Mining the *log-tree* of process traces: current approach and future perspectives

L.Canensi
Department of Computer Science
Università di Torino, Italy

G.Leonardi,
S.Montani
and P.Terenziani
DISIT, Computer Science Institute,
Università del Piemonte Orientale,
Alessandria, Italy

***Abstract*—Logs recording the traces of execution of previous process instances can be exploited for different process management tasks, such as prediction and recommendation in operational support. Efficient retrieval of past traces can be very important to achieve such tasks, while building a model of the process from the log can support problem/anomaly detection and, more generally, process analysis. Trace retrieval is gaining attention in the Case Based Reasoning research community, but so far it has been faced in a completely separate way from the construction of a process model from the log; instead, we propose an approach aiming at integrating these two goals. The core notion of our proposal is the *log-tree*, which constitutes a "bridge" between the notions of (log) index and process model. We propose a mining algorithm to build it, and an algorithm exploiting it for trace retrieval. Future extensions of our initial contribution are also widely discussed.**

I. Introduction

Nowadays, many organizations keep electronic track of the processes they run. The recordings of completed process instances are stored in the so-called *event log*, which maintains the sequences (*traces* henceforth) of actions executed at the given organization, typically together with action execution parameters (e.g., times, costs, resources). Event logs can be exploited for several different tasks. For instance, in *operational support* (see [1], chapter 9), the log is exploited to assist users executing a new process instance, e.g., to make *predictions* about the instance completion (such as time to completion, costs, use of resources, possible problems), or to *recommend* suitable next actions, resources or routing decisions (in order, e.g., to minimize costs or time to completion). Predictions and recommendations heavily rely on the experiential knowledge stored in the event log; we thus believe that the ability to *retrieve* from the log those traces that are similar/equal to the running process instance would be greatly beneficial in these tasks. Indeed, despite the fact that traditional operational support tools like the one in ProM (see [2]) typically operate differently, i.e., by replaying log traces on the transition system [3], [4], the problem of trace retrieval is recently being considered in the area of Case Based Reasoning [5], where process management is attracting increasing attention [6].

Notably, operational support often also provides problem/anomaly *detection* facilities, that, together with more general *process analysis* tasks, require the abstraction of a *process model* from the log (provided, e.g., through process mining techniques [1]).

While, in general, trace retrieval (se, e.g., [7], [8], [9]) has been faced in a completely separate way from the construction of a process model from the log, in this paper, we propose an approach aiming at integrating these two tasks.

The basic idea of our approach is quite simple: mining-like techniques can be used in order to identify what we termed the *log-tree*, which is, informally speaking, "something in the middle" bridging an *index* of the traces and a *model* of the process. Differently from a "pure" index, the log-tree gives users an easy-to-interpret representation of the traces it refers to. Differently from process models [1], the log-tree explicitly indexes in its nodes all the traces in the log corresponding to (the path from the root of the tree to) such a node. In this way, the log-tree can be used both as a tool to achieve efficient trace retrieval (see section III), and as a high-level representation to analyze the processes in the organization.

From a slightly more technical point of view, it is important to stress that, since each node in the log-tree indexes some (at least one) traces in the log, our mining algorithm (discussed in section II) has to grant that a 100% *precision* (i.e., $precision = 1$ [10]) is reached (which means that each path in the model must correspond to at least one trace in the log). Achieving this precision has several different implications on the model we provide. From one side, the log-tree perfectly adheres to the input traces: no path in the tree has no correspondent in the log (other properties are discussed in section II). Such a property may be highly desirable in several application domains. For instance, in the medical context, 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, since it is vital that mining results are reliable as much as possible, in order to facilitate the work of physicians and hospital managers in guaranteeing the highest quality of care to each patient. However, achieving a 100% precision somehow limits the abstraction capabilities of the mining algorithm. Abstraction causing some loss of precision may be helpful, especially in applications where it is known that the input traces do not cover all possible cases. In section IV-A we briefly explore such a challenging issue, considering two specific examples to compare the log-tree and the model provided by Heuristic Miner [11] (as a representative of process mining approaches). Then, in section IV-B we provide a hint of how we aim at

extending our approach to increase its abstraction capabilities (at the prize of losing the 100% precision, and the indexing facility). Finally, in section IV-C we identify a different line of future research, which integrates our approach with standard process mining techniques. The idea we propose is simple: given its 100% precision, our log-tree can be used as a basis to provide domain experts with a tool to identify and analyze (in a quantitative and/or in a qualitative way) the (non 100% precision) abstractions provided by the other process miners.

II. Mining the log-tree

In this section, we first introduce our representation formalism; we then present the algorithm to mine the log-tree, and finally we discuss the properties of the obtained data structure.

A. Representation formalism and semantics

Our mining algorithm takes in input the event log, and outputs the *log-tree*, a data structure whose nodes represent actions, and arcs represent a precedence relation between them. Indeed, we exploit the temporal ordering of actions in the traces to mine the log-tree. When more than one edge exit a node, this node univocally represents a XOR splitting point, while actions in AND (or, more precisely, to be executed in any order) are collapsed into the same node.

Specifically, in the log-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/any-order relation. Note that, in such a way, each path from the root 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 log-tree, and ordering in every 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 is 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 by several sets of support traces, each one corresponding to a possible action permutation. T is typically a subset of the event log. Specifically, every time a XOR splitting point is encountered, the support traces are properly divided among the various alternative paths.

An example log-tree is shown in figure 1.a. Numbers in brackets in the nodes in figure 1.a provide the cardinalities of support traces. Every edge connecting a node A to another node B also stores the *edge frequency* of the sequence $A > B$ (A immediately preceding B), defined in the next subsection.

Details of the algorithm are presented in the next subsection as well.

B. Data structure and algorithm

The input 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 #.

Algorithm 1 builds the log-tree.

ALGORITHM 1: Mining pseudocode

```
1 Build-Tree (index,< P,T >) ;
2 nextP ← getNext(index+1, T, α) ;
3 if nextP not empty then
4 |   nextActions ← XORvsAND (nextP, T, β) ;
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* inspects the traces in *T* to find all possible next actions. At this stage, "rare" patterns will be ruled out. Specifically, if $P = \{A\}$, and B is a possible next action, B will be provided in output by *getNext* only if the *edge frequency* E_F of the sequence $A > B$ is above a user-defined threshold α, where

$$E_F(A > B) = \frac{|A > B|}{|T_A|} \tag{1}$$

being $|A > B|$ the number of traces in T in which A is immediately followed by B (i.e., the cardinality of the support traces of B), and being $|T_A|$ the cardinality of the support traces of A in T. On the remaining next actions, the function *XORvsAND* applies the formula below in order to identify which actions are in AND/any-order and which are in XOR relation: we calculate the *dependency frequency* $A \rightarrow B$ between every action pair $< A, B >$ in $nextP \times nextP$:

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

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 (user-defined) threshold β, 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 viceversa;
- 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 $\langle P', T' \rangle$, one for each maximal set of actions to be AND-ed. Note that, for each one of such sets P', the corresponding set T' of *support traces* is also computed. T' is more precisely composed by several sets of support traces, each one corresponding to a possible action permutation in P'.

Finally, each new node is appended to the log-tree (function *AppendSon*), and *Build-Tree* is recursively applied to each node (with the parameter *index* properly set).

C. Properties

By construction, the properties below hold for the log-tree.

Property 1: **Trace Indexing.** Each path (and sub-path) in the log-tree indexes all and only its support traces in the log.

As a consequence of Property 1, Property 2 also holds.

Property 2: **Precision.** For each path in the log-tree there is at least one support trace in the log. Thus, the precision [10] of the log-tree is 1.

Each path in the log-tree represents a specific (set of) temporally-oriented action paths (the support patterns of the path). Ordering in the path represents strict ordering of actions in the traces, while actions in the same node of a path correspond to the fact that, in the traces, such actions are not temporally constrained (i.e., they appear in any order in the corresponding path). Thus, Property 3 also holds.

Property 3: **Temporal Accuracy.** The log-tree is temporally accurate, in that it models the temporal ordering of actions in the input traces. Specifically, ordering between nodes in the path represents temporal ordering of actions in the traces, while actions in the same node frequently (up to a user-defined threshold) occur in any order (at that point of the path) in the log.

Finally, property 4 also holds.

Property 4: **Context awareness.** Since each node is a pair $< P, T >$ where T maintains the support traces, the log-tree is context aware; indeed, the support traces of each alternative path implicitly define the execution context of the corresponding model branch.

III. Trace retrieval

In order to retrieve log traces that correspond to a specific process execution, we have implemented a procedure (see algorithm 2) that takes in input a node N of the log-tree (initially the root), a sequence of actions S to be searched for in the tree (i.e., the query), and a set of support traces T (initially the entire log) which, in the end, will contain the outcome of retrieval.

The algorithm calculates the set of traces, which exactly match the sequence of actions S, or contain it as an exact prefix. These traces are (a subset of) the support traces in a node of the log-tree.

ALGORITHM 2: Pseudo-code of the procedure $Retrieval\_Process$.

```
1  Retrieval_Process(N, S, T)
2  result ← {}
3  foreach son ∈ son(N) do
4      cond ← null
5      foreach perm ∈ get_permutations(son) do
6          if perm = first(|son|, S) then
7              cond ← son
8          end
9      end
10     if cond ≠ null then
11         if tail(|cond|,S) is empty then
12             result = result ∪ (T ∩ get_support(cond,perm))
13         end
14         else
15             result = result ∪ Retrieval_Process(cond,tail(|cond|, S),
               T ∩ get_support(cond,perm) )
16         end
17     end
18 end
19 return result
```

Basically, $Retrieval\_Process$ executes a depth-first search in the log-tree. First the algorithm chooses the proper son of N according to the sequence S; in detail, for each son of N (line 3), if one of the permutations of the set of actions in son is equal to the head portion of S (line 5-8), son is assigned to $cond$. The number of actions to be compared to the head portion of S in function $first$ (line 6) is provided by the cardinality $|son|$ of the number of actions to be executed in any order in *son* (possibly 1).

If $cond$ is not null (line 10) and the complete sequence S has been analyzed (line 11), the algorithm calculates the intersection between T and the support traces of node $cond$ that verify the permutation $perm$ (function $get\_support$, line 12) and adds it to $result$ (line 12).

Otherwise (line 14), the function $Retrieval\_Process$ is applied recursively to $cond$, to the tail portion of the sequence S, and to the intersection between T and the support traces of node $cond$ that verify the permutation $perm$ (line 15). Note that if $cond$ contains a single action, the function $tail$ (in lines 11 and 15) just returns the tail portion of S obtained by deleting the first element; otherwise, $tail$ returns the sequence obtained deleting from the head of S as many symbols as the number of actions to be executed in any order in the node $cond$. The output of the recursive call is then added to $result$ (line 15). Finally, $result$ is returned (line 19).

A. Experiments

The speed up in trace retrieval provided by the use of the log-tree as an indexation facility has been tested through some experiments. Experiments were conducted on an 8 core i7-4810MQ processor running at 2.8 GHz, equipped with 8 Gb of RAM. Tests were run on a real world event log, containing 441 stroke patient management traces. The number of actions in each trace ranged from 7 to 22 (15 as an average). The total number of actions in the event log was 6686.

We executed 1000 random queries of length 5 (corresponding to a 5-action trace prefix), 1000 random queries of

length 10, and 1000 random queries of length 15. The average retrieval times for the different types of queries, resorting to our approach, are provided in the first row in table 1. Retrieval times resorting to a flat search are reported in the second row of the table, for comparison.

As it can be observed, our approach was able to reduce retrieval time in all the experiments, especially when working on short queries (where it was 6.2 times faster than the flat search). Indeed, a short query typically does not require to visit the tree down to the leaves, and works on portions of the data structure very close to the root, where the support trace sets of the nodes are larger. The computational advantage when working on longer queries was more contained (about 5 times faster), but still relevant.

Even more significant advantages are foreseen in larger event logs, and will be tested in the future.

IV. Bridging trace retrieval and process mining: analysis and future work

In this section, we first illustrate the differences between our approach and "standard" process mining approaches from the point of view of their abstraction capabilities; two examples are shown, illustrating that a 100% precision is sometimes required, while a higher abstraction level in the output model is desirable in other cases. Then, we provide a hint of how we aim at extending our approach to increase its abstraction capabilities (at the prize of losing some precision, and the indexing support). Finally, we identify a different line of future research, where our log-tree can be used as a basis to provide domain experts with a tool to identify and analyze the (non 100% precision) abstractions provided by other process miners.

A. Abstraction in log-trees and process models

In this section, we briefly compare our approach to "standard" process mining approaches. For the sake of clarity and brevity, we will just refer to Heuristic Miner (HM) [11], that can be considered as a significant representative as well as a milestone in the area of process mining algorithms. Notably, the goal here is just to emphasize what are the main differences in terms of the kind of abstraction the approaches aim at achieving. It is worth mentioning that HM, as well as several other process miners, (and differently from the algorithm we have proposed in section II) assumes the *uniqueness of actions* (i.e., the fact that each action can appear only once in the model).

In the following we will consider two simple examples, which clearly highlight the consequences of this assumption, and the resulting abstraction level. Each example consists of a log, represented as set of pairs ⟨pattern, trace number⟩. For instance, ⟨#CAE, 10⟩ represents the fact that the log contains ten traces "#CAE").

Ex. 1: ⟨#CAE, 10⟩, ⟨#BAF, 10⟩, ⟨#CAF, 10⟩

Figure 1 shows the process model obtained by HM (in the form of a Petri Net; figure 1.b) and the log-tree obtained by our approach (figure 1.a) in Ex. 1. Since the log-tree is also an index, so that a 100% precision is required, our approach cannot generate any abstraction that is not supported by any input trace. On the other hand, HM proposes a *more abstract* model. Roughly speaking, HM abstracts that the process starts with the execution of one action chosen between C and B (in mutual exclusion), then A must be performed, followed by either E or F. Thus, also the pattern BAE is captured by this model, despite the fact that it never appears in the input log.

Ex. 2: ⟨#AD, 1000⟩, ⟨#DB, 1000⟩

Because of the uniqueness assumption, HM is unable, in Ex. 2, to distinguish between D in the first set of traces (i.e., D in the context of being preceded by #A) and D in the second set of traces (i.e., D in the context of being preceded by # and followed by B). Thus, the process model in Figure 2.d is given as output. Notably, besides the patterns #AD and #DB, the output Petri Net also models the patterns #D and #ADB, which do not correspond to any trace in the input log. Such patterns are not mined by our approach (see Figure 2.c), since it achieves a 100% precision.

There cannot be a neat conclusion concerning the above comparisons. In general, there is no formal/automatic way of deciding whether an abstraction not supported by the input traces is desirable or not. For example, Tobin and Vogel [12] propose a system that insures precision, enabling users to add precise rules for abstraction. However, only domain experts can judge (supposing that they indeed have enough domain knowledge to do so).

However, there is an important fact that we want to remark: our approach must *not* be seen *in contrast* to the standard mining algorithms (like HM). Indeed, it has different goals, and may be *conciliated* with standard algorithms to achieve further objectives. This is the goal we want to achieve in our future work, following two innovative lines of research, described in the next two sections.

B. Domain-expert-driven abstraction (with loss of precision)

The first line of research we aim at pursuing consists in extending our current approach with an additional optional facility, to provide further abstractions. This would necessarily lead to losing the 100% precision property, as well as the indexing capability, in the resulting output model. However, of course, the log-tree can be still maintained as the output of an intermediate step of the overall mining process, and as an independent indexing structure.

In detail, we aim at investigating two different ways to achieve further (not 100% precision) abstractions.

The first option will generate a more abstract model by applying to the log-tree a *merging* process, which will merge identical nodes in the log-tree in order to enforce the uniqueness assumption. As observed in section IV-A, this direction is quite "in line" with many current process miners. However, there is a main difference. While in classical process mining the uniqueness assumption is used from the very beginning (i.e., while analyzing the input log), here it is applied as a second step. Notably the second step does not need to consider the input log any more. It directly (and more efficiently) applies to the output of the first step (i.e., to the log-tree, which might be orders of magnitude smaller than the log).

The second option aims at deeply involving domain experts in the achievement of (not 100% precision) abstractions. There are two premises that make this move feasible:

TABLE I: Experimental results. Time is in msec.

	Length-5 query	Length-10 query	Length-15 query
Log-tree	0.17	0.26	0.27
Flat	1.06	1.37	1.39

- the log-tree is already a "high-level" representation of the log data. While it is not reasonable to assume that domain experts directly operate on the log (whose dimension is generally too large to allow human processing), it is reasonable to think that they can look at the log-tree, with the goal of analyzing it, and of proposing new abstractions, visualizing and evaluating the resulting model in an interactive way;
- given the reduced dimension of the log-tree (with respect to the dimension of the log), computing new abstractions from it would be in general a fast process, suitable for an interactive session of work.

We thus aim at defining a suite of facilities to support domain experts in the analysis of the log-tree, in the choice of which parts of it should be further merged, and in the inspection of the resulting model, in an interactive session of work based on the "what-if" paradigm.

As an example, this approach will permit the identification of loops, which (in the log-tree) are currently completely unfolded in their iterations.

Notably, this will be an innovative approach in the area of process mining, where many algorithms behave more like "black-boxes", out of domain experts control (possibly with the exception of system parameters tuning, which, however, is more a system manager task). This approach is likely to provide interesting results in those areas where domain knowledge is strong and consolidated enough to gain significant benefits from a log-tree-based "what if" analysis (like, e.g., many medical ones).

C. Using our approach for analyzing loss-of-precision abstractions

As widely discussed above, the log-tree only contains 100% precision abstractions. In some sense, it thus represents

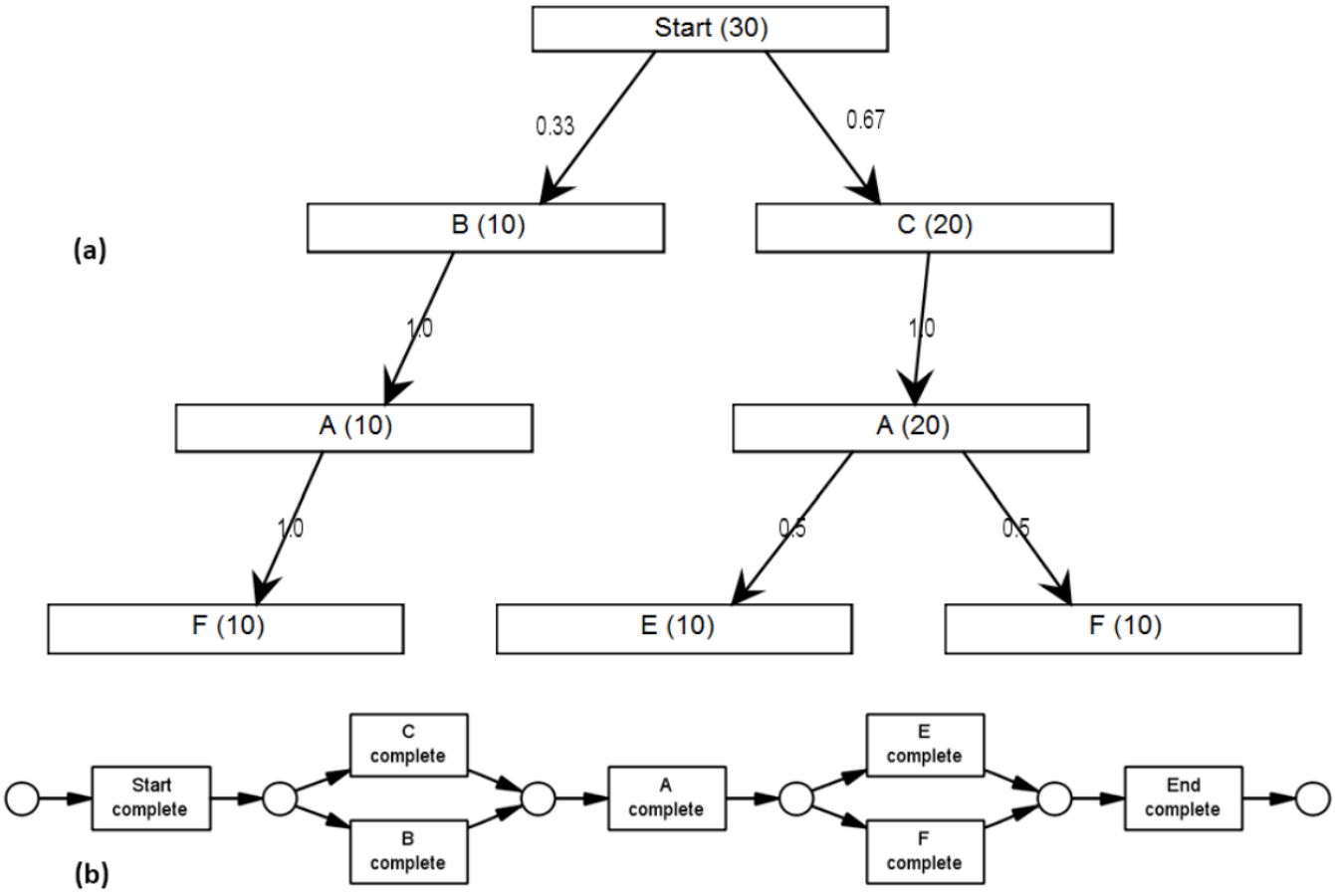

Fig. 1: Example 1: Process models obtained using our approach (a) and using Heuristic Miner (b).

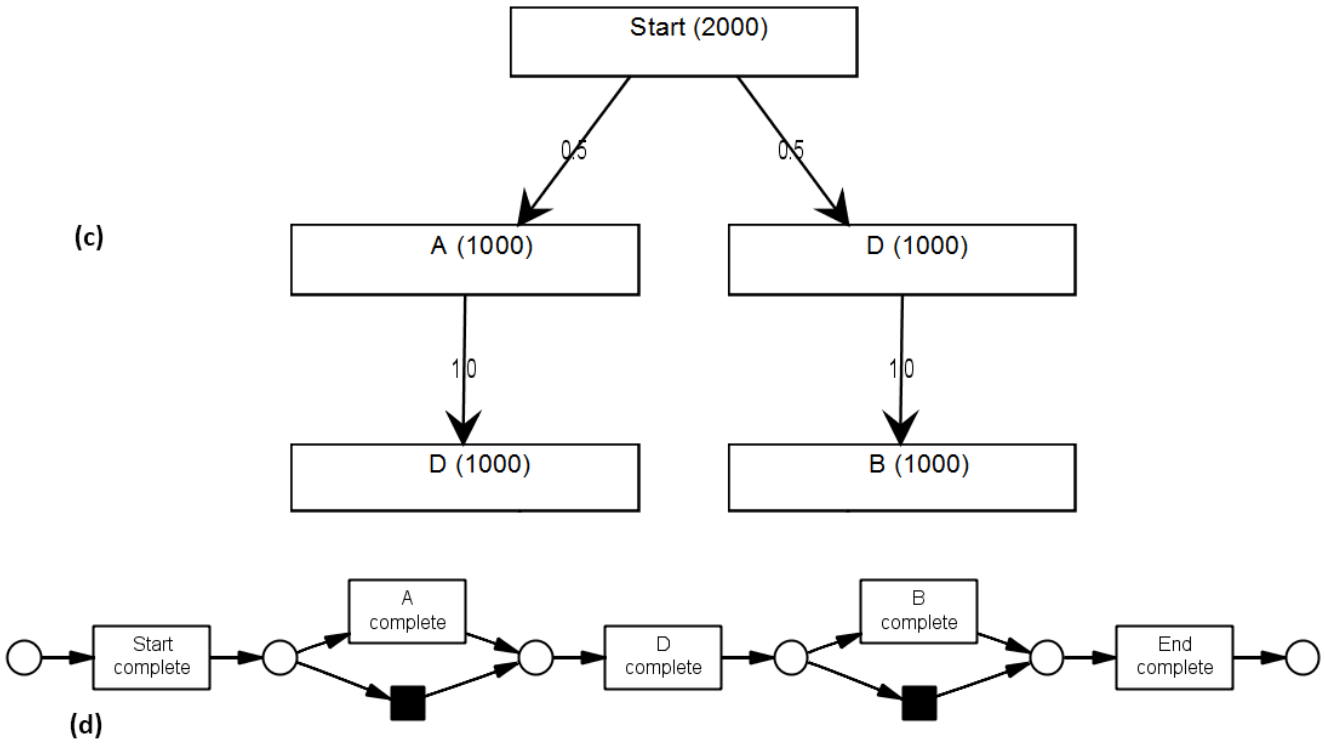

Fig. 2: Example 2: Process models obtained using our approach (c) and using Heuristic Miner (d).

a "fully reliable" model of the input log. As such, it can be used to analyze the not 100% precision abstractions provided by other miners (like, e.g., HM). This facility can be obtained by comparing the log-tree to the model provided by a different miner, both quantitatively and qualitatively.

In the quantitative comparison, we plan to exploit a set of graph distance calculation algorithms (such as [13], [14], [15], [16], [17]), which differ for their capability of considering semantic information on the models, or of distinguishing between action nodes and control flow information. The most suitable one will be selected case by case.

As for the qualitative analysis, interestingly it could be performed interactively by domain experts. A tool will be devised to help experts find the differences between the two models, leaving them free in the evaluation of which not 100% precision abstractions are "desirable" in their domain.

V. Conclusions

In many areas, logs recording the traces of execution of process instances are used for *trace retrieval*, or for *mining a model of the process*. In the current literature, such two tasks have been managed independently of each other, and relying on very different techniques. In this paper, we propose an innovative approach aiming at integrating them. The core notion of our proposal is the log-tree, which constitutes a bridge between the notions of (log) index and of process model. We propose a mining algorithm to build the log-tree, and an algorithm exploiting it for trace retrieval.

Being also an index, the log-tree must have a 100% precision, thus (usually) providing less abstractions than process models built by other process miners. However, our approach is *not in contrast* to the standard mining algorithms (like HM). Indeed, it has different goals, and may be *conciliated* with them to achieve further objectives. In Section 4, we have identified two new lines of research to do so, that we will explore in our future work. Additionally, in our future work, we want to investigate more flexible ways of providing trace retrieval on the log-tree, along the line we proposed in the context of time series retrieval [18].

We will also complete the implementation of our facility as a plugin in the ProM 6(see [2]) framework.

Acknowledgments

This research is partially supported by the GINSENG Project, Compagnia di San Paolo.

References

[1] W. V. der Aalst, *Process Mining. Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[2] H. Verbeek, J. Buijs, B. Dongen, and W. Aalst, "ProM 6: The Process Mining Toolkit," in *Proc. of BPM Demonstration Track 2010*, ser. CEUR Workshop Proceedings, M. L. Rosa, Ed., vol. 615, 2010, pp. 34–39.

[3] B. F. V. Dongen, N. Busi, and G. M. Pinna, "An iterative algorithm for applying the theory of regions in process mining."

[4] V. Rubin, B. F. V. Dongen, E. Kindler, and C. W. Gnther, "Process mining: A two-step approach using transition systems and regions," BPM Center Report BPM-06-30, BPM Center, Tech. Rep., 2006.

[5] A. Aamodt and E. Plaza, "Case-based reasoning: foundational issues, methodological variations and systems approaches," *AI Communications*, vol. 7, pp. 39–59, 1994.

[6] M. Minor, S. Montani, and J. A. Recio-Garca, "Process-oriented case-based reasoning," *Inf. Syst*, pp. 103–105, 2014.

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

[8] A. Cordier, M. Lefevre, P.-A. Champin, O. Georgeon, and A. Mille, "Trace-Based Reasoning — Modeling interaction traces for reasoning on experiences," in *The 26th International FLAIRS Conference*, May 2013. [Online]. Available: http://liris.cnrs.fr/publis/?id=5955

[9] D. B. Leake and J. Kendall-Morwick, "Towards case-based support for e-science workflow generation by mining provenance," in *Proc. ECCBR 2008, Advances in Case-Based Reasoning*, ser. Lecture Notes in Computer Science, K. Althoff, R. Bergmann, M. Minor, and A. Hanft, Eds., vol. 5239. Springer, 2008, pp. 269–283.

[10] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *Int. J. Cooperative Inf. Syst.*, vol. 23, no. 1, 2014. [Online]. Available: http://dx.doi.org/10.1142/S0218843014400012

[11] A. Weijters, W. V. der Aalst, and A. A. de Medeiros, *Process Mining with the Heuristic Miner Algorithm, WP 166*. Eindhoven University of Technology, Eindhoven, 2006.

[12] J. Tobin and C. Vogel, "A user-extensible and adaptable parser architecture," *Know.-Based Syst.*, vol. 22, no. 7, pp. 516–522, oct 2009.

[13] R. Dijkman, M. Dumas, and R. Garca-Banuelos, "Graph matching algorithms for business process model similarity search," in *Proc. International Conference on Business Process Management*, ser. Lecture Notes in Computer Science, U. Dayal, J. Eder, J. Koehler, and H. Reijers, Eds., vol. 5701. Springer, Berlin, 2009, pp. 48–63.

[14] M. LaRosa, M. Dumas, R. Uba, and R. Dijkman, "Business process model merging: An approach to business process consolidation," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 2, p. 11, 2013.

[15] S. Montani, G. Leonardi, S. Quaglini, A. Cavallini, and G. Micieli, "Improving structural medical process comparison by exploiting domain knowledge and mined information," *Artificial Intelligence in Medicine*, vol. 62, no. 1, pp. 33–45, 2014.

[16] R. Bergmann and Y. Gil, "Similarity assessment and efficient retrieval of semantic workflows," *Information Systems*, vol. 40, pp. 115–127, 2014.

[17] S. Montani, G. Leonardi, S. Quaglini, A. Cavallini, and G. Micieli, "A knowledge-intensive approach to process similarity calculation," *Expert Syst. Appl.*, vol. 42, no. 9, pp. 4207–4215, 2015.

[18] A. Bottrighi, G. Leonardi, S. Montani, L. Portinale, and P. Terenziani, "A time series retrieval tool for sub-series matching," *Appl. Intell.*, vol. 43, no. 1, pp. 132–149, 2015.