MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Workflow Trees for Representation and Mining of Implicitly Concurrent Business Processes

Daniel Nikovski, Akihiro Baba

TR2007-072 October 2007

Abstract

We propose a novel representation of business processes called workflow trees that facilitates the mining of process models where the parallel execution of two or more sub-processes has not been recorded explicitly in workflow logs. Based on the provable property of workflow trees that a pair of tasks are siblings in the trees if and only if they have identical respective workflow-log relations with each and every remaining third task in the process, we describe an efficient business process mining algorithm of complexity only cubic in the number of process tasks, and analyze the class of processes that can be identified and reconstructed by it.

MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Workflow Trees for Representation and Mining of Implicitly Concurrent Business Processes

Daniel Nikovski
Akihiro Baba

TR2007-72 October 2007

Abstract

We propose a novel representation of business processes called workflow trees that facilitates the mining of process models where the parallel execution of two or more sub-processes has not been recorded explicitly in workflow logs. Based on the provable property of workflow trees that a pair of tasks are siblings in the tree if and only if they have identical respective workflow-log relations with each and every remaining third task in the process, we describe an efficient business process mining algorithm of complexity only cubic in the number of process tasks, and analyze the class of processes that can be identified and reconstructed by it.

Released October 2007.

Workflow Trees for Representation and Mining of Implicitly Concurrent Business Processes

Daniel Nikovski[1] and Akihiro Baba[2]

[1] Mitsubishi Electric Research Laboratories,
201 Broadway, Cambridge, MA 02139, USA
nikovski@merl.com
[2] Mitsubishi Electric Corporation,
5-1-1 Ofuna, Kamakura, Kanagawa 247-8501, Japan
Baba.Akihiro@ab.MitsubishiElectric.co.jp

Abstract. We propose a novel representation of business processes called workflow trees that facilitates the mining of process models where the parallel execution of two or more sub-processes has not been recorded explicitly in workflow logs. Based on the provable property of workflow trees that a pair of tasks are siblings in the tree if and only if they have identical respective workflow-log relations with each and every remaining third task in the process, we describe an efficient business process mining algorithm of complexity only cubic in the number of process tasks, and analyze the class of processes that can be identified and reconstructed by it.

Key words: Process mining, business process management, implicit concurrency

1 Introduction

The organization and optimization of business processes within an enterprise is essential to the success of that enterprise in the marketplace, and the explicit management of business processes within dedicated software suites has emerged as an important class of information technology [1]. Key to the successful management of business processes is the nature of the models used for process representation, construction, maintenance, and improvement. Whereas some kind of graphical representation has been used almost universally, the types of proposed models and the semantics associated with them have varied widely. Some of the more popular representations include Petri nets [2], finite state machines and Markov models [3], as well as special-purpose graphic formalisms such as AND/OR trees [4] and block diagrams [5]. In most cases, these graphic representations are also associated with a corresponding formal language that is interpretable by BPM sequencing middleware. For an extensive comparison between business process modeling formalisms from several perspectives, see, for example [6].

The abundance of modeling formalisms suggests that there isn't a single best representation, but rather, multiple trade-offs exist when adapting formalisms to

a particular task, and the wide choice of available formalisms is in fact beneficial. The specific task of interest addressed in this paper is the learning from data of representations for processes with implicit concurrency. We propose a solution to this problem in the form of a novel representation for business processes, and an associated algorithm for mining such models from data with very favorable computational complexity (cubic in the number of process tasks). Whereas the proposed model has been tailored to facilitate mining of processes with implicit concurrency, it is still fully compatible with the most popular modeling languages and their underlying formalisms, such as BPMN, UML Activity Diagrams, and Workflow nets (WF-nets), and can easily be converted to any of them.

The general problem of process mining from data is described in Section 2, and several inductive and constructive solutions are reviewed. One specific problem of these solutions is identified and discussed: their inability to mine processes with implicit concurrency. Section 3 describes a new formalism for representation of business processes, called workflow trees (WF-trees). The properties and semantics of WF-trees are explained, and their relationship to more established formalisms, such as WF-nets, is discussed. It is shown that the relations between a pair of tasks and all remaining tasks are completely sufficient to indicate the relative position of the two tasks in the workflow tree. This property of WF-trees is exploited in Section 4 to develop an efficient algorithm for mining WF-trees from data. Section 5 discusses the current restrictions of WF-trees and several possible directions for eliminating them, and concludes the paper.

2 Process Mining and Implicit Concurrency

The objective of process mining algorithms and systems is to construct an explicit process model from recorded event logs [7]. This functionality is especially useful when a new business process management (BPM) system is deployed at a customer site and explicit models of the existing processes have to be produced as a starting point for analysis, process re-engineering, etc. The traditional alternative to process mining — the manual construction of process models, usually using graphic editors — can be very time- and labor-intensive, because it typically involves interviews with executives, and also very imprecise, because humans can only describe the way they *imagine* business processes operate, and not the way these business processes actually operate. At the same time, if the business processes already involve information technology (e.g. enterprise resource planning systems, customer relationship management systems), in all likeliness, abundant execution logs from these systems already exist. In such cases, using these execution logs to automatically extract process models can result in major savings in time and effort and improve model accuracy significantly.

To this end, process mining has been an active area of research and software development in recent years. The problem is to find a model of a business process (represented in a suitable formalism) solely by inspecting the relative order of tasks as manifested in logs collected from the repeated execution of the business process. It is assumed that N different tasks t_i, $i = 1, N$, $t_i \in T$ from the set

T can be distinguished in the execution log. The workflow logs are divided into disjoint episodes that correspond to the processing of one work case each. During one episode, the case takes one possible path through the process. An episode is represented as a sequence of tasks, and indicates the sequential order in which a particular case was processed. The objective of process mining algorithms, then, is to inspect the entire workflow log and induce a process model that could have produced this log. It is usually desired that the induced model be as compact as possible, and have no duplicate tasks.

Initial research recognized that process mining is a special case of inductive machine learning (ML), hence generic ML techniques, most commonly based on heuristic search, are applicable to this problem. Early examples of this approach included the algorithms of Cook and Wolf [3, 8], which employed greedy induction over model spaces representing Markov models and Petri nets. While successful, the heuristic nature of search in model spaces does not guarantee the discovery of the optimal model, where optimality is usually defined as a trade-off between model accuracy and parsimony, much like in other machine learning problems [9]. Further complicating the problem of finding the optimal model is the issue of data sufficiency — certainly, if the exact relationship among tasks is not manifested in the execution logs, a correct (and much less, optimal) model cannot be mined from these logs.

A major shift from heuristic search and inductive methods occurred with the emergence of constructive algorithms, such as α, $\alpha+$, and β [2, 10, 11]. These algorithms pre-compute the relations between each pair of tasks as manifested in the execution log and organize the identified relations in a tabular format. After that, the algorithms construct a model based on this relations table only, without having to examine the execution log ever again. This approach effectively renders the complexity of the mining part of such algorithms independent of the size of the execution log, which can be a very favorable property when large execution logs have to be mined. Furthermore, by making the assumption that the relation table is correct, the ability of the algorithm to find the optimal model can be analyzed in isolation from any data sufficiency and sample complexity issues.

Perhaps the best known example of this class of constructive algorithms is the α algorithm proposed by van der Aalst et al. [2]. The business process representation used by this algorithm is structured workflow nets (SWF-nets) — a carefully chosen and precisely defined subset of Petri nets that avoids undesirable situations such as deadlocks, incomplete tasks, indeterminate synchronization, etc. While the restrictions of SWF-nets with respect to general Petri nets are fairly significant, van der Aalst et al. [2] argue convincingly that SWF-nets in fact match the type of processes that exist in the real world, correspond to the constructs used in most deployed workflow systems, and also result in process descriptions that are easier to understand and maintain by human designers.

A significant novel idea of the α algorithm is to pre-process the execution log and determine the pair-wise relations between all pairs of tasks. These so called log-based ordering relations between a pair of tasks a and b are as follows:

- $a > b$ iff there exists at least one episode of the log where a is encountered immediately before b,
- $a \rightarrow b$ iff $a > b$ and $b \not> a$,
- $a \# b$ iff $a \not> b$ and $b \not> a$, and
- $a \parallel b$ iff $a > b$ and $b > a$.

The assumption of these algorithms is that the supplied workflow log is complete, i.e. it reflects correctly the relations between the tasks in the real process that produced the log. In practice, this requirement means that if all tasks that can potentially follow each other, in fact do so in at least one trace of the log. The issue of sample complexity, i.e. the problem of estimating the expected necessary size of the workflow log so that it is complete merits additional interest, but is orthogonal to the operation of the mining algorithm — rather, the mining algorithm assumes that the workflow is complete, and the computed relations table is correct.

Once the relation between each pair of tasks has been identified to be one of these four relations, the algorithm proceeds to construct a minimal SWF-net that satisfies the relations. Based on the provable property that $a \rightarrow b$ implies that a SWF-net place exists immediately between tasks a and b, van der Aalst et al. [2] devised an algorithm that builds an SWF-net in eight steps, without any heuristic search. The key step of the algorithm is to identify pairs Y of maximal sets of tasks A and B, such that all tasks in A have relation $\#$ between each other; similarly, all tasks in B have relation $\#$ between each other; for any pair of tasks $a \in A$ and $b \in B$ it is true that $a \rightarrow b$; no supersets of A and B, respectively, exhibit these properties. When such a pair (A, B) has been identified, the algorithm creates a new place P of the SWF-net, adds transitions from all tasks $a \in A$ to P, and transitions from P to all tasks $b \in B$.

The α algorithm is able to mine a large class of SWF-nets, with several limitations. One of them is that the algorithm cannot mine correctly nets with short loops (of length one or two tasks). This problem was remedied by subsequent extensions of the algorithm: de Medeiros et al. [10] proposed the $\alpha+$ algorithm based on an extended notion of log completeness and two new relations between tasks, and Wen et al. [11] proposed the β algorithm which exploits the temporal span of tasks (interval between start and end of tasks) to discover short loops.

Another limitation of the α algorithm and its derivatives is that they cannot detect all cases of concurrency in a business process. Concurrent tasks in SWF-nets are represented by means of a construct involving auxiliary AND-split and AND-join tasks (cf. Fig. 1). We will refer to this construct as an AND-block. If we compare it to the case of task choice (exclusive OR, or an OR-block), where only one of several tasks is executed, it is evident that an OR-block involves no such auxiliary tasks (cf. Fig. 2). The α algorithm can mine processes with AND-blocks as long as the two auxiliary tasks, the AND-split and the AND-join, have been recorded explicitly in the workflow log. We will call such processes explicitly concurrent, i.e., when concurrency is present, the initiation and completion of parallel execution is explicit in the log.

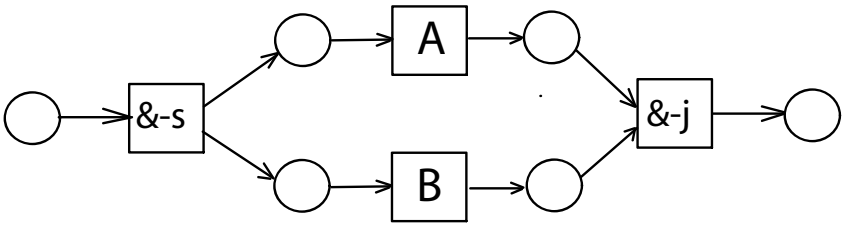

Fig. 1. A WF-net for representing parallel execution: tasks A and B are executed concurrently. Here the tasks labeled &-s and &-j are auxiliary and have the sole purpose of explicitly specifying concurrency.

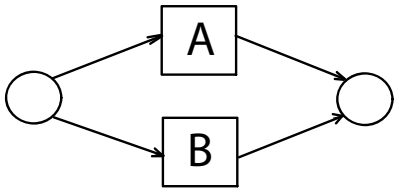

Fig. 2. A WF-net for representing exclusive choice: either task A or task B is executed, but not both.

However, it cannot be expected that workflow logs would contain explicit AND-splits and AND-joins, because they do not correspond to actual tasks in the problem domain — whenever parallel execution has been performed in a given legacy IT system, the decision to initiate it and the logic to synchronize its completion is usually buried somewhere deep into executable code, and it is precisely the objective of the process mining algorithm to extract it and model it explicitly.

When explicit AND-splits and AND-joins are absent from the workflow file (which we expect to be the typical situation), the mining algorithm would have to deal with implicitly concurrent business processes. In numerous cases, the α algorithm and its descendants would have difficulties in handling implicit concurrency. One specific instantiation of this problem is when an AND-block is nested within an OR-block. For example, van der Aalst [2] discussed the process in Fig. 3, and concluded that if the synchronizing AND-split and AND-join tasks were not present in the workflow log, the exact workflow net could not be recovered by the α algorithm. (Still, a behaviorally equivalent, but not sound, WF-net could be discovered by that algorithm.)

There are several possible explanations of why implicit concurrency is challenging for the α algorithm and its descendants. The first is in the nature of SWF-nets as process representations — although they are very powerful and versatile in terms of the type of processes that can be represented, there is an inherent asymmetry in the way AND-blocks and OR-blocks are represented. Since the α algorithm does not create new tasks other than those already present in the workflow log, it cannot create explicit AND-blocks.

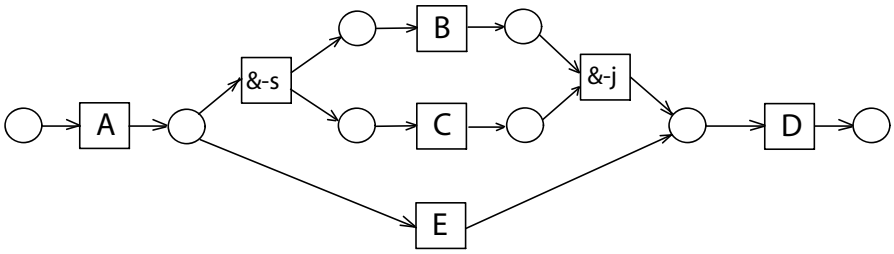

Fig. 3. This WF-net that cannot be recovered by the α algorithm, if the auxiliary tasks &-s and &-j are missing from the workflow log.

A second possible explanation lies in the way the α algorithm constructs the WF-net: it identifies sets of tasks which are in the $\#$ and $\rightarrow$ relations between each other, but *never* analyzes tasks that have the $\|$ relation between each other. The $\|$ relation is indicative of possible concurrency, but this class of algorithms never identifies this concurrency as a step of their operation; rather, concurrent tasks end up being represented as such merely as a side effect of placing the tasks in the correct sequential or exclusive-choice order.

This analysis suggests that perhaps it would be worthwhile to explore alternative representations and mining algorithms that can handle implicit concurrency better, while still aiming at constructive solutions that build compact relation tables from workflow logs. Another desirable property of such algorithms would be more favorable computational complexity — the α algorithms and its derivatives are usually exponential in the number of tasks N, since they involve search within the space of all pairs of sets of tasks, i.e. the powerset of the set of all tasks. For practical purposes, a mining algorithm of low-degree polynomial complexity would be much more desirable.

3 Workflow Trees for Representation of Business Processes

We propose a representation of business processes that is based on the natural hierarchical organization of work in most enterprises. The representation is in the form of an ordered tree, where the leaves of the tree represent tasks, and the internal nodes of the tree represent the functional blocks in which these tasks are organized. This representation is similar to the block representation used by Schimm [5] and the AND-OR graphs proposed by Silva et al. [4] in the type of the blocks used. Based on its hierarchical organization, it is also close to the Hierarchical Task Networks [12] that are popular in the field of Artificial Intelligence, and to the way sub-diagrams can be defined in UML 2.0 Activity Diagrams.

In this paper, we will consider trees that have four building blocks, labeled as follows: parallel (AND), choice (OR), sequence (SEQ), and iteration (ITER). The meaning of the AND and OR blocks is as shown in Figs. 1 and 2, in Petri

net notation. The meaning of the SEQ construct is obvious, and is shown in Fig. 4. For the iteration construct, two definitions are possible, depending on whether zero executions of a task are allowed, or it has to be executed at least once. The two alternative definitions are shown in Fig. 5.

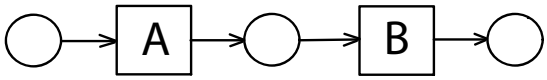

Fig. 4. A WF-net that specifies sequential execution: tasks A and B are always executed strictly in this order.

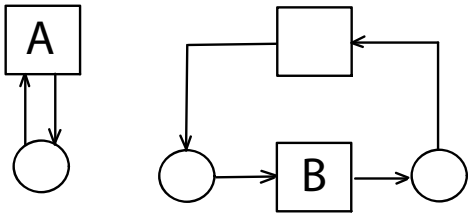

Fig. 5. Two possible WF-nets that specify iterative execution. The net on the left allows zero or more executions of task A, while the net on the right specifies that task B should be executed at least once (and possibly many more times).

These constructs are very similar to those used in van der Aalst and van Hee [1] (with the exception of the iteration construct, which must involve at least two tasks there). It has been shown that by starting with one of these constructs, and recursively substituting its component tasks with compound blocks of more tasks, a large class of sound and safe nets can be constructed. Our proposal for workflow trees formalizes this intuition: the structure of the tree prescribes the steps that must be taken during this process of top-down recursive construction of a business process. It also describes a way to convert a workflow tree (WF-tree) into a SWF-net: by traversing the WF-tree in any convenient order, each tree node is replaced by its corresponding Petri net, as described above, and if any of the children of this node are nodes themselves, the procedure is recursively repeated until all tasks in the resulting SWF-net are atomic tasks. As an example, Fig. 6 shows the WF-tree that would result in the SWF-net previously shown in Fig. 3, if expanded as described.

While this general approach to constructing business process models is intuitive and has been explored before, the specific representation in a tree-like form that we propose allows us to analyze and identify the properties of this representation that are useful for the purposes of process mining. In particular, we are interested in the relations between pairs of tasks that are entailed by

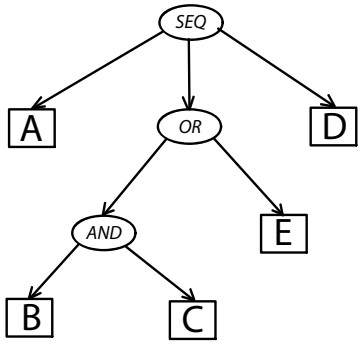

Fig. 6. A workflow tree that corresponds to the WF-net from Fig. 3.

this representation. We define a set of relations AND, OR, SEQ, and $ITER$ that are n-ary, and can hold between two or more tasks. Two tasks in the WF tree have one of these relations between each other. (In this case, the relation is binary.) *We specify that the binary relation between a pair of tasks in a WF-tree is determined by the node of the tree that is the least common ancestor (LCA) of these two tasks.* For example, for the SWF-net in Fig. 3 (respectively, the tree in Fig. 6), the tasks A and E are in the SEQ relation, and B and E are in the OR relation.

In the general case, it would be possible to have process models with nested blocks of the same type, for example an OR block nested immediately within another OR block. In the corresponding WF-tree, this would be expressed as one OR node having as a child (direct descendant) another OR node. While certainly possible and valid, such WF-trees are redundant, and it is usually desirable to eliminate this redundancy. We define a *compact workflow tree* (CWF-tree) to be a workflow tree where no two nodes of the same label have a direct parent/child relationship. Compacting a redundant WF-tree to a CWF-tree can be done by traversing the WF-tree using any suitable (depth-first, breadth-first, etc.) *post-order* walk of the WF-tree, and when the current node has a child node of the same label, eliminating that node and adding its children directly as children of the current node. We will also stipulate that apart from the $ITER$ node, all other nodes must have at least two children. (In case they don't, they can simply be eliminated from the tree, without loss of correctness.)

Before analyzing the properties of the described relations, we will note that as a corollary of this specification and the nature of *our* specific definition of an iterative block, no two tasks can be in the $ITER$ relation. This is due to the fact that a tree node labeled with ITER always has only one child, and hence cannot be the LCA of any pair of distinct tasks. (This is true regardless of which alternative definition of an ITER block is chosen from the two shown in Fig. 5).

The remaining three relations have the following properties. When these relations are binary, the binary AND and OR are transitive and symmetric, while the binary SEQ is transitive and asymmetric ($(aSEQb) \Rightarrow \neg(bSEQa)$).

Ternary relations can be defined by $aRb \wedge bRc \Rightarrow R(a,b,c)$, whereas relations of arbitrary arity have the property that $R(a_1, a_2, \ldots, a_{n-1}) \wedge a_{n-1}Ra_n \Rightarrow R(a_1, a_2, \ldots, a_{n-1}, a_n)$. Here R can be any of the three relations AND, OR, and SEQ. Note that in combination with the asymmetry of the binary SEQ relation, the n-ary SEQ relation is guaranteed to hold only between arguments in the correct order, while the symmetry of the binary AND and OR ensure that their n-ary counterparts hold for an arbitrary order of their arguments. For the sake of analytical convenience, we will also define the symmetric relation LIN, such that $aLINb$ iff $aSEQb \vee bSEQa$. The meaning of this relation is linear order — it holds true between two tasks when one of them follows the other.

Note also that if three or more tasks are in the same relation, it is not necessarily true that each pair of them has the same LCA — since more than one tree node can be labeled with the same block label, it is completely possible that three or more tasks are in the same relation, but are not descendants of three different children of the same node.

What is true, though, is that any three tasks a, b, and c of the same WF-tree can have at most two distinct relations R_1, R_2 from the set $\{AND, OR, LIN\}$ among them.

Lemma 1. *$aR_1b \wedge bR_2c \Rightarrow aR_1c \vee aR_2c$, for $R_1, R_2 \in \{AND, OR, LIN\}$.*

Proof. This lemma follows from the general property of trees that three nodes in the same tree can have at most two distinct pairwise LCA nodes. Let the LCA of tasks a and b be denoted by L_1 and labeled with relation R_1. Similarly, let the LCA of tasks b and c be denoted by L_2 and labeled with relation R_2. Since both L_1 and L_2 are in the same tree, there are three possible cases:

- $L_1 \equiv L_2$. This implies that $R_1 \equiv R_2$, and aR_1c, for a total of only one relation among a, b, and c.
- L_1 is a parent of L_2. Since L_2 is the LCA of b and c, they must be among its children. However, a cannot be a child of L_2, because then the LCA of a and b would be L_2, and not L_1. Then, since a is a child of L_1, and c is a child of L_2, which in its turn is a child of L_1, the LCA of a and c must be L_1, with aR_1c holding true. Note that the LCA of a and c cannot be some node between L_1 and L_2, because then that node would be the LCA of a and b, and not L_1. In this case, only two relations R_1 and R_2 hold among the three tasks.
- L_2 is a parent of L_1. Using similar reasoning, it follows that aR_2c, for a total of two relations among the three tasks, again.

We have established that the property holds in each of the three cases. □

Furthermore, due to the symmetry of the three relations AND, OR, and LIN, the lemma holds for all possible symmetric exchanges in the order of tasks in these relations. A direct corollary of this lemma (in one respective instantiation as regards relation symmetry) is that if two tasks a and b are in relation R_1 (aR_1b), and one of them (a) is in relation R_2 with some third task c (aR_2c),

there are only two possibilities for the relation between b and c: it is either bR_1c or bR_2c. From the proof of the lemma, we saw that the former case (bR_1c) holds when the LCA of a and b is a descendant of the LCA of a and c, while the latter case (bR_2c) holds when the LCA of a and c is a descendant of the LCA of a and b.

The latter case is of particular interest, and it is true that the logical implication in question holds in the other direction, too, even regardless of the exact relation between a and b. By defining $LCA(\cdot,\cdot)$ to be the function that returns the node of a WF-tree that is the LCA of its two arguments, and the binary relation *Descendant* such that $Descendant(d,a)$ holds true when node d is a descendant of node a, we can show that if nodes a and b happen to share the same relation R respectively with a third task c, it is necessarily true that their LCA is a descendant of their respective LCAs with this third task:

Lemma 2. $aR_1b \wedge aR_2c \wedge bR_2c \wedge R_1 \not\equiv R_2 \Rightarrow Descendant[LCA(a,b), LCA(a,c)]$.

Proof. Suppose that the consequent were not true. Since $R_1 \not\equiv R_2$, it is not possible that the LCA of each pair of nodes is the same, because it can be labeled with only one relation. The only remaining possibility is that $L_1 \doteq LCA(a,b)$ is an ancestor to $L_2 \doteq LCA(a,c)$. But then, since c is a descendant of L_2, and L_2 is a descendant of L_1, the LCA of b and c would be L_1. Then it would be true that bR_1c, because L_1 is labeled with relation R_1, whereas we know from the premise that bR_2c and $R_1 \not\equiv R_2$. So, by contradiction, it must be true that $LCA(a,b)$ is a descendant of $LCA(a,c)$. □

The same stipulation about the validity of this lemma with respect to the symmetry of R_1 and R_2 applies here, too. It follows immediately that $LCA(a,b)$ is a descendant of $LCA(b,c)$, as well. We can also prove that $LCA(a,c) \equiv LCA(b,c)$:

Lemma 3. $(aR_1b \wedge aR_2c \wedge bR_2c \wedge R_1 \not\equiv R_2) \Rightarrow (LCA(a,c) \equiv LCA(b,c))$.

Proof. Since both $LCA(a,c)$ and $LCA(b,c)$ are ancestors to $LCA(a,b)$ by virtue of Lemma 2, and three leaves in the same tree can have at most two distinct LCA nodes, then they must be the same node, i.e. $LCA(a,c) \equiv LCA(b,c)$. □

Also note that the condition that exactly *two* relations hold among the three tasks in Lemmata 2 and 3 is essential: if it is the same (one) relation that holds between each pair of tasks, nothing can be said about the relative position in the tree or number of their respective LCA nodes.

Now we are prepared to analyze the relations between a pair of tasks and *all* other tasks, and prove that *two tasks are children (direct descendants) of the same node iff they are in the same respective relation with all other tasks.* This property holds for compact workflow trees that do not contain redundant parent/child nodes of the same label, and also do not contain intermediate nodes of type $ITER$.

Theorem 1. $(\forall_c \exists_R aRc \wedge bRc) \Leftrightarrow [\exists_L Child(a,L) \wedge Child(b,L)]$.

Proof. First, we will prove the theorem in the forward direction: $(\forall_c \exists_R aRc \wedge bRc) \Rightarrow [\exists_L Child(a, L) \wedge Child(b, L)]$. Let $L_1 \doteq LCA(a, b)$, and let the relation between a and b be R_1. We will prove the forward property by showing that under its hypothesis $(\forall_c \exists_R aRc \wedge bRc)$, no internal node of the CWF-tree can exist on the paths between L_1 and either a or b, so L_1 must be the parent of both a and b.

For any task c different from a and b, the relation between a and c and b and c is the same, from the hypothesis of the theorem. Let that relation be R_2. There are two cases, depending on whether R_1 and R_2 are the same:

- $R_1 \not\equiv R_2$. Let also $L_2 \doteq LCA(a, c) \equiv LCA(b, c)$, the latter by virtue of Lemma 3. Then, by virtue of Lemma 2, L_2 is an ancestor of L_1, that is, it is *not* on the path between L_1 and either a or b.
- $R_1 \not\equiv R_2$. Neither Lemma 2 nor Lemma 3 apply. Let $L_2 \doteq LCA(a, c)$ and $L_3 \doteq LCA(b, c)$. At least two of $\{L_1, L_2, L_3\}$ must coincide, and possibly all three might coincide. If all three coincide, then neither L_2, nor L_3 are descendants of L_1, so, just like above, they are not on the path between L_1 and either a or b. (In this case, a, b, and c are siblings.)
  If one of the nodes L_2 or L_3 differ from L_1 (say, L_2 for definiteness), then we can prove that it must be strictly an ancestor of L_1. (The other one, L_3 in this case, must coincide with either L_1 or L_2.) We can do that by means of contradiction to the hypothesis of the theorem. Let's suppose that L_2 is not an ancestor of L_1, but a descendant of L_1. Then, there must be at least one other node L_4 on the path from L_1 to L_2 that is labeled differently from L_1 and L_2 (both labeled with R_1). (This is true, because we are considering compact workflow trees which do not allow a parent/child relationship between identically labeled nodes.) Let this node L_4 be labeled with relation $R_4 \not\equiv R_1$. However, then this node must have at least one descendant different from a, b, and c. This is true because L_4 must have at least two subtrees — one of them contains L_2, and hence a and b; c is not in any of the two subtrees, because it is in a subtree of L_1 that does not contain L_4; and the remaining subtree(s) of L_4 must have at least one task leaf. Call this task d. But from the position of L_4 between L_1 and L_2, it follows that $L_4 = LCA(a, d)$ and $L_1 = LCA(b, d)$. Since L_4 and L_1 are labeled differently, it would appear that aR_4d and bR_1d, thus violating the hypothesis that a and b are in the same relation with *all* other tasks in the tree. This would lead to contradiction, so we can conclude that L_2 (and by the same reasoning, L_3), cannot possibly be descendants of L_1.

Above we showed that, in all situations, no internal node that is the LCA of either a or b with some third node c can be a descendant of L_1. However, it is also true that any internal node of the tree is the LCA of either a or b with at least some other node c in the tree, since any node has at least two subtrees, one of which contains a (respectively, b), and the other subtree has at least one task as descendant. It then follows that no internal node in the tree can be a descendant of L_1, that is, the LCA of a and b is also their direct parent node. This proves the theorem in the forward direction.

Proving the entailment in the reverse direction — $[\exists_L Child(a, L) \wedge Child(b, L)] \Rightarrow (\forall_c \exists_R aRc \wedge bRc)$ — is straightforward. If a and b are children of the same internal node L_1, then the paths connecting either of them to any third task c are identical, except for the edge from L_1 to each of them. Since these two paths include the exact same sequence of internal nodes, whichever one of these nodes is $LCA(a, c)$, is also $LCA(b, c)$. This proves the theorem in the reverse direction, too. □

This theorem shows that we can identify tasks that must have the same parent node in the CWF-tree by comparing their respective relations with every other task — if they all match, then the two tasks share the same parent. We will use this theorem to devise a computationally efficient process mining algorithm in the next section. Note that the analysis will be limited to CWF-trees without $ITER$ nodes, since the presence of such nodes makes impossible the application of this theorem.

4 Mining of Workflow Trees

In the previous section, we assumed that a CWF-tree was given, and analyzed the properties of the relations among its tasks. In this section, we describe how such a tree can be constructed, if all that is given is a complete workflow log from the operation of the corresponding process. We will use a definition of log completeness identical to that proposed by van der Aalst et al. [2].

Before we describe the algorithm for mining workflow trees, we have to explain how all possible pairwise relations between two tasks in a model are determined. The binary relation AND is identical to the relation $\|$ used in the α algorithm: $aANDb \Leftrightarrow a \parallel b$. The relation SEQ is based on the relation $\rightarrow$ from that algorithm ($a \rightarrow b \Rightarrow aSEQb$), but unlike the latter, is transitive, and is more comprehensive. From the above implication and the transitivity property $aSEQb \wedge bSEQc \Rightarrow aSEQc$, it follows that $aSEQb \wedge b \rightarrow c \Rightarrow aSEQc$, that is, the relation SEQ is simply the transitive closure of $\rightarrow$, and can be found by any suitable algorithm, for example Floyd-Warshall of complexity $O(N^3)$ [13]. As described previously, $aLINb \Leftrightarrow aSEQb \vee bSEQa$. Finally, the OR relation is based on the $\#$ relation, but is much more limited. It holds only when the SEQ relation does not hold: $aORb \Leftrightarrow a\#b \wedge \neg(aLINb)$.

Consequently, the first step of the mining algorithm is to partition the set of all task pairs (t_i, t_j), $i = 1, N$, $j = 1, N$, $i \neq j$ into three subsets of task pairs that obey the original three relations $\rightarrow$, $\|$, and $\#$, respectively. This is done by means of establishing the $>$ relation first by performing a single scan of all traces in the workflow log, identically to the operation of the α algorithm [2]. The computational complexity of this step is linear in the length of the workflow log, but is independent of the number of tasks N. Establishing $\rightarrow$, $\|$, and $\#$ from $>$ can be done in time $O(N^2)$.

The resulting partition of task pairs can be represented conveniently in the matrix M^α whose entry $M^\alpha_{i,j}$ contains the relation label for the pair (t_i, t_j),

$i = 1, N$, $j = 1, N$, $i \neq j$. The diagonal entries $M_{i,i}$, $i = 1, N$ are undefined and excluded from consideration. Note that M^{α} is *not* symmetric, in general.

The second step is to build the relation matrix M of the workflow tree mining algorithm, whose entries are based on the entries M^{α} and the definitions described above. The order of filling in the matrix M is strictly as listed above: AND, SEQ, LIN, and OR, and since LIN labels overwrite SEQ labels, the end result is a partition of the task pair set into three relation subsets labelled with AND, OR, and LIN. Again, the diagonal elements of M are undefined and excluded from consideration. Note that in contrast to M^{α}, M *is* symmetric. The complexity of this step is $O(N^2)$.

The third, and central, step of the algorithm is to find the difference $\Delta_{i,j}$ between each distinct pair of rows (i, j), $i \neq j$ in the matrix M, defined as follows:

$$\Delta_{i,j} = \sum_{k=1}^{N} \delta(i, j, k), \qquad \delta(i, j, k) \doteq \begin{cases} 1 \text{ iff } i \neq k \wedge j \neq k \wedge M_{i,k} \neq M_{j,k} \\ 0, \text{ otherwise.} \end{cases} \tag{1}$$

If $\Delta_{i,j} = 0$ for a distinct pair of tasks (i, j), $i \neq j$, this means that these two tasks have identical respective relations with respect to all remaining tasks, and by virtue of Theorem 1 applied in the forward direction, they must have the same parent. In such case, we can build a workflow subtree that has a root node labeled with $M_{i,j}$, and children t_i and t_j.

When more than one element $\Delta_{i,j} = 0$ (excluding the symmetric element $\Delta j, i$ which is also necessarily zero because of the symmetry of Δ), there are two possible cases, depending on whether they involve overlapping tasks, or not. In the workflow tree, when more than two tasks have the same parent node, every pair of them (i, j) will have pairwise distance $\Delta_{i,j} = 0$ (From Theorem 1, applied in the reverse direction.) In contrast, if $\Delta_{i,j} = 0 \wedge \Delta_{k,l} = 0 \wedge \Delta_{i,k} \neq 0$, it follows that also $\Delta_{i,l} \neq 0 \wedge \Delta_{j,k} \neq 0 \wedge \Delta_{j,l} \neq 0$, i.e., (i, j) and (k, l) form two distinct subtrees with two different parent nodes. (Of course, nothing precludes these two parent nodes from being labeled with the same relation.)

Which of these two situations applies can be determined easily if we consider the graph whose N vertexes correspond to the tasks, and whose edges exist only between pairs of vertexes (i, j) such that $\Delta_{i,j} = 0$. It can be seen that each separate set of tasks that share the same parent node forms a distinct clique in this graph, and these cliques are disjoint. Identifying these cliques can be done by scanning the matrix Δ row-wise until a row i that contains element(s) equal to zero is found. This indicates that task t_i is a member of a clique. From inspecting that row, all tasks besides i that belong to this clique can be identified, and their respective rows can be marked by a suitable flag as already processed. The row-wise scan of Δ then continues identically, skipping rows marked as already processed.

Once all cliques have been found, a sub-tree is constructed for each of them. The root of this subtree is labeled with the relation that holds among these tasks. Due to the semantics of WF-trees, a sub-tree is simply a composite task that

can participate in a higher-level block just like any other atomic task. Because of this, we can create a new task label for each sub-tree so identified. Let the set of these new composite tasks be T_{new}; this set complements the initial set of atomic tasks T. The tasks $t_i \in T_{new}$ are given successive ordinal numbers beyond N. Let also the atomic tasks that are members of one of the cliques be defined as T_{inc}; each task in T_{inc} is a child of a member of T_{new}. Finally, let also a set T_{act} of active tasks be identified, and initialized at this point as $T_{act} := T$.

The complexity of this (third) step is $O(N^3)$, because it is dominated by the cost of computing the matrix Δ. (Identifying cliques and building sub-trees requires a single scan of Δ, or only $O(N^2)$.)

The next series of steps are largely similar to the one just described, only they work on a progressively modified active set of tasks. During each of these steps, the following sub-steps are performed:

1. The active set is modified to exclude the tasks that have already been included in some composite task: $T_{act} := T_{act} \backslash T_{inc}$. Their corresponding rows and columns in the matrix Δ are marked as processed.
2. The set of active tasks is expanded to include the new composite tasks: $T_{act} := T_{act} \cup T_{new}$. Furthermore, rows and columns are allocated for the new tasks in the matrices M and Δ. Pointers are kept from each new task to its children.
3. For each new composite task $t_i \in T_{new}$, its relation with the other tasks $t_j \in T_{act}$ is computed and stored in $M_{i,j}$. Let t_k be one of the children of t_i. When t_j is an atomic task, $M_{i,j} = M_{k,j}$, i.e., the composite task has the same relation with a third task as (any) one of its children has with this third task. (By construction, all of the children of t_i have the same relation with t_j.) When t_j is also a composite task, and t_l is one of its children, then $M_{i,j} = M_{k,l}$.
4. For each new composite task $t_i \in T_{new}$, its row difference with all other *active* tasks $t_j \in T_{act}$ is computed, similarly to Equation 1, but with the important distinction that this difference is taken only with respect to *active* tasks:

$$\Delta_{i,j} = \sum_{k=1, t_k \in T_{act}}^{N} \delta(i,j,k). \tag{2}$$

5. Cliques of tasks that have zero pairwise distance are identified exactly as described in Step 3 of the algorithm, new parent nodes for each of the cliques are created, and labeled with the respective relation. Each of the nodes forms a new subtree and corresponds to a new composite task. Analogously to Step 3, the subset of active tasks that are now included in some subtree is assigned to T_{inc}, and the set of new composite tasks is assigned to T_{new}.

The above five sub-steps are iterated until the set of active tasks T_{act} remaining after sub-step 2 includes only a single task. This task becomes the root of the mined workflow tree, and corresponds to the outermost block construct. The overall complexity of this series of steps is again $O(N^3)$, because new rows

and columns of the matrices M and Δ are introduced only for new composite tasks, and there can be at most $N-1$ such tasks. Each new row or column has $O(N)$ elements, and the computation of each element takes $O(N)$.

The last step of the algorithm is to re-order the children of all LIN nodes, so that the SEQ relation among them holds, and re-label those nodes with the label SEQ. This completes the construction of the workflow tree. Since, by construction, each composite node has at least two children, this workflow tree is also compact. The complexity of this step is $O(N^2 \log N)$, since the induced tree has at most $N-1$ internal nodes, each of which has $O(N)$ children which are sortable in $O(N \log N)$ time. Based on the complexity of each step, the overall computational complexity of the algorithm is $O(N^3)$.

5 Conclusion

We have described a representation of business processes called workflow trees that is intuitive and matches the hierarchical organization of most business processes used in practice. While similar to other business process representations used in the past, workflow trees have precise semantics and properties which derive directly from their tree-like representation. These properties can be leveraged to devise a computationally efficient process mining algorithm that can recover business process models with concurrent tasks that have not been specified as such explicitly in workflow logs. The algorithm operates by analyzing and comparing the mutual relations between pairs of tasks, suitably organized in matrices, and this determines its favorable computational complexity — cubic in the number of process tasks.

This computational efficiency is achieved at the expense of a slight sacrifice in the representational power of workflow trees in comparison to other formalisms, such as workflow nets [2]. The set of process models that can be represented by workflow trees is a strict subset of the set of models that can be represented by workflow nets — there exist some processes that can be represented by workflow nets, but not by workflow trees, most notably some processes with complex concurrency and mixed synchronization. It is debatable, though, whether such complex synchronization would be encountered often in actual business situations, and whether this would be a serious restriction in practice.

A more serious limitation of the current version of the mining algorithm is that it cannot recover models with looped execution. While such models are easily represented by workflow trees, using several possible iterative constructs, the mining algorithm proposed in this paper relies on the property of induced trees that each of their internal nodes must have at least two children. This effectively excludes iterative constructs from the set of blocks that can be used for building induced models.

However, this is not a principled restriction — in fact, the presence of looped execution can easily be detected as a by-product of computing the transitive closure SEQ of the $\rightarrow$ relation. If there exists a task a such that $aSEQa$, then the process must contain a loop. However, identifying how many loops exist,

where the corresponding $ITER$ constructs should be positioned in a workflow tree, and how the tree should be mined in the presence of such constructs, is still an open problem to be addressed by future work.

References

1. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
2. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16**(9) (2004) 1128–1142
3. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**(3) (1998) 215–249
4. Silva, R., Zhang, J., Shanahan, J.G.: Probabilistic workflow mining. In: KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, New York, NY, USA, ACM Press (2005) 275–284
5. Schimm, G.: Mining exact models of concurrent workflows. Comput. Ind. **53**(3) (2004) 265–281
6. List, B., Korherr, B.: An evaluation of conceptual business process modelling languages. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2006) 1532–1539
7. van der Aalst, W.M.P., Weijters, A.J.M.M.: Process mining: a research agenda. Comput. Ind. **53**(3) (2004) 231–244
8. Cook, J.E., Wolf, A.L.: Event-based detection of concurrency. In: SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (1998) 35–45
9. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education (1997)
10. de Medeiros, A., van Dongen, B., van der Aalst, W., Weijters, A.: Process mining: Extending the α-algorithm to mine short loops. BETA Working Paper Series, WP 113, Eindhoven University of Technology, Eindhoven (2004)
11. Wen, L., Wang, J., van der Aalst, W., Wang, Z., Sun, J.: A novel approach for process mining based on event types. BETA Working Paper Series, WP 118, Eindhoven University of Technology (2004)
12. Erol, K., Hendler, J., Nau, D.S.: Semantics for hierarchical task-network planning. Technical Report UMIACS-TR-94-31, University of Maryland at College Park, College Park, MD, USA (1994)
13. Sedgewick, R.: Algorithms in C. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)