Trace retrieval for business process operational support

Trace Retrieval for Business Process Operational Support

Alessio Bottrighi[a], Luca Canensi[b], Giorgio Leonardi[a,*], Stefania Montani[a], Paolo Terenziani[a]

[a] *DISIT, Computer Science Institute, Università del Piemonte Orientale, Alessandria, Italy*

[b] *Department of Computer Science, Università di Torino, Italy*

Abstract

Operational support assists users while process instances are being executed, by making predictions about the instance completion, or recommending suitable actions, resources or routing decisions, on the basis of the already completed instances, stored as execution traces in the event log.

In this paper, we propose a case-based retrieval approach to business process management operational support, where log traces are exploited as cases. Once past traces have been retrieved, classical statistical techniques can be applied to them, to support prediction and recommendation. The framework enables the user to submit queries able to express complex patterns exhibited by the current process instance. Such queries can be composed by several simple patterns (i.e., single actions, or direct sequences of actions), separated by delays (i.e., actions we do not care about). Delays can also be imprecise (i.e., the number of actions can be given as a range). The tool also relies on a tree structure, adopted as an index for a quick retrieval from the available event log.

Our approach is highly innovative with respect to the existing literature panorama, since it is the first work that exploits case-based retrieval techniques in the operational support context; moreover, the possibility of

*Corresponding author. Tel. +39 0131 360340

Email addresses: alessio.bottrighi@uniupo.it (Alessio Bottrighi), canensi@di.unito.it (Luca Canensi), giorgio.leonardi@mfn.unipmn.it (Giorgio Leonardi), stefania.montani@uniupo.it (Stefania Montani), terenz@di.unito.it (Paolo Terenziani)

retrieving traces by querying complex patterns and the indexing strategy are major departures also with respect to other existing trace retrieval tools proposed in the Case Based Reasoning area.

Thanks to its characteristics and methodological solutions, the tool implements operational support tasks in a flexible and efficient way, as demonstrated by our experimental results.

Keywords: Trace retrieval, Case Based Reasoning, Operational Support

---

1. Introduction

Operational support is a process management activity meant to assist users while process instances are being executed (see der Aalst (2011), chapter 9). If an instance is still running, it is possible that information about already completed instances can be exploited to ensure the correct or efficient handling of the current instance itself. Completed instances are stored in the so-called *event log*, which records the sequences (*traces* henceforth) of actions executed at a given organization, typically together with action execution parameters (e.g., times, costs, resources).

Specifically, operational support can be articulated into three main tasks:

- *detection*: compares an existing process model to the current instance data (i.e., the information about the already executed actions in the current trace). If the model control flow (or other predefined rule) is violated, an alert is generated;
- *prediction*: exploits the event log and the current instance to make predictions about the instance completion (e.g., time to completion, costs, use of resources, possible problems);
- *recommendation*: uses the event log and the current instance to recommend suitable actions, resources or routing decisions (in order, e.g., to minimize costs or time to completion).

While *detection* is more concerned with constraint satisfaction and with the exploitation of an existing normative model, *prediction* and *recommendation* heavily rely on experiential knowledge, stored in the event log in the form of past process traces.

Traditional operational support tools, like those in the ProM (van Dongen et al., 2005) framework, operate by building a transition system and by replaying log traces on it (van Dongen, 2007).

However, also a direct use of the experiential knowledge in the log can provide crucial contributions to support prediction and recommendation. In particular, analogical reasoning (as enabled, e.g., by the Case Based Reasoning (CBR) (Aamodt & Plaza, 1994) methodology) can be adopted to this end.

The CBR methodology is articulated into four steps: (1) retrieve past experiential knowledge, in the form of past "cases", similar to the current input situation; (2) reuse the retrieved case solution(s) (possibly after an adaptation procedure), to solve the input situation; (3) revise the newly solved case for correctness; (4) retain the new case, to enrich the system knowledge base.

In this paper, we propose a **case-based retrieval** framework for operational support. We therefore focus on the first step of the CBR methodology.

In detail, we represent log traces as cases, and, given an ongoing process instance as an input, we retrieve past traces similar to it. Once past traces have been retrieved, classical statistical techniques can be applied to them, to support prediction and recommendation. For instance, the percentage of retrieved traces that, e.g., were completed on time, can be used to calculate the probability that the current instance will complete on time too. A similar approach can be adopted to estimate costs, or predict problems. Moreover, the best actions to execute next can also be extracted from the retrieved traces. In the following, we will however concentrate on trace retrieval, being the discussion on statistical techniques out of the scope of this paper.

The approach we propose is, to the best of our knowledge, the first one to provide such a direct exploitation of experiential knowledge in the context of operational support. Notably, trace retrieval is, in general, a quite challenging and complex task. The input process instance is a partial execution trace, and the easiest form of retrieval can be achieved by looking for all the traces in the log that contain the input trace as a prefix. However, in many applications, more flexibility is required. Indeed, the user may observe that the input instance contains specific patterns (i.e., consecutive or non-consecutive sequences of actions), able to strongly influence the next actions to be executed, or the overall performances (e.g., completion time); patterns can be rather complex, and involve temporal indeterminacy. Therefore, s/he may want to abstract from the specificities of the input trace, by identifying

these patterns, and looking for all the traces that match them, in order to get a recommendation on how to deal with these problems on the basis of past experience.

The medical domain, among others, includes many situations where this non-trivial trace retrieval can provide a valuable help to tackle complicated clinical cases. For example, in the treatment of stroke, particular attention must be paid to suspected Subarachnoid Hemorrhage (SAH). The latest clinical guidelines[1] suggest performing a Lumbar Puncture (LP) to patients where the SAH is suspected but not confirmed by the proper laboratory examinations, in order to investigate the presence of blood in the cerebrospinal fluids. However, LP is an invasive procedure and not all the patients can tolerate it without paying some consequences. For this reason, the chance of executing it must be carefully evaluated, and the use of information collected from the past similar situations can help in deciding whether or not the exam should be performed.

Let us consider a particular case, where a suspicion of SAH arises for a non-epileptic patient who experiences a sudden epileptic attack. Once the patient has been transported to the hospital, specific tests are performed, such as Transcranial Doppler (TD) and a subsequent Cerebral Angiography (CA). If the results of these tests do not clearly indicate the presence of SAH (i.e., there is a positive TD report, then a negative CA report), the clinical guidelines suggest the execution of a Computerized Tomography (CT) scan as a further exam. If also the TC scan report is negative, the differential diagnosis could be conducted through the observation of the spinal fluid, to check whether the blood has penetrated into the spinal liquid. However, as mentioned before, this procedure is invasive and stressful for patients affected by serious conditions, but it could be important in order to gain a definitive and certain answer for assessing the actual presence of SAH. This assessment is also important because it can radically alter the treatment plan to be administered to the patient.

In order to estimate more accurately the cost/benefit ratio of performing the SP, it is very useful to look at what has been done in the past, by retrieving all the execution traces which follow a query pattern which starts with an epileptic attack, followed by a positive TD report, then by a negative CA report and finally by a negative TC report. No matter if and when other

[1] http://www.iso-spread.it/, last accessed on September, the 30th, 2015

medical procedures appear among the diagnostic exams mentioned before: stroke patients typically undergo many other routine tests (such as blood test or chest RX) that are not specifically relevant for SAH, but may be interleaved between the key actions. Trace retrieval by querying complex patterns as the one illustrated in this example is supported in our framework.

It is then possible to analyze the retrieved traces in order to calculate: the percentage of patients who received the SP; the percentage of patients who tolerated this procedure; how many times the execution of the SP allowed for a correct differential diagnosis, and what therapeutic plans have been set up depending on the different cases. On the basis on this information, an appropriate cost/benefit analysis can be conducted, in order to decide whether or not the SP should be executed, and to obtain some suggestions about the most common treatment plans to be administered to the patient.

Interestingly, our framework also relies on a tree structure, called *trace tree*, allowing for a quick retrieval, by avoiding a flat search for all the traces in the log that satisfy the input pattern. The trace tree is a sort of "model" of the traces, that we learn using a process mining technique we recently implemented (Canensi et al., 2014). As we will see in this paper, the *trace tree* can be used as an index of the traces in the log, supporting also the search of traces corresponding to complex query patterns. It is worth noting that the work in Canensi et al. (2014) only dealt with the construction of the process model, i.e., the tree structure, but did not propose any exploitation of it as an index for trace retrieval. As such, the research objective of the present paper, fully focused on trace retrieval for operational support, and the content of all the technical sections (except Section 2.1, which summarizes the approach in Canensi et al. (2014)), are completely new with respect to our previous work.

In synthesis, in this paper we describe a flexible and efficient case-based retrieval approach, which also allows to query complex pattern to be searched for in the traces. Our approach is highly innovative with respect to the existing literature panorama, in that:

- we propose, to the best of our knowledge, the first work that exploits case-based retrieval techniques on the event log in the operational support context;
- we not only support exact prefix retrieval, but also non-trivial trace retrieval, in which complex query patterns are looked for;

- we take advantage of the trace tree structure to speed up the retrieval process.

The latter two points represent major departures also with respect to other existing trace retrieval tools proposed in the CBR area (see Section 4).

The paper is organized as follows. In Section 2 we technically describe our approach. For the sake of completeness, we first briefly summarize our algorithm to build the log tree (see Section 2.1), which acts like an index of the traces. We then move to the novel technical contribution of this paper, describing our query language and retrieval algorithms. In Section 3 we present our experimental results. In Section 4 we discuss related work. In Section 5 we present our concluding remarks and future work directions.

2. Methods

This Section presents the details of our approach. In particular, in Section 2.1 we describe the trace tree structure, and sketch the algorithm to build it, which was extensively illustrated in Canensi et al. (2014). In Section 2.2, which represents the core novel technical contribution of this paper, we illustrate our retrieval approach.

2.1. Mining the trace tree

Our framework relies on a tree structure, called trace tree, allowing for an efficient retrieval of the traces that satisfy the input pattern. The trace tree is a sort of "model" of the traces, that we learn using a process mining technique we recently implemented (Canensi et al., 2014), and built in such way that it can be used as an index.

Several approaches to process mining exist in the literature (e.g., those in ProM (van Dongen et al., 2005)), but, despite some differences, many of them show important similarities, and have common limitations (see also (Cnudde et al., 2014)):

- they learn "context-free" patterns of processes;
- they can mine paths that do not correspond to any input trace in the log (i.e., they can have a limited *precision* (Buijs et al., 2012));
- they do not explicitly relate the mined patterns to the log (in the sense that there is no explicit correspondence between mined patterns, and the traces in the log "supporting" them).

Such limitations are quite relevant in general, and very relevant in some specific domains, such as the medical one.

Concerning the first limitation, it is well known that, e.g., the same (set of) actions may produce different effects, depending on the context (e.g., on the medical actions previously performed on the patient).

The impact of the second limitation is obvious and dramatic: if the miner precision is limited, in the sense that it may also learn a path that never appears in any input trace, this can be very harmful in all those applications where it is vital that mining results are reliable as much as possible. However, surprisingly, limited precision is a common limitation of many current miners.

The third limitation is less critical, but still significant. Indeed, maintaining an explicit link between mined patterns and the input traces matching such patterns, can be important not only to characterize contexts, but also to provide users with an evidence of the learned output, and also to provide support for retrieving traces corresponding to a given pattern - which is our current objective.

In Canensi et al. (2014), we therefore proposed an innovative approach, able to support support "context-aware" process mining, and overcome all the above limitations. The technical details of the approach are summarized below.

Our mining algorithm takes in input an event log. The event log is stored as a matrix with n rows and m columns, where n is the number of traces in the log and m is the maximum length of these traces.

Each cell $Matrix[i, j]$ contains the j-th action of the trace i. Actions in the different traces are aligned on the basis of their order of execution (i.e., the j index). All traces start with a dummy common action #.

The algorithm outputs the mined process as a *trace tree*, where nodes represent actions, and arcs represent a control flow (i.e., precedence, XOR choice) relation between them. Indeed, we exploit the temporal ordering of actions in the log traces to mine the process model, maintaining all the local and global ordering relationships between the actions.

More precisely, in the trace tree, each node is represented as a pair $< P, T >$.

P denotes a (possibly unary) set of actions; actions in the same node are in AND relation, or, more properly, may occur in *any order* with respect to each other. Note that, in such a way, each path from the starting node of the tree to a given node N denotes a set of possible process patterns (called *support patterns* of N henceforth), obtained by following the order represented by

the arcs in the path to visit the trace tree, and ordering in each possible way the actions in each node (for instance, the path $\{A, B\} \rightarrow \{C\}$ represents the support patterns "ABC" and "BAC").

T represents a set of pointers to all and only those traces in the log whose prefixes exactly match the path from the root to one of the patterns in P (called *support traces* henceforth). Specifically, prefixes correspond to the entire traces if the node at hand is a leaf. In the case of a node representing a set of actions to be executed in any order, T is more precisely composed of several sets of support traces, each one corresponding to a possible action permutation. This choice enhances retrieval performances, as we will discuss in section 2.2.2.

Algorithm 1 below builds the trace tree.

ALGORITHM 1: Mining pseudocode

```
1 Build-Tree (index,< P, T >) ;
2 nextP ← getNext(index+1, T) ;
3 if nextP not empty then
4 |   nextActions ← XORvsAND (nextP, T) ;
5 |   foreach node < P′, T′ > ∈ nextActions do
6 |   |   AppendSon(< P′, T′ >,< P, T >) ;
7 |   |   Build − Tree(index+ |P′|,< P′, T′ >) ;
8 |   end
9 end
```

The function *Build-Tree* in Algorithm 1 takes in input a variable *index*, representing a given position in the traces (i.e., a column in the input matrix), and a node. Initially, it is called on the first position, and on the root of the tree (which is a dummy node, corresponding to the # action; thus, initially, *index*=0, P=# and T is the set of all the traces). The function *getNext* simply inspects the traces in T to find all possible next actions. On these actions, the function *XORvsAND* applies the formula below in order to identify which actions are in AND and which are in XOR relation: we calculate the *dependency frequency* $A \rightarrow B$ between every action pair $< A, B >$ in $nextP \times nextP$:

$$A \rightarrow B = \frac{1}{2}\left(\frac{|A > B|}{\sum_{X \in Act_T} |A > X|} + \frac{|A > B|}{\sum_{Y \in Act_T} |Y > B|}\right) \qquad (1)$$

where, always considering the traces in T, $|A > B|$ is the number of traces in which A is immediately followed by B, $|A > X|$ is the number of traces in which A is immediately followed by some action X (with $X \in Act_T$, being Act_T the set of all the actions appearing in the traces in T), and $|Y > B|$ is the number of traces in which B is immediately preceded by some action Y (with $Y \in Act_T$). After evaluating the dependency frequency value $A \rightarrow B$ and $B \rightarrow A$, we can have 3 possible situations:

- if both the values are below a given threshold, this means that A and B rarely appear in the same trace, therefore they are in XOR relation;
- if $A \rightarrow B$ is above the threshold and $B \rightarrow A$ is below, then A precedes B, and vice versa;
- if both the values are above the threshold, then A and B are in AND (any-order) relation.

The output *nextActions* of the function *XORvsAND* is a set of nodes $< P', T' >$, one for each maximal set of actions to be AND-ed. *P'* is therefore a set of action in AND (and a subset of P). Note that, for each one of such sets *P'*, the corresponding set *T'* of *support traces* where these actions in AND take place is also computed, as a subset of T.

Finally, each new node is appended in the output tree (function *Append-Son*), and *Build-Tree* is recursively applied to each node (with the parameter *index* properly set, and passing $< P', T' >$ as the last parameter).

2.2. Trace retrieval

In this Section, we first describe our query language. Then, we present the retrieval algorithm, specifying how traces satisfying a query can be retrieved, taking advantage of the trace tree structure.

2.2.1. Query language

In our framework, the user can issue a query, composed of one or more simple patterns to be searched for. In turn, simple patterns are defined as one or more actions in direct sequence. Multiple simple patterns can be combined in a complex pattern, by separating them by delays. A delay is a sequence of actions between two simple patterns; the semantics is that we do not care about these actions, so they will not be specified by the query. Instead, only their number will be provided, possibly in an imprecise way

(i.e., we allow the user to express the number of actions as a range). Formally, a query is represented in the following format:

$$\langle (min_1, max_1)SP_1 ... (min_k, max_k)SP_k(min_{k+1}, max_{k+1})\rangle \tag{2}$$

where:

- SP_j is a simple pattern (i.e. a sequence of symbols, representing the actions we are looking for; these actions have to be in direct sequence);
- (min_j, max_j) is the delay between two items (i.e., two simple patterns, or a simple pattern and the trace starting/ending point), expressed as a range in the number of actions.

As an example, the query

$$\langle (0,0)B(0,1)E(2,2)Z(0,1)\rangle \tag{3}$$

looks for action B, which has to start at the very beginning of the trace. This first simple pattern B must be followed (with zero or a single action in between) by action E. E must be followed by two actions, which we do not care about; after them, Z is required. Z can be the final action, or can be followed by one additional action we do not care about.

It is worth noting that a query written as above corresponds to a whole set of queries, each one obtained by choosing a specific delay value and specific actions in each of the (min_j, max_j) intervals.

Every query in this set can be made partially explicit as a string, containing as many dummy symbols $*$ as needed, to cover the corresponding delay length (where the dummy symbol is chosen because we are not interested in the specific actions).

For example, the query above would correspond to the following four partially explicit queries, whose length ranges from 5 to 7 actions (not counting the intial dummy action #), where the dummy symbol $*$ has been properly inserted, according to the delay values information:

$BE**Z$; $BE**Z*$; $B*E**Z$; $B*E**Z*$

Since each $*$ could be substituted by any of the N types of actions recorded in the log and/or existing in the application domain, the example query corresponds to $N^2 + 2 * N^3 + N^4$ totally explicit queries.

The problem is obviously combinatorial, with respect to the possible delay ranges and action types. We thus believe that extensional approaches (in

which only explicit queries can be issued) would not be feasible in many domains. Our query language, allowing for compact "intensional" queries, is therefore a significant move in the direction of implementing an efficient and user-friendly operational support tool.

Notably, exact prefix retrieval can be seen as a special case of the more general retrieval possibilities we offer, where only a single pattern is provided, and no delays are needed.

2.2.2. Retrieval

In order to retrieve the log traces that satisfy a query, we have implemented a multi-step procedure, articulated as follows:

- automaton generation;
- tree search;
- filtering.

In the following, we will provide the details of the various steps, along with a running example based on the example query introduced in section 2.2.1, exploiting the trace tree in Figure 1.

Automaton generation In our approach to trace retrieval, we first generate a deterministic automaton, that represents the query at hand. To build the automaton, we implement the following procedure:

- (A) transform the query into a regular expression;
- (B) apply the Berry & Sethi (1986) algorithm, to build a non-deterministic automaton that recognizes the regular expression above;
- (C) unfold the non-deterministic automaton;
- (D) transform the unfolded non-deterministic automaton into a deterministic automaton (Lam et al., 2006).

Steps (A) and (D) are trivial. As regards Step (A) note that our query language is just a variation of regular expressions, useful to express delays and "do not care" (i.e., dummy) symbols in a compact way. The cost of Step (A) is linear in the number of delays used in the query. Steps (B) and (C) use classical algorithms in the area of formal languages. The cost of Step (B) is linear in the number of symbols in the query expressed as a regular

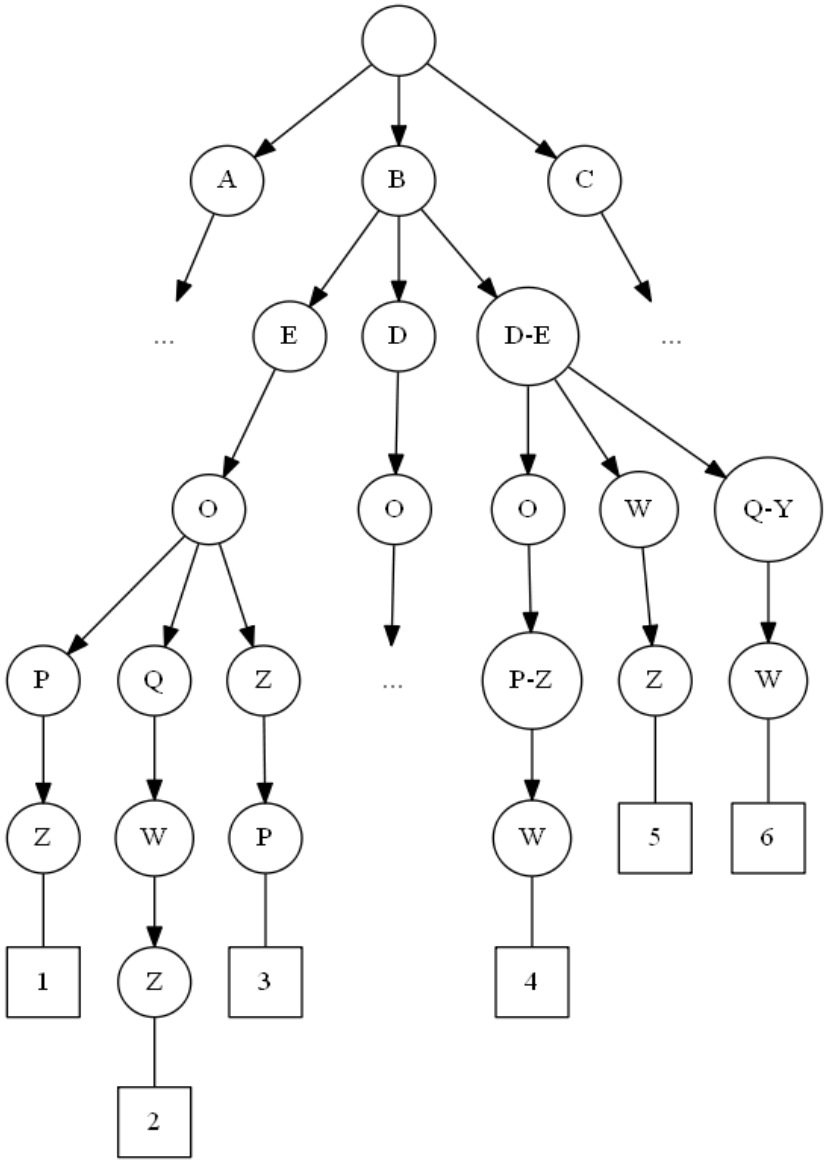

Figure 1: Trace tree in the example.

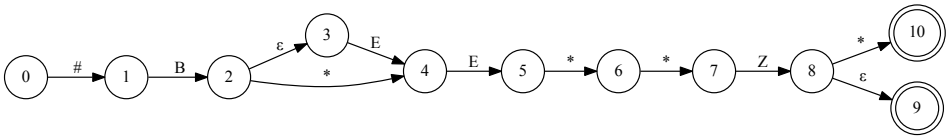

Figure 2: Non deterministic automaton in the example. On the edges: symbol consumed in the corresponding transition; $*$ means that any symbol can be consumed; ε means that no symbol is consumed.

expression (Berry & Sethi, 1986) (i.e., the output of Step (A)), and the cost of Step (C) is the product between the number of dummy symbols in the query and the cardinality of the action symbols available in the log. Step (D) substitutes each arc labeled by the dummy symbol in the automaton with a set of arcs, one for each action in the event log. Although in the worst case Step (D) is exponential with respect to the number of states in the automaton (i.e., the output of Step (B)), note that the worst case is rare in practice (van Leeuwen, 1994).

Example Referring to our example query, the generated non-deterministic automaton (see Fig. 2) has 9 states and 10 transitions; the state 0 is the start state and states 8 and 9 (double circled in Fig. 2) are the accepting states.

Tree search Once the deterministic automaton has been obtained, it would be possible to exploit it in a classical way, by providing all event log traces in input to it, to verify which of them satisfy the query. However, some of these traces may be identical, or share common prefixes of various length, so that the straightforward approach would lead to repeated analyses of the common parts.

In order to optimize efficiency, we have therefore proposed a novel approach, that provides the trace tree as an input to the automaton. Each path in the trace tree may index several identical support traces, that will be considered only once, thus speeding up retrieval with respect to a flat search into the event log. Moreover, in the tree, common prefixes of different traces are represented just once, as common branches close to the root (different postfixes can then stem from the common branches, to reach the various leaves). These common parts will be executed on the automaton only once,

without requiring for repeated, identical checks.

It is worth noting that providing a tree as an input to the automaton represents a significantly novel contribution, since in the formal languages literature the input to be executed on the automaton is typically a string. The work in Baeza-Yates & Gonnet (1996) represents an exception, but the tree it exploits (a Patricia tree) has very different semantics (and usage) with respect to ours.

In detail, our approach operates as follows: the algorithm *Search_Process* (see Algorithm 2) takes in input the trace tree T and the deterministic automaton A, and provides as an output a set of pairs, composed of a trace tree leaf node and a corresponding string. Each of these strings is an explicit instantiation of the query represented by the automaton, verified by (some of) the support traces in the leaf node. The output support traces can then be retrieved from the event log and presented to the user, or provided as an input to the filtering step.

Basically, *Search_Process* executes a breadth first visit of the trace tree; it exploits the variable *searching*, defined as a set of triples, composed by a trace tree node, an automaton state, and the string that has been recognized on the automaton so far. Initially (lines 5-7), *searching* contains the sons of the root (since the root is a dummy action #, see section 2.1), all paired to the initial state of the query automaton and to the empty string. The visit procedure (lines 8-35) extracts one triple at a time from the set *searching*. If the *node* in the triple contains a set of actions to be executed in any order (line 11), we simulate all the permutations on the automaton, and save the states we reach and the corresponding recognized strings into the *new_states* set (line 12). If the node contains one single action, we simply simulate it on the automaton, and save the state we reach and the corresponding string into the *new_states* set (line 15). In both cases, the string saved in *new_states* is the one in the input triple properly updated with the newly recognized symbols.

After the simulation, if the *node* at hand is a leaf (line 18), then for each item in *new_states* we check whether the state component is a final state (lines 19-20); if this is the case, *node* and the string paired to the final state are saved in the output variable *result* (line 21). Otherwise, if *node* is not a leaf, we pair its sons to all the items in *new_states*, and save these objects into *searching* (lines 26-34). The visit terminates when *searching* is empty, i.e., all tree levels have been visited.

The visit procedure is linear in the number of the trace tree nodes.

Example Referring to our example query, providing the trace tree in Figure 1 as an input to the Algorithm *Search_Process*, *searching* initially contains the sons of the root A, B, and C, paired with the initial state of the deterministic automaton obtained from the non-deterministic one in Figure 2, and with the empty string. We simulate the actions A, B, and C on the automaton. Only B is recognized, generating a state saved in *new_states* with the corresponding string B (line 15). We then pair the sons of node B (E, D, DE) to the item in *new_states* and save these triples into *searching* (lines 26-34). Continuing the visit, particularly interesting is the case of node DE, which requires to consider all the possible permutations of actions D and E. Both the permutations DE and ED are initially recognized. However, as the visit proceeds and node PZ is reached, it turns out that DE must be followed by the permutation PZ to satisfy the query; on the other hand, if the choice ED is made, it must be followed by ZP. Indeed, the query imposes some *constraints that cannot be checked only locally*, i.e., referring to a single node along the branch. After this step of the visit (depth 4 in the tree), the recognized partial strings paired to node PZ are $BDEOPZ$ and $BEDOZP$. When reaching the leaf node W, however, only $BDEOPZW$ is recognized, and reported in output together with the corresponding leaf node W.

Considering all the branches, the overall output of the *Search_Process* algorithm is: $\langle$ Z in branch 1, $BEOPZ$ $\rangle$, $\langle$ W in branch 4, $BDEOPZW$ $\rangle$, $\langle$ Z in branch 5, $BEDWZ$ $\rangle$.

Filtering

As observed above, the tree search step outputs a set pairs, each composed of a leaf node and a string, the latter corresponding to an explicit instantiation of the input query issued by the user.

If an output leaf node ends a branch which includes one or more nodes with actions to be executed in any order, it is possible that only some of the permutations of these actions are acceptable to answer the corresponding explicit instantiation of the input query; therefore, the support traces must be filtered accordingly.

The filtering algorithm (see Algorithm 3) takes in input the trace tree T, and the output set of the tree search step (in *search_result*), and operates as follows.

For each item in *search_result*, it extracts the node component and saves it in *node* (lines 5-6). If *node* is a leaf (line 9), function *support* is applied

ALGORITHM 2: Pseudo-code of the procedure *Search_Process*.

```
1  Search_Process(T, A)
2  Output: set of < node, string >
3  result ← {}
4  searching ← {}
5  foreach node ∈ sons(root(T)) do
6  |  searching ← searching ∪ < node, 0, empty >
7  end
8  repeat
9  |  tmp ← {}
10 |  foreach < node, state, string > ∈ searching do
11 |  |  if node is an any-order-actions node then
12 |  |  |  new_states ← anyorder_simulate(A, < node, state, string >)
13 |  |  end
14 |  |  else
15 |  |  |  new_states ← simulate(A, < node, state, string >)
16 |  |  end
17 |  |  if new_states ≠ {} then
18 |  |  |  if node is a leaf then
19 |  |  |  |  foreach < state, string > ∈ new_states do
20 |  |  |  |  |  if final(state) then
21 |  |  |  |  |  |  result ← result ∪ < node, string >
22 |  |  |  |  |  end
23 |  |  |  |  end
24 |  |  |  end
25 |  |  |  else
26 |  |  |  |  foreach n ∈ sons(node) do
27 |  |  |  |  |  foreach < state, string > ∈ new_states do
28 |  |  |  |  |  |  tmp ← tmp ∪ < n, state, string >
29 |  |  |  |  |  end
30 |  |  |  |  end
31 |  |  |  end
32 |  |  end
33 |  end
34 |  searching ← tmp
35 until searching ≠ {}
36 return result
```

ALGORITHM 3: Pseudo-code of the procedure *Filtering*.

```
1  Filtering(T, search_result)
2  Output: references to traces
3  result ← {}
4  tmp ← {}
5  foreach < n, string > ∈ search_result do
6      node ← n
7      tmp ← {}
8      foreach element ∈ string do
9          if node is a leaf then
10             tmp ← support(node, element)
11         end
12         else
13             if node is an any-order-actions node then
14                 tmp ← tmp ∩ support(node, element)
15             end
16         end
17         node ← father(node)
18         element ← update(element)
19     end
20     result ← result ∪ tmp
21 end
22 return result
```

to the node itself, and to the tail portion (*element*) of the corresponding string: in particular, if *node* contains a single action, *element* is a just the last symbol in the string; otherwise, it is a sequence of as many symbols as the number of actions to be executed in any order in *node*, always computed from the last position.

The function *support* outputs in *tmp* the traces verifying *element*: they are a subset of the support traces of *node*, if *node* contains actions in any order, since *element* identifies the single acceptable permutation, in this case. Otherwise, all the support traces of *node* are returned (line 10).

The variable *node* is then updated by considering the father of the leaf, along its branch (line 17); *element* is updated accordingly, moving towards the start of the string by as many symbols as the number of actions in *node* (line 18).

At the next iteration, if *node* contains actions to be executed in any order (line 13), the algorithm calculates the intersection between *tmp* and the output of the function *support* (calculated as above), in order to always keep all and only the traces supporting the portion of the string analysed so far (line 14).

The algorithm terminates when the string has been fully examined (i.e., we have reached the sons of the root in T), and outputs the references to the filtered traces in *result* (line 22). The complexity is therefore the number of elements in *search_result* (i.e., the number ⟨leaf node, string ⟩ pairs identified as an output of the previous step), by the length of the longest string in *search_result*.

Example We exemplify the algorithm *Filtering* on the item $\langle W$ in branch 4, $BDEOPZW \rangle$. We set $node = W$ (line 6) and $element{=}W$; node W is a leaf and does not contain actions to be executed in any order. Therefore, function *support* outputs in *tmp* all the support traces referenced by this node. We then update $node{=}PZ$. At the following iteration, $element{=}PZ$, and, since *node* contains two actions to be executed in any order, we calculate the intersection between *tmp* and the support traces corresponding to the permutation PZ (line 14). Variable *node* is then set to O. At the following iteration, $element{=}O$, and *tmp* does not change. *node* is set to DE. At the next iteration, $element{=}DE$, and we calculate the intersection between *tmp* and the support traces corresponding to the permutation DE (line 14). The last iteration is trivial, and does not change the content of *tmp*, which, in the end, contains the references to all and only the traces supporting the string

$BDEOPZW$, added to the output *result* (line 22).

Obviously, if the leaf node ends a branch that contains no nodes with actions to be executed in any order, the leaf support traces can be directly presented to the user, and the filtering step is not required.

3. Results

We have compared our method to a very classical approach, i.e., an existing regular expression processor provided by the Java Regex APIs, not coupled with any indexing strategy[2].

The experimental database was taken from the Datacentrum website[3]. Datacentrum archives research data in a standardized, secure and well documented manner, and provides permanent access to them. Specifically, we used a synthetic dataset referring to a loan assessment process. The dataset is composed of 10000 traces, expressed as sequences of 8 to 27 actions. 90% of the traces are 8 to 12 action long. Overall, the dataset contains 14 different action types.

For the experiments, we used a machine equipped with an Intel i7-4810MQ CPU @ 2.80GhZ, 8GB RAM, SSHD Hybrid 64MB Cache.

In detail, we performed a scalability test. To this end, we made random samplings of the overall available traces, defining 5 subsets of the original database, of dimensions 1000, 3000, 5000, 7000 and 10000 traces respectively (where the last sample is obviously the whole database).

We then defined two different query types, and executed 1000 queries for each type. We then calculated average query answering times.

The first query type was characterized by the presence of a short delays: the sum of the maximum ranges of the delays is at most three. Overall, the query could contain up to three delays, as in Examples (A), (B) and (C) below.

Example (A):
(0,0)
Loan_application_received
Check_application_form_completeness

[2]The interested reader can find information about Regex at the link http://www.regular-expressions.info/engine.html - last accessed on July 16, 2015

[3]http://datacentrum.3tu.nl/en/home/, last accesses on July 16, 2015

(3,5)
Reject_application
Loan_application_rejected
(0,0)

Example (B):
(0,2)
Appraise_property
(1,2)
Assess_eligibility
Reject_application
Loan_application_rejected
(0,0)

Example (C):
(0,1)
Check_application_form_completeness
(2,3)
Assess_loan_risk
Assess_eligibility
(1,2)
Loan_application_rejected
(0,0)

The second query type was characterized by the presence of longer delays, whose overall length could be up to 5 actions, as in Example (D) below.
Example (D):
(0,0)
Loan_application_received
Check_application_form_completeness
(0,5)
Reject_application
Loan_application_rejected
(0,0)

The same queries were executed both using our approach and using the Regex Java processor on the 5 archives of different dimensions. The average retrieval times for the two query types are shown in Tables 1 and 2. In the Tables, referring to our method, the times of the three steps (automaton generation, tree search and filtering) are also detailed.

Table 1: Comparison of retrieval times (in msec) using our method (trace tree - TT) and using the Regex Java regular expression processor in a scalability test on type 1 queries (short delays)

DB dimension	**TT autom.**	**TT search**	**TT filter.**	**TT tot.**	**regex JAVA**
1000	1.71	0.05	0.08	1.84	8.48
3000	1.74	0.06	0.23	2.03	18.82
5000	1.75	0.06	0.40	2.21	29.13
7000	1.73	0.06	0.54	2.33	41.57
10000	1.76	0.06	0.77	2.59	58.72

Table 2: Comparison of retrieval times (in msec) using our method and using the Regex Java regular expression processor in a scalability test on type 2 queries (long delays)

DB dimension	**TT autom.**	**TT search**	**TT filter.**	**TT tot.**	**regex JAVA**
1000	3.57	0.08	0.11	3.76	11.90
3000	3.66	0.08	0.31	4.05	18.80
5000	3.63	0.09	0.54	4.26	31.13
7000	3.67	0.09	0.74	4.50	46.23
10000	3.59	0.09	1.02	4.70	60.13

As it can be observed, our method always outperformed Regex. The time of both methods grew as the database dimension increased, but the growth of our method was very limited, referring to both query types (as shown in Figure 3, which plots the values of the two Tables).

In both methods, type 2 queries took more time than type 1 queries. This was expected. In particular, as regards our method, the non-deterministic automaton corresponding to type 2 queries has a larger number of states than the one corresponding to type 1 queries (built using Berry and Sethi approach (Berry & Sethi, 1986)). Thus, the transformation cost from the non-deterministic automaton to the deterministic automaton increases too, since this cost depends on the number of states of the non-deterministic automaton.

Looking at our method in more detail, referring to both query types, as expected, the cost of building the automaton for queries with a given length of delays was almost constant. The same also holds for searching time, since tree search navigates the index, whose dimension increased only slightly with respect to the size of the dataset. On the other hand, not surprisingly, filtering time (which depends on the number of the retrieved

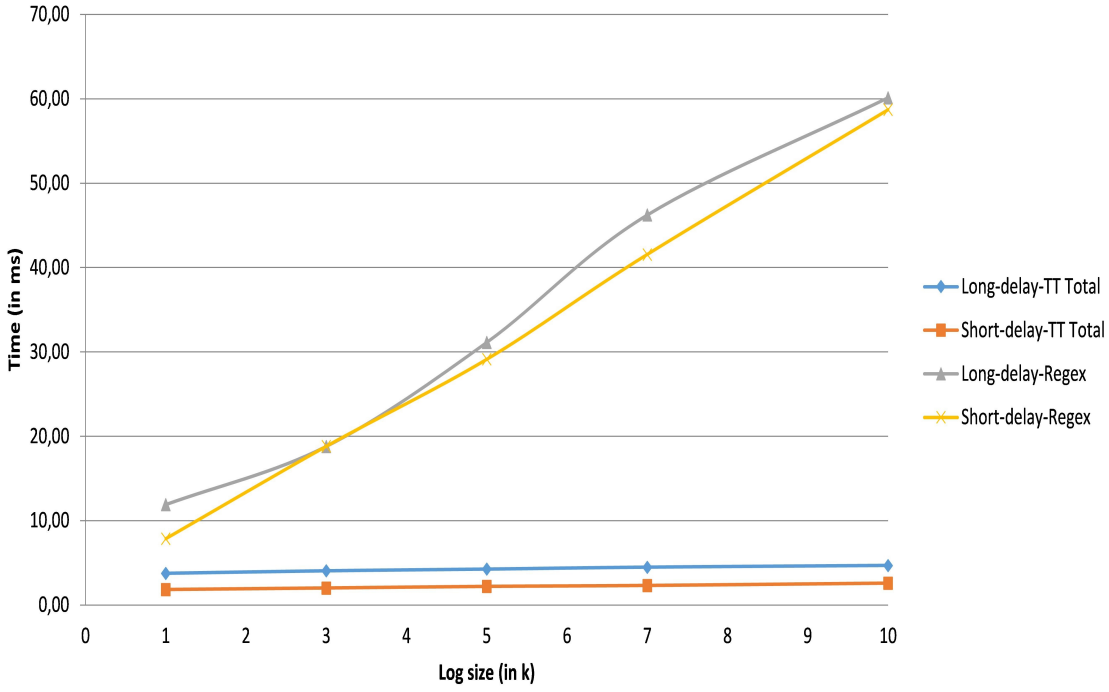

Figure 3: Scalability results.

traces) grew linearly with respect to the size of the dataset. However, the whole computation was dominated by the cost of building the automaton. Specifically, automaton generation took from 70 to 95% of the total query answering time (see Tables 1 and 2). On the other hand, the Regex approach does not exploit any index, so that each trace in the dataset is directly checked. As a consequence, the time complexity of Regex grew linearly with respect to the dimension of the dataset.

In conclusion, the approach described in this paper can answer the queries in significantly shorter times with respect to classical approaches, and the advantage increases as the database grows.

4. Related work

Operational support is typically not provided by commercial Business Intelligence and Business Process Management tools. However, operational support techniques are implemented in the open source framework ProM

(van Dongen et al., 2005), developed at the Eindhoven University of Technology, which represents the state of the art in process mining research. In ProM, prediction and recommendation are typically supported by replaying log traces on the transition system (van Dongen, 2007), a state-based model that explicitly shows the states a process can be in, and all possible transitions between these states. The replay activity allows to calculate, e.g., the mean time to completion from a given state, or the most probable next action to be executed. In ProM's approach, statistics on event log traces are thus used for operational support, but the overall technique is very different from the one we propose in this paper, and no trace retrieval on the basis of complex patterns search is supported.

On the other hand, traces have been recently considered in the CBR literature, as sources for retrieving and reusing user's experience. As an example, at the International Conference on CBR in 2012, a specific workshop was devoted to this topic (see Floyd et al. (2012)).

In 2013, Cordier et al. (2013) proposed trace-based reasoning, a CBR approach where cases are not explicitly stored in a library, but are implicitly recorded as "episodes" within traces. The elaboration step, in which a case is extracted from a trace, is thus one of the most challenging parts of the reasoning process. Zarka et al. (2013) extended that work, and defined a similarity measure to compare episodes extracted from traces. In these works, traces are typically intended as observations captured from users' interaction with a computer system. Trace-based reasoning was exploited in recommender systems (Adomavicius et al., 2011; Zarka et al., 2012), and to support the annotation of digitalized cultural heritage documents in Doumat et al. (2010).

Huang et al. (2013) and Montani & Leonardi (2014) propose two trace retrieval approaches, where different metrics, based on extensions of the edit distance, are exploited. No indexing strategy is however provided to make retrieval faster.

Leake (2010) used execution traces recording provenance information to improve reasoning and explanation in CBR. In the Phala system (Leake & Kendall-Morwick, 2008), the authors supported the generation and composition of scientific workflows by mining execution traces for recommendations to aid workflow authors. Finally, Lanz et al. (2010) used annotated traces recorded when a human user played video games in order to feed a case-based planner. All these approaches implement forms of reasoning on traces, but do not aim at providing operational support.

There is a growing interest on business process management applications in the CBR community (see, e.g., (Minor et al., 2014b)). Most works, however, focus on process models rather than on traces, and aim at retrieving models (typically represented as graphs) similar to an input one. These works have required the introduction of different metrics for graph comparison. Most of them are based on extensions of the graph edit distance (Minor et al., 2008; Bergmann & Gil, 2014; Kunze & Weske, 2011; Li et al., 2008; Montani et al., 2015; Dijkman et al., 2009; LaRosa et al., 2013).

The work in Minor et al. (2008) makes use of a normalized version of the graph edit distance. The approach is used to support workflow modification in an agile workflow system, and takes into account control flow information as well as activity information. However, the work is limited to considering (small) changes with respect to a running process instance. The work in Dijkman et al. (2009) also provides a normalized version of the graph edit distance for comparing business process models, and defines syntactical edit operation costs for activity node substitution, activity node insertion/deletion, and edge insertion/deletion. The work in LaRosa et al. (2013) extends the work in Dijkman et al. (2009) by explicitly representing gateway nodes, in order to describe, e.g., parallelism and mutual exclusion, and exploiting them in distance calculation. The work in Kunze & Weske (2011) relies on graph edit distance, and exploits string edit distance on node names to determine the cost of node substitutions. The work in Li et al. (2008) encapsulates a set of edit operations into the so-called "high-level change operations", and measures distance on the basis of the number of high-level change operations needed to transform one graph into another.

All the above contributions typically make use of syntactical information in the definition of the edit operation costs. The works in Bergmann & Gil (2014); Montani et al. (2015), on the other hand, exploit semantic information in activity comparison. In Bergmann & Gil (2014), a system working on workflows represented as semantically labeled graphs is presented. The paper proposes to use a metric in which the similarity between two mapped nodes or arcs makes explicit use of their semantic description. The work is particularly focused on the data flow. The work in Montani et al. (2015) exploits domain knowledge and temporal information to parametrize the cost of node substitutions and edge substitution. As in LaRosa et al. (2013), gateway nodes are also considered.

The approach in Goderis et al. (2006), on the other hand, affords the problem of graph comparison by relying on graph isomorphism. It focuses

on scientific workflows, which have a strong focus on the data flow, typically restricting the control flow to a partial ordering of the tasks. The work in Ma et al. (2014) focuses on data oriented workflows as well. It defines a formal structure called Time Dependency Graph (TDG), and exploits it as a representation model of data oriented workflows with variable time constraints. A distance measure is proposed for computing workflow similarity by their normalization matrices, established on the basis on their TDGs.

The work in Kapetanakis et al. (2010) exploits a maximum common subgraph approach for similarity-based process retrieval, in a retrieval system for supporting business process monitoring. Interestingly, the metric in Kapetanakis et al. (2010) takes into account temporal information, since it combines a contribution related to activity similarity, and a contribution related to delays between activities.

Finally, in Madhusudan et al. (2004) a retrieval system for supporting incremental workflow modeling is presented. The system proposes a similarity-based retrieval of workflow templates using a planner that employs an inexact graph matching algorithm based on similarity flooding. For computing similarities, the algorithm relies on the idea that elements of two distinct graphs are similar, when their adjacent elements are similar. The algorithm propagates the similarity from a node to its respective neighbors based on the topology in the two graphs.

It is worth noting that all of these works, including our own previous contribution (Montani et al., 2015), are however only loosely related to the present paper, since they do not focus on traces, but on process models (i.e., graphs), and do not aim at enabling operational support.

Very interestingly, some recent papers in the area of CBR for business process management (Minor et al., 2011, 2014a; Müller & Bergmann, 2014) also consider the reuse/adaptation step of the CBR cycle (Aamodt & Plaza, 1994). An automation of the adaptation step in process model retrieval is easier in very specific domains, such as the one of cooking recipes (Müller & Bergmann, 2014). In this setting, the graph structure is usually quite simple, and adaptation is often limited to ingredient substitution. However, the issue of implementing a reuse/adaptation strategy in trace retrieval may be investigated in our future research as well.

5. Conclusions

In this paper, we have introduced a novel framework for trace retrieval, studied to implement operational support tasks in a flexible and efficient way.

With respect to existing operational support facilities, our tool is significantly innovative, since, to the best of our knowledge, it is the first one to provide a direct exploitation of experiential knowledge, by means of case-based retrieval techniques. Once past traces have been retrieved, classical statistical techniques can then be applied to them, to support prediction and recommendation.

With respect to existing trace retrieval tools in the CBR area, our approach is more efficient and flexible, since:

- by allowing for the use of (imprecise) delays in the query language, it enables to express a very large number of explicit queries in a compact way;
- by providing the trace tree as an input to the automaton:
  - it executes common prefixes of different traces only once on the automaton, avoiding repeated, identical checks;
  - it speeds up retrieval with respect to a flat search into the event log (as testified by our experiments).

In the future, we would like to enable also the retrieval of traces that include a prefix similar (but not necessarily identical) to the input trace. To this end, we plan to rely on classical techniques, involving a visit of the trace tree, and a comparison between the trace tree branch at hand and the input trace by means of edit distance calculation. This approach, while quite straightforward from the methodological viewpoint, will further increase the flexibility and the usability of our tool.

Finally, we plan to extensively test the overall framework on real world traces, which log the actions executed during stroke patient management in a set of Northern Italy hospitals.

der Aalst, W. V. (2011). *Process Mining. Discovery, Conformance and Enhancement of Business Processes*. Springer.

Aamodt, A., & Plaza, E. (1994). Case-based reasoning: foundational issues, methodological variations and systems approaches. *AI Communications*, *7*, 39–59.

Adomavicius, G., Mobasher, B., Ricci, F., & Tuzhilin, A. (2011). Context-aware recommender systems. *AI Magazine*, *32*, 67–80.

Baeza-Yates, R. A., & Gonnet, G. H. (1996). Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, *43*, 915–936.

Bergmann, R., & Gil, Y. (2014). Similarity assessment and efficient retrieval of semantic workflows. *Information Systems*, *40*, 115–127.

Berry, G., & Sethi, R. (1986). From regular expressions to deterministic automata. *Theor. Comput. Sci.*, *48*, 117–126.

Buijs, J., van Dongen, B., & van der Aalst, W. (2012). On the role of fitness, precision, generalization and simplicity in process discovery. In *On the Move to Meaningful Internet Systems: OTM 2012* (pp. 305–322). Springer.

Canensi, L., Montani, S., Leonardi, G., & Terenziani, P. (2014). Chapman: a context aware process miner. In *Proc. Workshop on Synergies between Case-Based Reasoning and Data Mining, International Conference on Case Based Reasoning (ICCBR)*.

Cnudde, S. D., Claes, J., & Poels, G. (2014). Improving the quality of the heuristics miner in prom 6.2. *Expert Syst. Appl.*, *41*, 7678–7690.

Cordier, A., Lefevre, M., Champin, P., Georgeon, O., & Mille, A. (2013). Trace-based reasoning - modeling interaction traces for reasoning on experiences. In C. Boonthum-Denecke, & G. M. Youngblood (Eds.), *Proceedings of the Twenty-Sixth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2013, St. Pete Beach, Florida. May 22-24, 2013.*. AAAI Press.

Dijkman, R., Dumas, M., & Garca-Banuelos, R. (2009). Graph matching algorithms for business process model similarity search. In U. Dayal, J. Eder, J. Koehler, & H. Reijers (Eds.), *Proc. International Conference on Business Process Management* (pp. 48–63). volume 5701 of *Lecture Notes in Computer Science*.

van Dongen, B. (2007). *An Iterative Algorithm for Applying the Theory of Regions in Process Mining*. BETA publicaties: Preprints. Beta, Research School for Operations Management and Logistics.

van Dongen, B., De Medeiros, A. A., Verbeek, H., Weijters, A., & der Aalst, W. V. (2005). The proM framework: a new era in process mining tool support. In G. Ciardo, & P. Darondeau (Eds.), *Knowledge Mangement and its Integrative Elements* (pp. 444–454). Springer, Berlin.

Doumat, R., Egyed-Zsigmond, E., & Pinon, J. (2010). User trace-based recommendation system for a digital archive. In I. Bichindaritz, & S. Montani (Eds.), *Case-Based Reasoning. Research and Development, 18th International Conference on Case-Based Reasoning, ICCBR 2010, Alessandria, Italy, July 19-22, 2010. Proceedings* (pp. 360–374). Springer volume 6176 of *Lecture Notes in Computer Science*.

Floyd, M. W., Fuchs, B., Gonzalez-Calero, P., Leake, D., Ontanon, S., Plaza, E., & Rubin, J. (2012). *TRUE: Traces for Reusing Users Experiences Cases, Episodes, and Stories, International Conference on Case Based Reasoning (ICCBR)*. Lyon.

Goderis, A., Li, P., & Goble, C. A. (2006). Workflow discovery: the problem, a case study from e-science and a graph-based solution. In F. Leymann, & L. Zhang (Eds.), *Proc. IEEE International Conference on Web Services* (pp. 312–319). IEEE, USA.

Huang, Z., Juarez, J. M., Duan, H., & Li, H. (2013). Length of stay prediction for clinical treatment process using temporal similarity. *Expert Syst. Appl.*, *40*, 6330–6339.

Kapetanakis, S., Petridis, M., Knight, B., Ma, J., & Bacon, L. (2010). A case based reasoning approach for the monitoring of business workflows. In I. Bichindaritz, & S. Montani (Eds.), *Proc. International Conference on Case Based Reasoning (ICCBR)* (pp. 390–405). Springer, Berlin volume 6176 of *Lecture Notes in Computer Science*.

Kunze, M., & Weske, M. (2011). Metric trees for efficient similarity search in large process model repositories. In M. Muehlen, & J. Su (Eds.), *Proc. Business Process Management Workshops* (pp. 535–546). Springer, Berlin volume 66 of *Lecture Notes in Business Information Processing*.

Lam, M. S., Aho, A. V., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Lanz, A., Weber, B., & Reichert, M. (2010). Workflow time patterns for process-aware information systems. In *Proc. BMMDS/EMMSAD* (pp. 94–107).

LaRosa, M., Dumas, M., Uba, R., & Dijkman, R. (2013). Business process model merging: An approach to business process consolidation. *ACM Trans. Softw. Eng. Methodol.*, *22*, 11.

Leake, D. B. (2010). Case-based reasoning tomorrow: Provenance, the web, and cases in the future of intelligent information processing. In Z. Shi, S. Vadera, A. Aamodt, & D. B. Leake (Eds.), *Intelligent Information Processing V - 6th IFIP TC 12 International Conference, IIP 2010, Manchester, UK, October 13-16, 2010. Proceedings* (p. 1). Springer volume 340 of *IFIP Advances in Information and Communication Technology*.

Leake, D. B., & Kendall-Morwick, J. (2008). Towards case-based support for e-science workflow generation by mining provenance. In K. Althoff, R. Bergmann, M. Minor, & A. Hanft (Eds.), *Proc. ECCBR 2008, Advances in Case-Based Reasoning* (pp. 269–283). Springer volume 5239 of *Lecture Notes in Computer Science*.

van Leeuwen, J. (1994). *Handbook of Theoretical Computer Science*. Mit Press.

Li, C., Reichert, M., & Wombacher, A. (2008). On measuring process model similarity based on high-level change operations. In Q. Li, S. Spaccapietra, E. S. K. Yu, & A. Olivé (Eds.), *Proc. International Conference on Conceptual Modeling* (pp. 248–264). Springer, Berlin volume 5231 of *Lecture Notes in Computer Science*.

Ma, Y., Zhang, X., & Lu, K. (2014). A graph distance based metric for data oriented workflow retrieval with variable time constraints. *Expert Syst. Appl.*, *41*, 1377–1388.

Madhusudan, T., Zhao, J., & Marshall, B. (2004). A case-based reasoning framework for workflow model management. *Data and Knowledge Engineering*, *50*, 87–115.

Minor, M., Bergmann, R., & Görg, S. (2014a). Case-based adaptation of workflows. *Inf. Syst.*, *40*, 142–152.

Minor, M., Bergmann, R., Görg, S., & Walter, K. (2011). Reasoning on business processes to support change reuse. In B. Hofreiter, E. Dubois, K. Lin, T. Setzer, C. Godart, E. Proper, & L. Bodenstaff (Eds.), *13th IEEE Conference on Commerce and Enterprise Computing, CEC 2011, Luxembourg-Kirchberg, Luxembourg, September 5-7, 2011* (pp. 18–25). IEEE Computer Society.

Minor, M., Montani, S., & Recio-García, J. (2014b). Process-oriented case-based reasoning. *Inf. Syst.*, *40*, 103–105.

Minor, M., Tartakovski, A., Schmalen, D., & Bergmann, R. (2008). Agile workflow technology and case-based change reuse for long-term processes. *International Journal of Intelligent Information Technologies*, *4*, 80–98.

Montani, S., & Leonardi, G. (2014). Retrieval and clustering for supporting business process adjustment and analysis. *Information Systems*, *40*, 128–141.

Montani, S., Leonardi, G., Quaglini, S., Cavallini, A., & Micieli, G. (2015). A knowledge-intensive approach to process similarity calculation. *Expert Syst. Appl.*, *42*, 4207–4215.

Müller, G., & Bergmann, R. (2014). Workflow streams: A means for compositional adaptation in process-oriented CBR. In L. Lamontagne, & E. Plaza (Eds.), *Case-Based Reasoning Research and Development - 22nd International Conference, ICCBR 2014, Cork, Ireland, September 29, 2014 - October 1, 2014. Proceedings* (pp. 315–329). Springer volume 8765 of *Lecture Notes in Computer Science*.

Zarka, R., Cordier, A., Egyed-Zsigmond, E., Lamontagne, L., & Mille, A. (2013). Similarity measures to compare episodes in modeled traces. In S. J. Delany, & S. Ontañón (Eds.), *Case-Based Reasoning Research and Development - 21st International Conference, ICCBR 2013, Saratoga Springs, NY, USA, July 8-11, 2013. Proceedings* (pp. 358–372). Springer volume 7969 of *Lecture Notes in Computer Science*.

Zarka, R., Cordier, A., Egyed-Zsigmond, E., & Mille, A. (2012). Contextual trace-based video recommendations. In A. Mille, F. L. Gandon, J. Misselis,

M. Rabinovich, & S. Staab (Eds.), *Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume)* (pp. 751–754). ACM.