we can define notions of approximation for the non-tandem repeats (maximal repeats, super-maximal and near-super maximal repeats). Approximate repetitions are specified by authorizing some number of errors between repeated copies. The set of allowed errors can be defined under the hamming distance and edit distance framework. If only replacements are allowed, this yields the classic Hamming distance, defined as the number of mismatches between the two sequences; if both replacements and insertions/deletions are permitted, then we are operating in the edit distance framework.

A repeat pair (α, α') is a $k-$approximate repeat if and only if the distance

between them $d(\alpha, \alpha') \leq k$. Consider Fig 4(i) which contains a parallelism construct. T_1 = `abcde`, T_2 = `acdbe`, T_3 = `adcbe` are some of the traces of the construct. The pairs (`abcde`, `acdbe`), (`acdbe`, `adcbe`) and (`abcde`, `adcbe`) are all 2−approximate under the Levenshtein edit distance.

6 Approaches for Discovering the Patterns

Maximal, super maximal and near super maximal repeats can be efficiently discovered in linear time using suffix trees for strings [5], [6]. Repeats that exist across the traces in the event log can be determined by applying the repeat identification algorithms on the sequence obtained by concatenating the traces in the event log with a delimiter not present in the alphabet $\mathcal{A}$. Such a concatenation of traces might yield a very long sequence. One can adopt efficient suffix-tree construction techniques such as [7] to handle very long sequences. Approximate repetitions can be found by first identifying exact repetitions and searching for all sub-sequences within a distance of k with the exact repetitions.

Gusfield and Stoye [8] proposed a linear time algorithm based on suffix trees to detect tandem repeats. Discovering tandem arrays takes $\mathcal{O}(n + z)$ time, where n is the length of the trace and z is the number of primitive tandem repeat types in the trace. Sokol et al [9] proposed an approach for a variant of approximate tandem arrays under the edit distance in $\mathcal{O}(nk \log k \log(n/k))$ time and $\mathcal{O}(n+k^2)$ space (where k is the threshold on the edit distance for similarity). The generic problem of approximate tandem arrays is still an open research problem. We have adopted Ukkonen's algorithm [10] for the construction of suffix-trees in linear-time.

7 Pre-Processing Traces and Resolving Complex Constructs

7.1 Pre-Processing Traces with Abstractions

The discovery of process models is based on the dependency relations that can be inferred among the activities in the log. More specifically, the dependency that is often explored is the succession relation (activity `B` succeeds activity `A`); process models are generated by assigning the control-flow links between tasks/activities based on the succession relation. Invocations of an activity in different contexts are treated alike. The fan-in/fan-out of the control-flow links on an activity increases by such a treatment thereby making the final model look spaghetti-like. Spaghettiness of process models can be reduced by first mining common functionalities/constructs, abstracting them and then discovering process models on the abstracted log. By doing so, *multiple invocations of an activity can be distinguished based on the context of its occurrence.* Algorithm 1 presents a single-phase of the pre-processing. The *basic idea is to first process for any loop constructs* (find patterns pertaining to loops viz., tandem arrays and approximate tandem arrays) *and replace the manifestation of loops with abstract entities.* Subsequence

patterns that are conserved within a trace and/or across the event log (signifying common functionality) are then discovered and abstracted. This can be iterated over any number of times with the event log for iteration $i+1$ being the output event log of iteration i.

1: Given an event log $\mathcal{L} = \{T_1, T_2, T_3, \ldots, T_m\}$.
2: Remove duplicate traces from $\mathcal{L}$. Let the set of unique traces be $\mathcal{L}' \subseteq \mathcal{L} = \{T'_1, T'_2, \ldots, T'_n\}$; Each $T'_i \in \mathcal{L}$.
3: Let $\mathcal{L}'' = \phi$
4: {Identify loop manifestations}
5: for all $T'_i \in \mathcal{L}'$ do
6: Identify all primitive tandem arrays, approximate tandem repeats in T'_i. Let $\mathcal{PTR}_i$ denote the set of all primitive tandem repeat types in trace T'_i.
7: end for
8: Let $\mathcal{PTR} = \bigcup_{i=1}^{n} \mathcal{PTR}_i$.
9: Find abstractions over the set of repeat alphabets of $\mathcal{PTR}$. Let $\mathbb{A}$ be the set of such abstractions. For each abstraction $\mathfrak{a}_i \in \mathbb{A}$, there exist a set of repeats that constitute the abstraction. Let $f : \mathcal{PTR} \rightarrow \mathbb{A}$ be the function defining the abstraction for each repeat.
10: {Process the traces and replace the loop manifestation with abstract entities}
11: for all $T'_i \in \mathcal{L}'$ do
12: Let T''_i be an empty trace
13: for $j = 1$ to $|T'_i|$ do
14: if there exits a maximal primitive tandem array (j, α, k), $\alpha \in \mathcal{PTR}, k \geq 1$ then
15: {check whether there exist any larger tandem array overlapping with this one}
16: if there exist another primitive tandem array (j', β, k') such that $|\beta| > |\alpha|$ and $j' \leq j + k * |\alpha|$ then
17: Set $k = \lfloor (j' - j)/|\alpha| \rfloor$.
18: end if
19: Append $f(\alpha)$ to T''_i
20: Set $j = j + k * |alpha|$
21: else
22: Append $T'_i(j)$ to T''_i
23: end if
24: end for
25: $\mathcal{L}'' = \mathcal{L}'' \cup \{T''_i\}$
26: end for
27: Find conserved regions across all traces in $\mathcal{L}''$
28: Let $\mathcal{CR}$ be the set of conserved regions
29: Find abstractions over the set of repeat alphabets of $\mathcal{CR}$. Reuse abstractions already defined over $\mathcal{PTR}$ for repeats that are common to both $\mathcal{PTR}$ and $\mathcal{CR}$. Let $\mathbb{A}'$ be the set of complete set of abstractions. For each abstraction $\mathfrak{a}_i \in \mathbb{A}'$, there exist a set of repeats that constitute the abstraction. Let $g : \mathcal{CR} \rightarrow \mathbb{A}'$ be the function defining the abstraction for each repeat.
30: Process the traces and replace the conserved regions with abstract entities

Algorithm 1: Single-phase preprocessing of traces

The algorithm is straightforward but the steps $14 - 20$ pertaining to the treatment of overlapping loop manifestations deserve attention. Fig 5 depicts a scenario. $(1, \texttt{ab}, 5)$ and $(9, \texttt{abcd}, 3)$ are two tandem arrays in the trace. The prefix of the second loop manifestation overlaps with the suffix of the first loop. In this case, we shorten the first tandem array to $(1, \texttt{ab}, 4)$ and give preference to longer patterns.

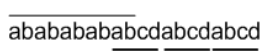


Fig. 5: Overlap of primitive tandem arrays

7.2 Iterative approach to resolve complex constructs

Consider the process model depicted in Figure 4(iv) which consists of a parallelism construct within a loop. Consider two traces $T_1 =$ `abcdeacbdeabcde` and $T_2 =$ `acbdeabcde`. $r_1 =$ `abcde` and $r_2 =$ `acbde` constitute the maximal repeats present in the two traces (obtained in the concatenated sequence of T_1 and T_2). Now, repeats r_1 and r_2 are equivalent under the repeat alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$. Let us represent this equivalence class with an abstract entity, say `A`. Processing traces T_1 and T_2 and replacing all occurrences of repeats within this equivalence class with the abstract entity, we get the transformed traces $T_1' =$ `AAA` and $T_2' =$ `AA`. In the second iteration of preprocessing, the loop construct can be discovered in the transformed traces.

Loops within loops can also be discovered using a multi-phase approach. Consider Fig 4(ii). $T_1 =$ `abcdabcbcbcd`, $T_2 =$ `abcbcd` are two of the traces pertaining to the construct. $(6, \mathtt{bc}, 3)$ and $(2, \mathtt{bc}, 2)$ would be identified as tandem arrays in traces T_1 and T_2 respectively. Let us assume that the tandem repeat type `bc` is represented with an abstract entity `A`. Replacing all occurrences of tandem arrays with tandem repeat type `bc` in the log with the abstract entity, we get the transformed traces $T_1 =$ `aAdaAd` and $T_2 =$ `aAd`. Now in the second iteration, `aAd` would be identified as a tandem array. Thus loops within loops can be uncovered. Though this example relates to simple loops within loops, iterative approach of identifying loops and conserved regions (both *exact* and *approximate*) alternatively and performing consistent abstractions over them would help in realizing more complex constructs.

8 Experimental Results and Discussion

We have analyzed the significance of the patterns described in this paper over a large set of event traces (of real systems) with varying alphabet sizes. We present one such study here where we have considered a set of 1372 event traces of a health care system. The traces correspond to the commands of clinical usage logged by the system. There were a total of 213 distinct commands (activities/event classes) in the event log (alphabet size, $|\mathcal{A}| = 213$) and the entire event log had $215,623$ events.

In this study, we have done analysis only on the exact repetitions. The analysis of approximate repetitions (both for the manifestation of loop constructs and conserved regions) is underway. Table 3 depicts a few examples of primitive tandem repeat types identified in the log. It can be seen that the commands involved in the loop manifestation all belong to a common functionality. The primitive tandem repeat types 1 and 2 correspond to some *image processing* functionality. The difference between 1 and 2 being that in the former, an *image reverse* operation is performed where as in the latter an *image forward* operation is invoked. The primitive tandem repeat type 3 corresponds to a functionality of *beam/detector* movement while that of 4 corresponds to a *wedge* movement functionality. Primitive tandem repeat type 5 corresponds to *geometry* functionality. There were a total of 826 primitive tandem repeat types in the event log.

The shortest primitive tandem repeat type is of length 1 (signifying a loop over a single activity/command) while the longest spans over 13 activities. Under the equivalence class of repeat alphabets, the number of distinct classes were 363.

Table 3: A few examples of primitive tandem repeat types

S.No	Primitive Tandem Repeat Type	Frequency
1	(SetReplayScope, SetReplayType, SetSpeedAndDirection, StartReplay, StartStepImgRev, StopStepImgRev, StartStepRunFwd, StopStepRunFwd)	206
2	(SetReplayScope, SetReplayType, SetSpeedAndDirection, StartReplay, StartStepRunFwd, StopStepRunFwd, StartStepImgFwd, StopStepImgFwd)	319
3	(MoveDetectorLateral.Move, AngulateBeamLateral.Move, RotateBeamLateral.Move, AngulateBeamLateral.Move)	43
4	(BLWedge2RotateClockwise, BLWedge2TranslateIn, BLWedge2TranslateStop, BLWedge2RotateStop)	51
5	(ResetGeo.Start, ResetGeo.Stop)	85

There were a total of 170 maximal elements in the Hasse diagram on the repeat alphabet over the primitive tandem repeat types. A lot of these maximal elements were found to be similar. For example, the three maximal element repeat alphabets, {`StartStepImgRev`, `StopStepImgRev`, `AngulateBeamFrontal.Move`}, {`StartStepImgRev`, `StopStepImgRev`, `MoveDetectorFrontal.Move`} and {`StartStepImgRev`, `StopStepImgRev`, `RotateBeamFrontal.Move`} are all similar in that there exist some beam limitation related movements in conjunction with some image analysis. Using extended joins, these three maximal elements can be combined to a extended repeat alphabet {`StartStepImgRev`, `StopStepImgRev`, `AngulateBeamFrontal.Move`, `RotateBeamFrontal.Move`, `MoveDetectorFrontal.Move`}. The number of abstractions can be reduced thus.

Table 4 depicts some of the examples of near super maximal repeats (*nsm*) over the data set. It can be clearly seen that all image processing related commands used as a functionality are captured in *nsm* 1. The *nsm* 2 pertains to wedge related movements while *nsm* 3 depicts the zoom functionality used in succession/conjunction with the image processing functionality which can be easily imagined from an application point of view. In the data set there were a total of $26,000$ near super maximal repeats (5391 repeat alphabets) after the first iteration. Though the numbers sound large, a lot of these are similar which can be seen from the fact that there were only 1129 maximal elements in the Hasse diagram on these repeat alphabets. Using (δ_c and δ_d) as parameters, these can further be reduced using extended joins. Using one such parameter setting, we were able to reduce the number of abstractions to 40.
Fig 6 depicts a process model mined using the heuristics miner in the ProM tool on the abstracted log of Philips Healthcare. The original log was first filtered to remove highly infrequent activities (frequency of occurrence less than 0.005%). This resulted in a log with 141 distinct event classes and $215,399$ total number of events. The abstractions are defined over the exact tandem array patterns capturing the manifestation of loop constructs. For the abstracted log we con-

Table 4: Few examples of near super maximal repeats over the dataset

S.No	Near Super Maximal Repeats
1	(StartStepRunFwd, StopStepRunFwd, SetSpeedAndDirection, StartReplay, StartStepRunFwd, StopStepRunFwd, StartStepImgFwd, StopStepImgFwd, StartStepRunRev, StopStepRunRev, SetSpeedAndDirection, StartReplay, StartStepImgFwd, StopStepImgFwd, StartStepRunFwd, StopStepRunFwd, SetSpeedAndDirection, StartReplay, StartStepRunFwd, StopStepRunFwd)
2	(BLWedge1TranslateIn, BLWedge1RotateClockwise, BLWedge1RotateStop, BLWedge1TranslateStop, BLWedge1TranslateIn, BLWedge1RotateCounterClockwise, BLWedge1RotateStop, BLWedge1TranslateStop, BLWedge1RotateCounterClockwise)
3	(StartStepImgRev, StopStepImgRev, StartStepImgFwd, StopStepImgFwd, SetZoomFactor, SelectView, SetZoomCentre)

ducted only one iteration of pre-processing. The process model from the original log (without the abstraction) is shown in Fig 1(a). It is evident that the process model mined from the abstracted log is more comprehensible (less spaghetti-like). Further, the abstractions were formed over activities that are related by a functionality. For example, all the shutter movement operations were grouped to an abstract entity. Similarly, wedge related movements, commands pertaining to image processing functionality have been grouped as an abstract entity. In other words, we were to able to identify *conceptually-valid* abstractions.

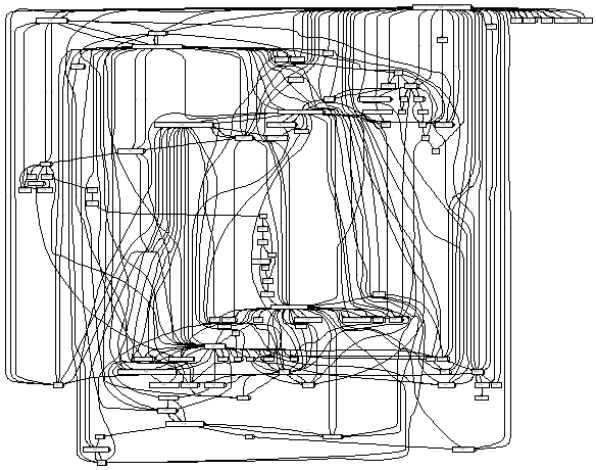


Fig. 6: Process model mined using heuristics miner on log abstracted with loop construct manifestations

The approximate notions of patterns induces the flexibility and thereby the variety over the class of patterns. Another notion of flexibility is introduced in the approach for abstraction over the repeat pattern alphabet. Recall that we have introduced the notion of parameterized extended joins (over the fraction of common/different elements between two repeat alphabets). By choosing different thresholds for the approximation (similarity) between patterns, one can form

a multi-level abstraction. The assumption that we make over the definition of these patterns is that each functionality (process model construct, sub-process etc) gets manifested at least twice in the event log (either within the same trace or across traces), which is a reasonable assumption to make. While the notion of approximate patterns induces the flexibility, it also acts as a weakness, as choosing a right notion of approximation is non trivial. For the edit-distance based approximation, choosing a right cost function for edit operations is critical. However, it can be mitigated with approaches for automated derivation of costs such as in [4].

9 Related Work

Greco et al [11], [12] proposed an approach to mine hierarchies of process models that collectively represent the process at different levels of granularity and abstraction. The basic idea of their approach is to cluster the event log into different partitions based on the homogeneity of traces and mine process models for each of the clusters. Clustering induces a hierarchy in the form of a tree with the root node depicting the entire log and the leaf nodes corresponding to traces pertaining to concrete usage scenarios. Two kinds of abstractions over activities viz., *is-a* and *part-of* is then done by traversing the tree bottom-up and considering every pair of activities and checking whether they can be *merged* without adding too many spurious control flow paths among the remaining activities. This approach tries to analyze the mined process models (post-processing) for identifying activities that can be abstracted. However, for large complex logs, the mined process models (even after clustering) can be quite spaghetti-like thereby increasing the complexity of such analysis. In contrast, the approach proposed in this paper analyzes the raw traces and defines abstraction (pre-processing) and thereby reduces the spaghettiness of the mined process model. Our approach can be used complementarily as a precursor to [11], [12]. It is conjectured that such a hybrid approach will yield better results. Polyvyanyy et al [13] have proposed a slider approach for enabling flexible control over various process model abstraction criteria (such as activity effort, mean occurrence of an activity, probability of a transition etc.). The slider is employed for distinguishing significant process model elements from insignificant ones. Taking cartography as a metaphor, Günther and Aalst [14] have proposed a process mining approach to deal with the "spaghettiness" of less structured processes. The basic idea here is to assign significance and correlation values to activities and transitions, and depicting only those edges/activities whose significance/correlation is above a certain threshold. Less significant activities/edges are either removed or clustered together in the model. Günther and Aalst [14] too have used a slider based approach to specify the threshold and thereby alter the levels of abstraction. Approaches such as [13], [14] looks at abstraction from the point of retaining highly significant information and discarding less significant ones in the process model. In constrast, the approach proposed in this paper looks at abstraction from a functionality point of view. The approach proposed in this paper can be used as a preprocessing step for the logs and can be seamlessly integrated with other approaches for abstraction [11], [12], [14] as well as with approaches for process discovery.

10 Conclusions and Future Work

In this paper, we have presented a few pattern definitions and correlated them to the manifestation of process model constructs. We have also presented an approach to form abstractions of activities in the log based on the patterns. Further, a multi-phase approach for pre-processing the traces with the patterns and abstractions was presented. We have applied the proposed techniques on a real-life log and the results are promising. The pattern definitions proposed in this paper have multi-faceted applications such as enabling of hierarchical process mining (thereby reducing the spaghettiness of mined models), trace clustering and fault diagnosis.

Acknowledgments: The authors are grateful to Philips Healthcare for funding the research in Process Mining.

References

1. van der Aalst, W., Weijters, A., Maruster, L.: *Workflow Mining: Discovering Process Models from Event Logs.* IEEE Trans. Knowl. Data Eng. **16**(9) (2004) 1128–1142
2. de Medeiros, A.K.A., van der Aalst, W., Pedrinaci, C.: *Semantic Process Mining Tools: Core Building Blocks.* In: 16th European Conference on Information Systems. (2008) 1953–1964
3. Ristad, E.S., Yianilos, P.N.: *Learning String-Edit Distance.* IEEE Trans. PAMI **20**(5) (1998) 522–532
4. Bose, R.P.J.C., van der Aalst, W.: *Context Aware Trace Clustering: Towards Improving Process Mining Results.* In: SIAM International Conference on Data Mining. (2009) 401–412
5. Gusfield, D.: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press (1997)
6. Kolpakov, Kucherov: *Finding Maximal Repetitions in a Word in Linear Time.* In: IEEE Symposium on Foundations of Computer Science (FOCS). (1999) 596–604
7. Cheung, C.F., Yu, J.X., Lu, H.: *Constructing Suffix Tree for Gigabyte Sequences with Megabyte Memory.* IEEE Trans. Knowl. Data Eng. **17**(1) (2005) 90–105
8. Gusfield, D., Stoye, J.: *Linear Time Algorithms for Finding and Representing all the Tandem Repeats in a String.* Journal of Computer and System Sciences **69** (2004) 525–546
9. Sokol, D., Benson, G., Tojeira, J.: *Tandem Repeats Over the Edit Distance.* Bioinformatics **23**(2) (2007) e30–e36
10. Ukkonen, E.: *On-Line Construction of Suffix Trees.* Algorithmica **14**(3) (1995) 249–260
11. Greco, G., Guzzo, A., Pontieri, L.: *Mining Hierarchies of Models: From Abstract Views to Concrete Specifications.* In: Business Process Management. (2005) 32–47
12. Greco, G., Guzzo, A., Pontieri, L.: *Mining Taxonomies of Process Models.* Data Knowl. Eng. **67**(1) (2008) 74–102
13. Polyvyanyy, A., Smirnov, S., Weske, M.: *Process Model Abstraction: A Slider Approach.* In: Enterprise Distributed Object Computing. (2008) 325–331
14. Günther, C.W., van der Aalst, W.M.P.: *Fuzzy Mining - Adaptive Process Simplification Based on Multi-perspective Metrics.* In: Business Process Management. (2007) 328–343