Mining and Reasoning on Workflows

G. Greco, A. Guzzo, G. Manco, D. Saccà

RT-ICAR-CS-04-12 **Settembre 2004**

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Mining and Reasoning on Workflows

G. Greco[1], A. Guzzo[1], G. Manco[2], D. Saccà[1,2]

Rapporto Tecnico N.:
RT-ICAR-CS-04-12

Data:
Settembre 2004

[1] Università degli Studi della Calabria, DEIS, Via P. Bucci 41C, Rende (CS)
[2] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza, Via P. Bucci 41C, 87036 Rende(CS)

Mining and Reasoning on Workflows

Gianluigi Greco, Antonella Guzzo, Giuseppe Manco, Domenico Saccà

Abstract

Workflow management systems represent today a key technological infrastructure for advanced applications which is attracting a growing body of research, mainly focused in developing tools for workflow management, that allow the users both to specify the "static" aspects, like preconditions, precedences among activities, rules for exception handling, and to control its execution, by scheduling the activities on the available resources.

This paper deals with an aspect of workflows which has so far not received much attention even though it is crucial for the forthcoming scenarios of large scale applications on the web: providing facilities for the human system administrator for identifying the choices performed more frequently in the past that had lead to a desired final configuration.

In this context, we formalize the problem of discovering the most frequent patterns of executions, i.e., the workflow substructures that have been scheduled more frequently by the system. We attacked the problem by developing two data mining algorithms, on the basis of an intuitive and original graph formalization of a workflow schema and its occurrences.

The model is used both to prove some intractability results, that strongly motivate the use of data mining techniques, and to derive interesting structural properties for reducing the space of search for frequent patterns. Indeed the experiments we have carried out show that our algorithms outperform standard data mining algorithms adapted to discover frequent patterns of workflow executions.

Keywords: H.2.8.d Data Mining, H.2.4.p Workflow management.

I. Introduction

A workflow is a partial or total automation of a business process, in which a collection of *activities* must be executed by humans or machines, according to certain procedural rules. Modern enterprises increasingly use workflow technology for designing business processes, by means of management systems that provide mechanisms for formally specifying the schema of execution, for simulating its evolution under different conditions, for validating and testing whether it behaves as expected, and for evaluating the ability of a service to meet requirements with respect to throughput times, service levels, and resource utilization.

This paper deals with an aspect of workflows which has not so far received much attention even though it is crucial for the forthcoming scenarios of large scale applications on the web: providing facilities for the human system administrator to monitor the actual behavior of the workflow system in order to predict the "most probable" workflow executions. Indeed, in real world-cases, the enterprise must perform many choices during workflow execution; some choices may lead to a benefit, others should be instead avoided in the future. Data mining techniques may, obviously, help the administrator, by looking at all the previous instantiations (suitably collected into *log* files in any commercial system), in order to extract unexpected and useful knowledge about the process, and in order to take the appropriate decisions in the executions of future instances.

The discovered knowledge can be profitably used for solving problems such as:

Successful Termination Prediction: Assume that an execution is at a given point in which the administrator has to choose an activity to start, from a given set of potential activities. Then, she/he typically wants to know which is the choice performed in the past, that more frequently had led to a desired final configuration.

Identification of Critical Activities: In every workflow schema, there are some activities that can be considered *critical*, it the sense that they are scheduled by the system, in every successful execution. Some times, the system administrator may know in advance that a given activity is critical, but it often happens that this knowledge must be inferred by looking at the actual behavior of the system.

Failure/Success Characterization: By analyzing the past experience, a workflow administrator may be interested in knowing which discriminant factors characterize the failure or the success in the executions.

Workflow Optimization: The information collected into the logs of the system can be profitably used to reason on the "optimality" of workflow executions. For instance, the optimality criterion can be fixed w.r.t. some real-case interesting parameter, such as the quality of the service or the average completion time.

In this paper we concentrate on the first of the above problems: *Successful Termination Prediction*. We show that a crucial step towards an automatic solution to this problem consists of identifying the blocks of activities, called *patterns*, that have been more frequently scheduled together during the execution by the workflow system. To this aim, we propose two distinct algorithms for *frequent pattern mining* implementing sophisticated techniques which benefit from the peculiarities of the applicative context, thereby extending previous proposals for mining frequent structures in complex domains (such as frequent sequences, trees and graphs — see, e.g., [3], [31], [10], [16], [11], [28], [15], [29]) to the mining of workflow executions.

A. Related Work

The paper is about applying data mining techniques to the area of workflows and, as such, it presents a quite intuitive graph formalization of the main workflow concepts as the basic data structure on which data mining algorithms work. Therefore, a first area of related work is workflow modelling and analysis. Let us preliminarily point out that the paper is not aimed at developing a comprehensive workflow specification; so, even though our workflow model covers basic features required in workflow specification, it contains some simplifying assumptions. For instance, our model does not incorporate compensation or reset activities and assumes acyclicity, i.e., non-recursive workflows and non-iterative executions. Furthermore, the model does not directly support scheduling or verification tasks (see, e.g., [19]) and does not handle transactional properties of processes. The reader interested in the description of advanced features in workflow modelling is referred to [18], proposing a unifying model for concurrency control and recovery for processes. Other elaborated models are: the *Concurrent Transaction Logic-based model* ($\mathcal{CTR}$) [4], which enables to both describe and reason about workflow, by introducing a rich set of constraints [6]; the *state chart model* [26], [27], in which *triggers* are introduced in order to define *ECA* (Event Condition Action) rules for describing transitions among states; the *active object oriented model* [13], in which a workflow is modelled by integrating *ECA* rules with object-oriented concepts; the *Process algebra-based model* [20]; and, the *Petri Nets-based model* [21], that is a formalism having deep formal foundations, and that is profitably used for investigating different interesting properties for the process, such as liveness, and boundness (see, e.g., [22]).

Let us now review some work related to the the main topic of the paper, that is graph mining techniques specialized to handle constraints derived by the structures of workflow schemes and instances. The idea of *mining* execution traces has been already addressed in the context of process discovery (see, e.g.[1]), but the goal there is to use the information collected at run-time for deriving an "a posteriori" schema, that can model all logged executions and that is well suited for adapting the system to changing circumstances and removing imperfections in the initial design (see [23] for a survey on this topic).

Instead, in our approach the workflow schema is the starting point not the result: a number of executions are analyzed contextually and comparatively on the basis of the schema and with the goal of finding frequent patterns of activities, thus discovering useful knowledge for supporting the decision process in an enterprise. At the best of our knowledge, our work is the first in handling such a problem, that is a problem of mining graphs with constraints imposed by the structures of workflow schemes and instances. Under this perspective, the techniques we propose must be compared with other efforts paid by the database community for developing algorithms for mining frequent patterns both in relational databases and in complex domains. Most of these approaches are based on the *anti-monotone* property, first exploited in the seminal paper of Agrawal and Srikant [2] that introduces the *Apriori* algorithm: the idea is to generate the set of candidates of length $k + 1$, by combining in a suitable way the set of frequent patterns of size k, and then to check their frequencies. A quite simple generalization of this method is presented in [3] in order to mine sequential patterns.

A completely different approach has been proposed in [10], and goes under the name of *FP-growth* method. Essentially, the idea is to mine frequent instances with a top-down approach, i.e., by recursively projecting the database according to the frequent patterns already found, and then by combining the results of mining the projected databases. The extension to sequential pattern is the *PrefixSpan* algorithm [16]. A recent attempt for combining such a method with the *Apriori* approach has been done in [17].

As for the problem of mining patterns in complex domains, the discovery of frequent trees in a forest has been tackled in [31], while a first Apriori-based algorithm, called *AGM*, for identifying frequent substructures in a given graph dataset has been presented in [11]. We stress that this latter task constitutes nowadays a very active and still promising area of research for its interesting applications in web analysis and in bioinformatics.

For instance, the level-wise search performed by *AGM* has been adopted and further improved in the *FSG* algorithm [15], in which a smart strategy for labelling the generated subgraphs avoids many computational expensive sub-graph isomorphism computations. Moreover, some algorithms based on the projection method have been quite recently proposed as well: *gSpan* [28] discovers all the frequent subgraphs without candidate generation and false positive pruning, whereas *CloseGraph* [29] dramatically reduces the number of unnecessary frequent subgraphs generated, by exploiting the notion of *closed* patterns, i.e., patterns which are no proper subgraphs of any other pattern with the same support.

It is clear that such approaches could be in principle used to deal with the problem of mining frequent workflow instances, after a suitable adaptation for fitting the peculiarities of the specific applicative domain of workflow systems. In fact, one can think at modelling the workflow schema as a graph, and the executions of the workflow as a set of subgraphs complying with the graph representing the workflow schema.

However, the adaptation of the above mentioned methods to workflow mining is a challenging task, and it results unpractical from both the expressiveness and the efficiency viewpoint. Indeed, generation of patterns with such traditional approaches does not benefit from the exploitation of the executions' constraints imposed by the workflow schema, such as precedences among activities, synchronization and parallel executions of activities (see, e.g, [14], [24], [5]). In contrast, the algorithms proposed in the paper are novel mining techniques *specialized* to handle constraints derived by the structures of workflow schemes. And, in fact, several experiments, reported in Section V, confirmed that they outperform traditional data mining algorithms, even though suitably reengineered (in our implementations) to work with workflow instances.

We conclude the overview on related work by observing that, in order to model all the details of a workflow system, one viable way is to consider more expressive approaches, such as the multirelational data mining approaches [7]. Nonetheless, in Section V, we also show that as consequence of their generality in modelling different domains they poorly perform if compared with our algorithms specifically designed for the workflow domain.

B. Contribution of the Paper

In this paper, we investigate the possibility of exploiting data mining techniques within workflow management contexts, by proposing two algorithms for mining frequent workflow patterns of execution. Specifically, our contribution is as follows:

• We model the *Successful Termination Problem*, and we provide some intractability results that shed light into the intrinsic difficulty on reasoning over workflows. The in-depth theoretical analysis we provide strongly motivates the use of Data Mining techniques, thus confirming the validity of the approach.

• We define the notion of workflow *pattern* and of *weak pattern* of a workflow graph, where the latter is a syntactic restriction of the former. In particular we prove that weak patterns are well suited for mining tasks, as they can be recognized in a higly-parallelizable way and can be easily composed to discovery frequent patterns because of their interesting structural properties. Indeed we show that the space of all connected weak patterns constitutes a lower semi-lattice w.r.t. a particular relation precedence ($\prec$).

• By exploiting properties of weak patterns, we design two algorithms for mining frequent patterns that conform to the workflow specifications:

 – w-find, that performs a smart (level-wise) exploration of the lower semi-lattice, and

 – c-find, that mines frequent instances by composing connected components.

• We test w-find and c-find, by evaluating their performance and their scalability. We show that none of the algorithms is the best in absolute terms, by also evidencing the discriminant factors. Moreover, we compare these algorithms with existing techniques adapted to our particular domain. Several experiments confirmed the validity and the usefulness of these approaches.

We stress that our approach does not consider cyclic graphs (i.e., recursive workflow schemas and iterated executions) and other aspects of workflows such as compensation or reset activities. These assumptions have been required by the necessity of starting from a simplified model, yet covering important and typical features required in workflow specification, to take up an interesting and relevant topic that has not been given much attention in the literature so far. In fact, a significant number of technical challenges had to be faced for dealing even with basic features only. However, since there is no conceptual limitation in extending our algorithms for mining frequent instances w.r.t. more involved workflow models, we believe that this work might stimulate the data mining community in continuing our investigation and in facing some of the challenges we posed here.

C. Organization

The paper is organized as follows. Section II provides a formal model of workflows, and many complexity considerations on such a proposed model. A formalization of the problems of *Successful Termination Prediction* and of mining frequent patterns from workflow schemas is devised in Section III. The levelwise theory of worklow patterns is presented in Section IV, together with the algorithms w-find and c-find. Finally, Section V provides experimental validation of the approach.

II. The workflow abstract model

A significant amount of research has been already done in the specification of mechanisms for process modelling (see, e.g., [9] for an overview of different proposals). The most widely adopted formalism is the *control flow graph*, in which a workflow is represented by a labelled directed graph whose nodes correspond to the tasks to be performed, and whose arcs describe the precedences among them. Moreover, Workflow Management Coalition[1] has also identified additional controls, such as loops and sub-workflows.

In this paper, we do not refer to any particular model proposed in the literature. Rather, we next provide a simple (state based) model that covers most of the important important and typical features required in workflow specification. The model will be used for providing, in a rigorous way, both the syntax and the execution semantics. Hence, it will trace the formal framework (whose limitations have been already described in the Introduction) for developing our mining algorithms.

Definition II.1: A *workflow schema* $\mathcal{WS}$ is a tuple $\langle A, E, a_0, F, \mathtt{IN}, \mathtt{OUT}_{min}, \mathtt{OUT}_{max}\rangle$, where A is a finite set of *activities*, $E \subseteq (A - F) \times (A - \{a_0\})$ is an acyclic relation of precedences among activities, $a_0 \in A$ is the starting activity, $F \subseteq A$ is the set of final activities, while $\mathtt{IN}, \mathtt{OUT}_{min}$, and $\mathtt{OUT}_{max}$ are three functions assigning to each node a natural number ($A \mapsto \mathbb{N}$) as follows:

- $\forall a \in A - \{a_0\}$, $0 < \mathtt{IN}(a) \leq \mathit{InDegree}(a)$;
- $\forall a \in A - F$, $0 < \mathtt{OUT}_{min}(a) \leq \mathtt{OUT}_{max}(a) \leq \mathit{OutDegree}(a)$;
- $\mathtt{IN}(a_0) = 0$, and $\forall a \in F$, $\mathtt{OUT}_{min}(a) = \mathtt{OUT}_{max}(a) = 0$.

where $\mathit{InDegree}(a)$ is $|\{e = (b, a) \mid e \in E\}|$ and $\mathit{OutDegree}(a)$ is $|\{e = (a, b) \mid e \in E\}|$. $\square$

[1] *www.wfmc.org*

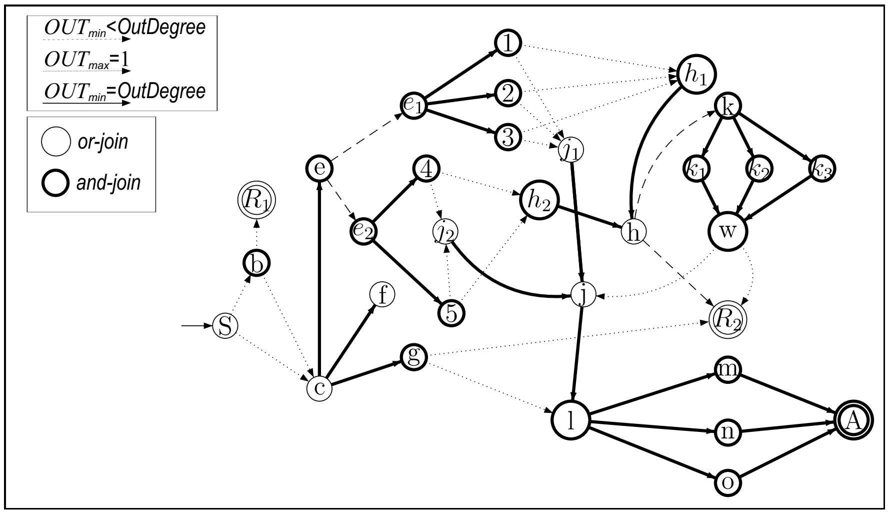


Fig. 1. An example Workflow Schema.

Roughly speaking, an activity a can start as soon as at least $\mathtt{IN}(a)$ of its predecessor activities have been completed. Two typical cases are: (i) if $\mathtt{IN}(a) = \mathit{InDegree}(a)$ then a is an *and-join* activity, for it can be executed only after all its predecessors are completed, and (ii) if $\mathtt{IN}(a) = 1$ is called *or-join* activity, for it can be executed as soon as one predecessor is completed. As commonly assumed in the literature, we will limit ourselves to consider only *and-join* and *or-join* activities, besides a_0: Indeed, by means of these two elementary types of nodes, it is possible to simulate also the behavior of any activity a such that $1 < \mathtt{IN}(a) < \mathit{InDegree}(a)$.

Once finished, an activity a activates some (non-deterministically chosen) subset of its outgoing arcs with cardinality between $\mathtt{OUT}_{min}(a)$ and $\mathtt{OUT}_{max}(a)$. If $\mathtt{OUT}_{max}(a) = \mathit{OutDegree}(a)$ then a is a *full fork* and if also $\mathtt{OUT}_{min}(a) = \mathtt{OUT}_{max}(a)$ then a is a *deterministic fork*, for it activates all its successor activities. Finally, if $\mathtt{OUT}_{max}(a) = 1$ then a is an *exclusive fork* (also called *XOR-fork* in literature), for it activates exactly one of their outgoing arcs.

For the sake of presentation, whenever it will be clear from the context, a workflow schema $\mathcal{WS} = \langle A, E, a_0, F, \mathtt{IN}, \mathtt{OUT}_{min}, \mathtt{OUT}_{max}\rangle$ will also be denoted by $\langle A, E, a_0, F\rangle$ or even simpler by $\langle A, E\rangle$.

A workflow schema can be represented in a graphical way by means of a directed acyclic graph, where the nodes corresponds to the activities in A, and the edges corresponds to the relation of precedence E (see Figure 1). Moreover, in order to represent the functions $\mathtt{IN}$,

$\texttt{OUT}_{min}$, and $\texttt{OUT}_{max}$, if an activity is an *and-join* (resp. *or-join*), we draw the corresponding node with a bold (resp. regular) circles; Finally, the nodes corresponding to *exclusive fork* (resp. *deterministic fork*) activities are such that their outgoing edges are marked with dotted (resp. bold) lines, while all the other edges are represented by dashed lines.

Example II.2: Figure 1 shows a sketch of a workflow schema representing a sales ordering process. The process is as follows. A customer issues a request to purchase a given product; the enterprise checks both the availability of the required stock and the reliability of the client. Moreover, if the client is reliable but the products are partially stocked, then a production will be planned. The final states can be the acceptance or the rejection of the order. Specifically, the initial task S corresponds to the "receive order" activity, the final tasks R_1 and R_2 are the rejecting of the order, while A is the acceptance. The activity e is the production that sends the request to some storehouse (either e_1 or e_2), which, in turns, forwards it to the respective repository $(1, 2, 3$ or $4, 5)$.

When at least one repository (no matter which one) has accepted the request, the task j_1 or j_2 proceeds to notify j. If there is no availability, the task h may send a request to the sales department (activity k) which forwards it to all wholesaler k_1, k_2, k_3; on the contrary the user request will be rejected (task R_2). Finally, the financial department (activity g) must assess if the reference are acceptable and if it is not, the order is rejected immediately (R_2); otherwise the activation of the task l will lead to a success.

It is worth noting that in this application, it could be crucial to characterize (with the help of the data mining techniques that we shall develop in the paper) the discriminant factors that will lead to an acceptance of the order requiring a planning of the production, in order to preventively accommodate the requests. $\triangleleft$

The formal semantics is specified by mapping the workflow schema into a transition system, where each execution consists of a sequence of states.

Definition II.3: Let $\mathcal{WS} = \langle A, E, a_0, F \rangle$ be a workflow schema. Then, the *state* S of an execution is identified by a tuple $\langle \mathit{Marked}, \mathit{Ready}, \mathit{Executed} \rangle$, with $\mathit{Ready}, \mathit{Executed} \subseteq A$, and $\mathit{Marked} \subseteq E$. $\Box$

Intuitively, the state of an execution is determined by the set ($\mathit{Executed}$) of activities which have been already executed, by the set (Ready) of activities which have received the inputs they need and which are, hence, ready for being executed, and by the set (*Marked*) of edges corresponding to the outputs of executed activities which will eventually be inputs to other activities. An execution is modelled by means of a transition system over such states. Then, if after t transitions (short: step t) the state is $S_t = \langle Marked_t, Ready_t, Executed_t \rangle$, the next state S_{t+1} is one of the outcomes of a non-deterministic transition function δ defined next.

Definition II.4: Let $\mathcal{WS} = \langle A, E, a_0, F \rangle$ be a workflow schema, and $S_t = \langle Marked_t, Ready_t, Executed_t \rangle$ be the state at the step t. Then, $\delta_{ws}(S_t)$ is the set of all states $\langle Marked_t \cup \delta Marked_{t+1}, Ready_{t+1}, Executed_t \cup Ready_t \rangle$, such that

i) $\delta Marked_{t+1}$ is a subset X of $\{(a,b) \mid a \in Ready_t, (a,b) \in E\}$ s.t. $\forall a \in Ready_t$, $\mathtt{OUT}_{min}(a) \leq |\{(a,b)| \, (a,b) \in X\}| \leq \mathtt{OUT}_{max}(a)$, i.e., each ready activity, say a, activates a number of outgoing arcs in the range defined by $\mathtt{OUT}_{min}(a)$ and $\mathtt{OUT}_{max}(a)$;

ii) $Ready_{t+1} = \{a | a \in (A - (Executed_t \cup Ready_t)), \, |\{(b,a)| \, (b,a) \in Marked_t\}| \geq \mathtt{IN}(a)\}$, i.e., an activity a becomes ready for execution as soon as at least $\mathtt{IN}(a)$ of its predecessor activities are completed. □

Now we are in the position to formally define a workflow execution. An execution starts with the state $S_0 = \langle \emptyset, \{a_0\}, \emptyset \rangle$, and at each step it applies the transition function δ_{ws}, until a final state is reached.

Definition II.5: Let $\mathcal{WS} = \langle A, E, a_0, F \rangle$ be a workflow schema, and δ_{ws} be a transition function. An *execution* e on a workflow schema $\mathcal{WS} = \langle A, E, a_0, F \rangle$ is a sequence of states $[S_0, ..., S_k]$ such that

i) $S_0 = \langle \{\emptyset\}, \{a_0\}, \{\emptyset\} \rangle$, and

ii) $S_{t+1} \in \delta(S_t)$ for each $0 < t < k$.

Moreover, if $Executed_k \cap F \neq \emptyset$ or $Ready_k \cup \delta Marked_k = \emptyset$ then e is said to be *terminating*; otherwise, it is said to be *partial.* □

Given an instance $e = [S_0, ..., S_k]$, the set $Executed_k$ is also denoted by $Executed(e)$. Note that in the above definition, a *terminating execution* e for which $Executed(e) \cap F = \emptyset$, corresponds to an abnormal execution which does not reach a final state. In this case, there are neither activities ready for being executed (i.e., $Ready_k = \emptyset$), nor outputs which may eventually activate other activities (i.e., $\delta Marked_k = \emptyset$); hence, e is said to be *unsuccessful.* Otherwise, e is said to be *successful* — observe that, a successful execution may terminate

step(t)	0	1	2	3	4	5	6	7
$\delta Marked_t$		(S, c)		$(c, e); (c, f); (c, g)$		$(e, e_1); (e, e_2); (g, R_2)$		$(e_1,1);(e_1,2);(e_1,3);(e_2,4);(e_2,5)$
$Ready_t$	S		c		$e;\ f;\ g$		$e_1;\ e_2;\ R_2$	
$Executed_t$		S	S	$S;\ c$	$S;\ c$	$S;\ c;\ e;\ f;\ g$	$S; c; e; f; g$	$S; c; e; f; g; e_1; e_2; R_2$

Fig. 2. Example of execution over the workflow of Example II.2.

with some ready activity that will be never executed, i.e., with $Ready_k \neq \emptyset$.

From now on, given a workflow schema $\mathcal{WS}$, the set of all the successful executions is denoted by $\mathcal{S}_{ws}$, while the set of all the unsuccessful executions is denoted by $\mathcal{U}_{ws}$.

Example II.6: An example of execution over the workflow schema presented in Example II.2 is reported in Figure 2. The indexed columns represent the steps of the execution.

Note that at the 5-th step, the financial department (activity g) has rejected the order (that is not been forwarded to l) causing the ending of the workflow execution. $\triangleleft$

As suggested by the previous example, the choices made during an execution may cause a success or a failure. Moreover, checking whether the workflow has a sequence of choices leading to a success is an intractable problem. Specifically, we next show that it is complete for the class **NP** of problems that are solvable in polynomial time by nondeterministic Turing machines — see [8], for some background on computational complexity.

Proposition II.7: Let $\mathcal{WS} = \langle A, E, a_0, F\rangle$ be a workflow schema. Then, (i) deciding whether there exists an execution e that reaches a final state (i.e., $Executed(e) \cap F \neq \emptyset$) is **NP***-complete, but (ii) the problem becomes* **P***-complete if all nodes in A are full forks.*

Proof:

(i) Membership in **NP** is trivial. For the hardness, recall that, given a Boolean formula Φ over variables $X_1, ..., X_m$ the problem of deciding whether it is satisfiable is **NP**-complete [8]. W.l.o.g. assume Φ to be in conjunctive normal form. Then, we define a workflow schema $\mathcal{WS}(\Phi) = \langle A, E, a_o, \{Sat\}\rangle$, such that A consists of an initial activity a_0, of the activities X_i, TX_i, FX_i for each $0 < i \leq m$, of the activities C_j for each distinct clause j of Φ, and of a final state Sat. Moreover, we define $\mathtt{IN}(Sat) = n$ (where n is the number of clauses contained in Φ), and $\mathtt{IN}(a) = 1$ for any other activity $a \neq a_0$.

The set of precedences E is defined as follows.

- For each X_i, (X_i, TX_i) and (X_i, FX_i) are in E, with constraints $\texttt{OUT}_{min}(X_i) = \texttt{OUT}_{max}(X_i) = 1$. Thus, each time the activity X_i is executed, it is required to make a choice between its possible successors; note that in our encoding, TX_i means that X_i is **true**, while FX_i means that X_i is **false**. Finally an arc (a_0, X_i) is in E, and constraints $\texttt{OUT}_{min}(a) = \texttt{OUT}_{max}(a) = m$ are added.
- For each C_j, we have that (C_j, Sat) is in E, with constraints $\texttt{OUT}_{min}(Sat) = \texttt{OUT}_{max}(Sat) = \mathit{1}$. Moreover, we have $(TX_i, C_j) \in E$ in the case X_j appears in the clause C_j, while we have $(FX_i, C_j) \in E$ in the case X_i appears negated in the clause C_j. Finally, for each node $a \in \{TX_i, FX_i\}$, $\texttt{OUT}_{min}(a) = \texttt{OUT}_{max}(a) = \mathit{OutDegree}(a)$.

Now, assume Φ is satisfiable. Then, it is possible to choose the successor of each X_i, in a way that all the activities C_j can be executed. Hence, the activity Sat will be eventually reached. On the other side, if there exists a path leading to Sat, it can be easily mapped into a satisfying assignment for Φ.

(ii) Assume that, for each $a \in A$, $\texttt{OUT}_{max}(a) = \mathit{OutDegree}(a)$. It is easy to see that for the problem of deciding whether a given activity can be executed, we can assume, w.l.o.g., that $\texttt{OUT}_{min}(a) = \mathit{OutDegree}(a)$, too. Indeed, it is always convenient to activate all the outgoing arcs in a: if an activity cannot be executed with the activation of all the outgoing arcs in a, then it cannot be executed in any other type of execution. Then, the problem can be solved in polynomial time by applying the function δ_{ws}, that it is actually a deterministic function. For the hardness, we consider the AND/OR GRAPH ACCESSIBILITY problem [12]: we are given an and/or graph $G = (V, E)$ (i.e., a directed graph such that each vertex is assigned either a $\vee$ or a $\wedge$ label), and two vertices s and t; the problem is to decide whether t can be reached from s. A vertex labelled by $\vee$ can be reached if and only if at least one of its predecessors are reached, whereas a vertex labelled by $\wedge$ can be reached if and only if all its predecessors are reached. Notice that vertices without predecessors can be reached by default. We construct a workflow schema $\mathcal{WS}(G)$ by adding a starting activity connected to all the vertices without predecessors and s as well. For a vertex a labelled by $\vee$ in G we fix $IN(a) = 1$, whereas $IN(a) = \mathit{InDegree}(a)$ is a is labelled by $\wedge$ in G. The only final activity is t. It is worth noticing that t is reachable from s in G if and only if $\mathcal{WS}(G)$ admits an execution reaching the final state. $\blacksquare$

III. Problem Description and Complexity Results

In this section we are interested in formalizing and analyzing the complexity of some interesting reasoning tasks, whose usage can help the system administrator in predicting the workflow evolution. The analysis is carried out on the basis of the formal model of workflow schema and execution provided so far.

Let us first of all address the following problem: assume that an execution has arrived at a given point and, before letting it proceed, the administrator wants to know whether it will lead to a successful termination or not. The problem can be formalized as follows.

Let $\mathcal{WS}$ be a workflow schema, and $e = [S_0, \ldots, S_h]$ be a partial execution on $\mathcal{WS}$. A successful execution $e' \in \mathcal{S}_{ws}$ whose first $h+1$ steps are $[S_0, ..., S_h]$ is said to be a *successful extension* of e, and is denoted by $e \rightsquigarrow e'$.

Definition III.1: **(Successful Termination Prediction - STP)**
Let $\mathcal{WS}$ be a workflow schema, and e be a partial execution. Then, the STP problem for e is deciding whether there exists a successful extension of e. $\square$
We point out that the STP problem appears in [18] in the form of guaranteed executions in the more complex setting of transactional processes. We next show that STP is intractable.

Proposition III.2: Let $\mathcal{WS}$ be a workflow schema and $e = [S_0, ..., S_h]$ be an execution that is not terminating. Then, the STP *problem for e is* **NP***-complete.*

Proof: Membership derives from the fact that we can guess an execution e' such that e' is successful, and check (in polynomial time in the size of $\mathcal{WS}$) whether its first $h+1$ steps are $[S_0, ..., S_h]$. For the hardness, let $h = 0$, and w.l.o.g. assume in the workflow schema the initial activity does not correspond to a final one. Thus, we can consider the problem of deciding whether there exists a successful execution e', with $[S_0] \rightsquigarrow e'$. The hardness follows, from Proposition II.7. $\blacksquare$

The above discussion sheds some light in the intrinsic difficulty of solving such problems "statically". Reasoning about the structure seems not to be a valuable approach; hence, we are motivated in using data mining techniques, that can be directly applied to a set of instances, collected in the log of the workflow system. Indeed, one could be interested in a more pragmatic version of the STP problem: *given the history of past executions, does the current execution have a chance to eventually succeed?* We formalize the problem next.

Definition III.3: **(Frequent Successful Termination Prediction - FSTP)**
Let $\mathcal{WS}$ be a workflow schema, $Se = \{e_1, \ldots, e_n\}$ be a set of successful executions on $\mathcal{WS}$, each one equipped with a frequency $f_i = f_i(e_i) \in \mathbf{N}$, *minFreq* be a natural number, and e be a non-terminating execution on $\mathcal{WS}$. Then, the problem `FSTP` for e w.r.t. Se and $minFreq$ is deciding whether$\sum_{\{i|e_i \in Se, e \rightsquigarrow e_i\}} f_i \geq$ *minFreq*, i.e., whether there the number of successful extensions of e in Se is greater than or equal to *minFreq*. $\square$

As a matter of fact, the `STP` problem is equivalent to an instance of the `FSTP` problem.

Proposition III.4: Let a workflow schema $\mathcal{WS}$ and a non-terminating execution e be given in input. Then, the `STP` *problem is equivalent to the* `FSTP` *problem for e w.r.t. $\mathcal{S}_{ws}$, where minFreq $= 1$ and $f_i(e_i) = 1$, for each execution $e_i \in \mathcal{S}_{ws}$.*

Proof: By definition, under the assumptions of the statement, the problem `FSTP` corresponds to check whether $|\{e_i | e_i \in \mathcal{S}_{ws}, e \rightsquigarrow e_i\}| \geq 1$. This happens if and only if there exists $e_i \in \mathcal{S}_{ws}$ such that $e \rightsquigarrow e_i$. $\blacksquare$

The complexity of the `FSTP` problem mainly depends on the number of executions in Se. If this number is low (e.g., polynomially bounded by the size of $\mathcal{WS}$) then the problem can be effectively solved, as the following proposition shows. Nevertheless, when the size of Se grows, one cannot expect to reduce the complexity by finding some succinct representation of Se: also in this case a time exponential in the size of $\mathcal{WS}$ cannot be avoided unless **P**=**NP**.

Proposition III.5: Let $\mathcal{WS}$ be a workflow schema and $e = [S_0, ..., S_h]$ be an execution that is not terminating. Then, given a set $Se = \{e_1, \ldots, e_n\}$ of terminating executions on $\mathcal{WS}$, each one equipped with a frequency $f_i = f_i(e_i) \in \mathbf{N}$, and a natural number minFreq,

i) the `FSTP` *problem for e w.r.t. Se and minFreq can be solved in time polynomial in the size of $\mathcal{WS}$ and Se, but*

ii) the succinct `FSTP` *problem, in which Se is represented by a data structure with size polynomially bound in the size of $\mathcal{WS}$, is* **NP***-complete.*

Proof:

i) Observe that, for each e_i checking whether $e \rightsquigarrow e_i$ can be done in polynomial time in the size of $\mathcal{WS}$. Hence, a naive polynomial algorithm (in the size of $\mathcal{WS}$ and Se) consists in summing the frequency associated to each execution $e_i \in Se$ with $e \rightsquigarrow e_i$, and hence checking whether the corresponding sum is greater than *minFreq*.

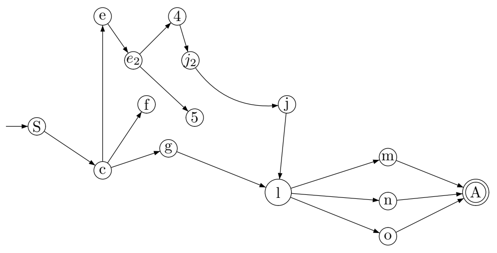


Fig. 3. An instance of the workflow schema of Figure 1.

ii) Membership is trivial. For the hardness, observe that the `FSTP` problem in the statement of Proposition III.4 can be obviously formulated in a succinct way: indeed Se needs not to be explicitly stored. But the succinct problem is nothing but the `STP` problem which is **NP**-complete by Proposition III.2. ■

An appealing way for solving the `FSTP` problem is to use specialized data mining techniques for graphs. To this end, we first need to characterize workflow executions in terms of connected subgraphs of the workflow schema.

Definition III.6: Let $\mathcal{WS} = \langle A, E, a_0, F\rangle$ be a workflow schema and $e = [S_0, ..., S_k]$ be an execution. Then, the *instance* associated to e is the graph $I_e = \langle A_e, E_e, a_0, F_e\rangle$, where $A_e = \cup_{t=1,k} \mathit{Executed}_t$, $E_e = \{(a,b) | (a,b) \in \cup_{t=1,k} \mathit{Marked}_t, b \in A_e\}$ and $F_e = A_e \cap F$. In case e is a successful execution, then I_e is said *successful instance.* □

An instance for the workflows schema presented in Example II.2 is shown in Figure 3.

In the following, given a workflow schema $\mathcal{WS}$, we denote by $2^{\mathcal{WS}}$ the family of all the subgraphs of the graph $\langle A, E\rangle$, and by $\mathcal{I}(\mathcal{WS})$ the set of all instances.

Observe that, while deciding whether a subgraph is an instance is polynomial, instead deciding whether there exists a successful instance is not tractable.

Proposition III.7: Let $\mathcal{WS} = \langle A, E, a_0, F\rangle$ be a workflow schema. Then,

i) given a subgraph I of $\mathcal{WS}$, deciding whether I is an instance of $\mathcal{WS}$ can be done in polynomial time in the size of E, and

ii) deciding whether $\mathcal{WS}$ admits a successful instance is **NP***-complete.*

Proof:

i) We construct the sequence of states corresponding to I by traversing the subgraph I starting from the initial node and by applying in a constructive way the function δ using all arcs in I as marked. Clearly the algorithm is polynomial in the size of E.

ii) Membership is trivial; for the hardness, observe that there exists a successful instance, say I_e, if and only if there exists a successful execution, e, to which I_e is associated. The hardness follows from Proposition II.7. ■

We now introduce the notion of pattern that will be crucial in the process of data mining. To provide a more uniform notation, given a graph p (e.g., a workflow schema or a pattern) and a node a of p, we denote by $\mathit{InDegree}_p(a)$ (resp. $\mathit{OutDegree}_p(a)$) the number of ingoing (resp. outgoing) edges of a.

Definition III.8: Let $\mathcal{WS}$ be a workflow schema, and $\mathcal{F}$ be a multiset of instances. Then, a graph $p = \langle A_p, E_p \rangle \in 2^{\mathcal{WS}}$ is an *$\mathcal{F}$-pattern* (cf. $\mathcal{F} \models p$) if there exists $I = \langle A_I, E_I \rangle \in \mathcal{F}$ such that $A_p \subseteq A_I$ and p is the subgraph of I induced by the nodes in A_p. In the case $\mathcal{F} = \mathcal{I}(\mathcal{WS})$, the subgraph is simply said to be a *pattern*. Moreover, if A_p contains some final activity in $\mathcal{WS}$, then p is said to be *successful*. □

Roughly speaking, an $\mathcal{F}$-pattern is a subgraph of a workflow instance in $\mathcal{F}$. Thus, we are using the notion of pattern with the meaning which is being adopted by the data mining community in several other applicative domains. For instance, patterns are subtrees in the mining of frequent trees (see, e.g., [31]), subsequences in the mining of sequences (see, e.g., [3]), and so on. Hence, Definition III.8 is inserted into a data mining context and is not related at all with the notion of pattern used in software engineering contexts (and recently by van der Aalst [25] for supporting workflow modelling[2]).

Let us now consider the following problem of data mining on graphs that consists in discovering patterns which frequently arise.

Definition III.9: **(Frequent pattern mining - FPM)**
Let $\mathcal{WS}$ be a workflow schema, $\mathcal{F}$ be a multiset of instances and, *minSupp* be a real number with $0 \leq \mathit{minSupp} \leq 1$. Then, the problem `FPM` for $\mathcal{F}$ consists in finding all the *frequent* $\mathcal{F}$-patterns, i.e. all the $\mathcal{F}$-patterns for which $\mathit{supp}(p) \geq \mathit{minSupp}$, where the support $\mathit{supp}(p)$ is defined as $|\{I|\{I\} \models p \wedge I \in \mathcal{F}\}|/|\mathcal{F}|$. □

Frequent patterns can be used for heuristically solving the problem `FSTP`, that is, for deciding whether a sequence of states will very likely (e.g., with a reasonable support) lead to a successful (or unsuccessful) termination. In fact, given a partial execution e and

[2]See also *http://tmitwww.tm.tue.nl/research/patterns/*.

a support *minSupp*, e will very likely lead to a successful end if there exists at least one successful *frequent* $\mathcal{F}$-pattern containing all the activities executed in e. Then, in order to make our approach effective, we shall show, in the following section, some techniques for the efficient computation of frequent patterns of executions.

IV. Mining Connected Frequent Patterns

In this section we present two algorithms for mining connected frequent patterns (i.e., subgraphs) in workflow instances. Let us assume that a workflow schema $\mathcal{WS} = \langle A, E, a_0, F \rangle$ and a multiset of instances $\mathcal{F} = \{I_1, ..., I_n\}$ are given. Then, a naive algorithm for mining frequent patterns can generate directly the subgraphs, and test in polynomial time whether they are instances of $\mathcal{WS}$. Our approach is based on the idea of reducing the number of patterns to generate, by only considering $\mathcal{F}$-patterns that are not only connected but also *deterministically closed*. This restriction is formalized next.

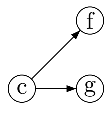
Definition IV.1: Given a graph $p = \langle A_p, E_p \rangle \in 2^{\mathcal{WS}}$, the deterministic closure of p (cf. *ws-closure*(p)) is inductively defined as the graph $p' = \langle A_{p'}, E_{p'} \rangle$ such that: (i) $A_p \subseteq A_{p'}$, and $E_p \subseteq E_{p'}$ (basis of induction), (ii) $a \in A_{p'}$ is an *and-join* implies that for each $(b, a) \in E$, $(b, a) \in E_{p'}$ and $b \in A_{p'}$, (iii) $a \in A_{p'}$ is a *deterministic fork* implies that for each $(a, b) \in E$ with b *or-join*,[3] $(a, b) \in E_{p'}$ and $b \in A_{p'}$. Moreover, a graph p such that $p =$ *ws-closure*(p) is said *ws-closed*. $\square$

Intuitively, the above definition provides a way for extending a subgraph p, by including all the activities that are enforced to be executed with some activity in A_p, by means of the constraints issued over $\mathcal{WS}$. And, in fact, the definition can be used to introduce a notion of pattern which only depends on the structure of the workflow schema, rather than on the instances $\mathcal{F}$ or $\mathcal{I}(\mathcal{WS})$. The need of this weaker notion will be clear in a while.

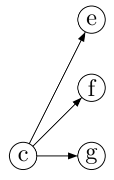
Definition IV.2: A *weak pattern*, or simply w-pattern, is a *ws-closed* graph $p \in 2^{\mathcal{WS}}$, such that for each node a, $|\{(a, b) | (a, b) \in E_p\}| \leq \texttt{OUT}_{max}(a)$. $\square$

Example IV.3: Consider the workflow graph of Figure 1, and the following subgraphs.

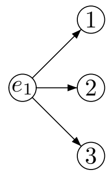
[3]Notice that relaxing the condition for b to be an *or-join* might lead to closures that cannot be traced by any execution. In fact, if b is an *and-join* synchronizing two mutually exclusive activities a and a' (e.g., that are activated by some *XOR-fork*), then a will never occur in the same execution with b.



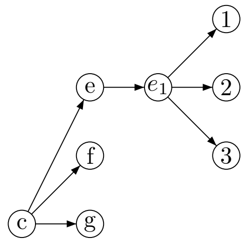
p_1



p_2



p_3



p_4

Then, p_1 is not a w-pattern, since $ws\text{-}closure(p_1) = p_2 \neq p_1$, and hence condition *ii)* of Definition IV.2 is not satisfied. Notice that p_2 is instead a w-pattern, since $ws\text{-}closure(p_1) = p_2$. Also, p_3 is not a w-pattern, since condition *ii)* of Definition IV.2 is not satisfied (indeed, $ws\text{-}closure(p_3) = p_4 \neq p_3$). Again, p_4 is a w-pattern, as $ws\text{-}closure(p_4) = p_4$. $\triangleleft$

The following proposition characterizes the complexity of recognition for the three notions of pattern; in particular, it states that testing whether a graph is a w-pattern is in **L** [8], i.e., it can be efficiently solved by a deterministic logarithmic-space bounded Turing machine. This efficiency is the result of the deterministic closure property and of the fact that w-patterns are defined over the schema, rather than on the instances.

Proposition IV.4: Let $p \in 2^{\mathcal{WS}}$. Then

1. deciding whether p is a pattern is **NP***-complete.*

2. given a multiset $\mathcal{F}$ of instances, deciding whether p is an $\mathcal{F}$-pattern can be done in polynomial time in the size of $\mathcal{F}$, but

3. deciding whether p is an $\mathcal{F}$-pattern is **NP***-complete, if $\mathcal{F}$ is succinct (i.e., if it can be represented by a data structure whose size is polynomially bounded by the size of $\mathcal{WS}$).*

4. deciding whether p is a w-pattern is in **L**.

Proof:

1. The problem is in **NP** as we can guess a subgraph I, by choosing for each node a the arcs to be activated, so that $\texttt{OUT}_{min}(a) \leq |\{(a,b)|(a,b) \in E_p\}| \leq \texttt{OUT}_{max}(a)$. Then, from Proposition III.7 we can check in polynomial time that I is an instance; finally, deciding whether p is a subgraph of I, can be done in polynomial time.

The hardness follows from Proposition II.7; indeed, we can assume p to be formed by a single activity, actually a final one (w.l.o.g. we can assume that $\mathcal{WS}$ has only one final activity, indeed we can add a new activity f to which all the final ones can be connected).

Thus, p is a pattern if and only if there exists a successful execution.

2. By definition of $\mathcal{F}$-pattern, we can simply test if p is a subgraph of any instance.

3. Membership is trivial, as we can check in polynomial time whether, for each instance I, $\{I\} \models p$ and $I \in \mathcal{F}$. For the hardness, observe that deciding whether p is a pattern corresponds to checking whether p is a $\mathcal{F}$-pattern with $\mathcal{F} = \mathcal{I}(\mathcal{WS})$.

4. The proof is given by defining a Turing machine that, given a workflow schema and a graph p encoded into the input tapes, can decide in deterministic logarithmic space whether p is a w-pattern. In fact, both $\mathcal{WS}$ and p can be encoded by fixing an arbitrary order on the activities. In order to verify properties i), ii), and iii) of Definition IV.2, we simply need to access each arc of p and $\mathcal{WS}$, and exploit two counters. Clearly, encoding such counters requires logarithmic space. ■

It turns out that the notion of weak pattern is the most appropriate from the computational point of view. Moreover, working with w-patterns is not an actual limitation, since the closure of each frequent $\mathcal{F}$-pattern is, in turn, a frequent w-pattern as well. Rather, it is a compact and efficient way for the mining of frequent patterns, as shown below.

Proposition IV.5: Let p be a frequent $\mathcal{F}$-pattern. Then i) ws-closure(p) is both a weak pattern and a frequent $\mathcal{F}$-pattern, and ii) each weak pattern $p' \subseteq p$ is a frequent $\mathcal{F}$-pattern.

Proof: In order to prove *i)*, we observe that for each $I \in \mathcal{F}$ s.t. $\{I\} \models p$, property $\{I\} \models$ *ws-closure*(p) holds. Indeed, if p is not a weak pattern, then according to Definition IV.1 there exists $a \in A_p$ such that one of the following cases occur:

- a is an *and-join* and there exists an edge $(b, a) \notin E$;
- a is a *deterministic fork* and there exists an edge $(a, c) \notin E_p$, with c *or-join*.

By Definitions II.4 and III.6, each instance $I \in \mathcal{F}$ containing a, must contain b, c and $(b, a), (a, c)$ as well. As a consequence, *ws-closure*(p) is frequent as well.

In order to prove *ii)*, it suffices to see that if there exists an unfrequent w-pattern $p' \subseteq p$, then it should contain at least either an unfrequent node a or an unfrequent edge (a, b). But this is a contradiction, since both a and (a, b) belong to p as well. ■

However, a weak pattern is not necessarily an $\mathcal{F}$-pattern nor even a pattern. As shown in the next sections, we shall use weak patterns in our mining algorithms to optimize the search space but we eventually check whether they are frequent $\mathcal{F}$-patterns.

A. Levelwise Search algorithm

The first algorithm we propose for mining frequent connected $\mathcal{F}$-patterns uses a levelwise theory. Roughly spraining, we incrementally construct frequent weak patterns, by starting from frequent "elementary" weak patterns (defined below), and by extending each frequent weak pattern using two basic operations: adding a frequent arc and merging with another frequent elementary weak pattern. As we shall show, the correctness follows from the results of Proposition IV.5, and from the observation that the space of all connected weak patterns constitutes a lower semi-lattice, with a precedence relation $\prec$, defined next.

The elementary weak patterns, from which we start the construction of frequent patterns, are obtained as the *ws-closures* of the single nodes.

Definition IV.6: Let $\mathcal{WS} = \langle A, E\rangle$ be a workflow schema. Then, for each $a \in A$, the graph *ws-closure*$(\langle\{a\},\{\}\rangle)$ is called an *elementary* weak pattern (cf. *ew*-pattern). □

Observe that the empty graph, denoted by $\perp$, is an elementary weak pattern. The set of all *ew*-patterns is denoted by EW. Moreover, let p be a weak pattern, then $\mathrm{EW_p}$ denotes the set of the elementary weak patterns contained in p. Note that given an *ew*-pattern e, $\mathrm{EW_e}$ is not necessarily a singleton, for it may contain other *ew*-patterns.

Given a set $E' \subseteq \mathrm{EW}$, $Compl(E') = \mathrm{EW} - \bigcup_{\mathrm{e}\in \mathrm{E'}} \mathrm{EW_e}$ contains all elementary patterns which are neither in E' nor contained in some element of E'.

We now introduce a precedence relation $\prec$ among connected weak patterns. First of all, let us denote by $E^{\subseteq}$ the subset of arcs in $\mathcal{WS}$ whose source is not a deterministic fork, i.e., $E^{\subseteq} = \{(a,b) \in E \mid \mathtt{OUT}_{min}(a) < OutDegree(a)\}$.

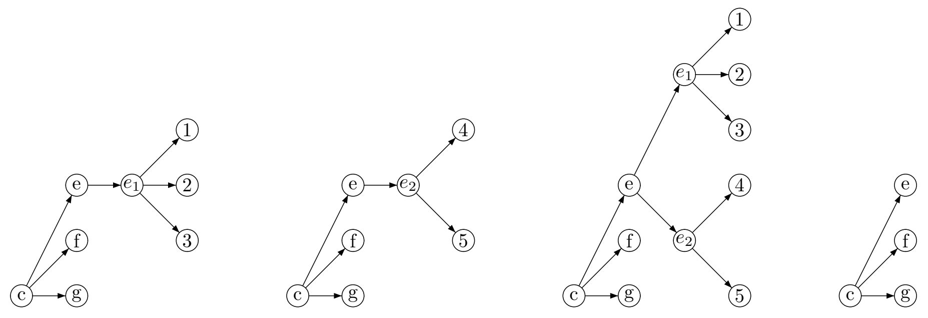
Definition IV.7: Given two connected w-patterns, say $p = \langle A_p, E_p\rangle$ and $p' = \langle A_{p'}, E_{p'}\rangle$, $p \prec p'$ if and only if:

a) $A_p = A_{p'}$ and $E_{p'} = E_p \cup \{(a,b)\}$, where $(a,b) \in E^{\subseteq} - E_p$ and $\mathtt{OUT}_{max}(a) > OutDegree_p(a)$ (i.e., p' can be obtained from p by adding an arc), or

b) there exists $p'' \in Compl(\mathrm{EW_p})$ such that $p' = p \cup p'' \cup X$, where X is either empty if p and p'' are connected or contains exactly an edge in $E^{\subseteq}$ with endpoints in p and p'' (i.e., p' is obtained from p by adding an elementary weak pattern and possibly a connecting arc).

Note that $\perp \prec e$, for each $e \in \mathrm{EW}$. □

Example IV.8: Consider again the workflow of Figure 1, and the following subgraphs:



p_1 p_2 p_3 p_4

The subgraphs p_1, p_2 and p_4 are elementary patterns: $p_1 = \textit{ws-closure}(\langle\{e_1\}, \emptyset\rangle)$, $p_2 = \textit{ws-closure}(\langle\{e_2\}, \emptyset\rangle)$ and $p_4 = \textit{ws-closure}(\langle\{c\}, \emptyset\rangle))$. p_3 is not an elementary pattern, as no node can generate it. Notice that $p_1 \prec p_3$ and $p_2 \prec p_3$. Finally, p_4 is contained in both p_1 and p_2 (and hence $p_4 \prec p_1$ and $p_4 \prec p_2$). $\lhd$

The following result states that all the connected weak patterns of a given workflow schema can be constructed by means of a chain over the $\prec$ relation.

Lemma IV.9: Let $p = \langle A_p, E_p\rangle$ be a connected w-pattern. Then, there exists a chain of connected w-patterns, such that $\perp \prec p_1 \prec ... \prec p_n = p$.

Proof: We prove this by induction on the size of p, $|p| = |A_p| + |E_p|$. The base case, i.e. $p \in \perp$, is trivial. For the case $p \notin \mathrm{EW}$, assume that for each weak pattern p', such that $|p'| < |p|$ there exists a chain $\perp \prec q_1 \prec ... \prec q_m = p'$. Two situations may occur:

1. $\exists (a, b) \in E_p \cap E^{\subseteq}$, such that (a, b) does not belong to any elementary pattern contained in p, and the graph p' obtained from p by deleting such an arc ($p' = \langle A_p, E_p - \{(a, b)\}\rangle$) is connected. In such a case, p' is a weak pattern, with $p' \prec p$. Hence, by induction, $\perp \prec q_1 \prec ... \prec q_m \prec p$. The theorem follows for $n = m$ and $p_1 = q_1, ..., p_n = q_n$.
2. for each $(a, b) \in E_p \cap E^{\subseteq}$, such that (a, b) does not belong to any elementary pattern contained in p, the graph $p' = \langle A_p, E_p - \{(a, b)\}\rangle$ is not connected. Two subcases can be further devised:

(a) there exists an elementary weak pattern $e \in \mathrm{EW_p}$, which is connected to the graph $p-e$ by means of exactly one arc in $E^{\subseteq}$; that is $e \in \textit{Compl}(\mathrm{EW_{p-e}})$, and, hence $(p-e) \prec p$, and the theorem follows by induction, otherwise

(b) elementary patterns are not connected by means of arcs in $E^{\subseteq}$. In this case, let $ep_0, ep_1, ..., ep_m$ be the elementary patterns contained in p, and $q = (p-ep_0) \cup ep_1 \cup ... \cup ep_m$

```
Input: A workflow Graph $\mathcal{WS}$, a set $\mathcal{F} = \{I_1, \ldots, I_N\}$ of instances of $\mathcal{WS}$.
Output: A set of frequent $\mathcal{F}$-patterns.
Method: Perform the following steps:
 1     $L_0 := \{e | e \in EW, e$ is frequent w.r.t. $\mathcal{F}\}$;
 2     $k := 0,\ R := L_0$;
 3     $FrequentArcs := \{(a,b) | (a,b) \in E^{\subseteq},\ \langle \{a,b\}, \{(a,b)\} \rangle$ is frequent w.r.t. $\mathcal{F}\}$;
 4     $E^{\subseteq}_f := E^{\subseteq} \cap FrequentArcs$;
 5     repeat
 6       $U := \emptyset$;
 7       forall $p \in L_k$ do begin
 8         $U := U \cup addFrequentArc(p)$;      //see (a) in Lemma IV.10
 9         forall $e \in Compl(EW_p) \cap L_0$ do
10           $U := U \cup addFrequentEWPattern(p, e)$;      //see (b) Lemma IV.10
11       end
12       $L_{k+1} := \{p | p \in U, p$ is frequent w.r.t. $\mathcal{F}\}$;      //see (c) in Lemma IV.10
13       $R := R \cup L_{k+1}$;
14     until $L_{k+1} = \emptyset$;
15     return $R$;
Function $addFrequentEWPattern(p = \langle A_p, E_p \rangle, e = \langle A_e, E_e \rangle)$: w-pattern;
  $p' := \langle A_p \cup A_e, E_p \cup E_e \rangle$;
  if $p'$ is connected, then return $p'$ else return $addFrequentConnection(p', p, e)$;
Function $addFrequentConnection(p' = \langle A_{p'}, E_{p'} \rangle, p = \langle A_p, E_p \rangle, e = \langle A_e, E_e \rangle)$: w-pattern;
  $S := \emptyset$
  forall frequent $(a,b) \in E^{\subseteq}_f - E_p$ s.t. $(a \in A_p,\ b \in A_e) \vee (a \in A_e,\ b \in A_p)$ do begin
    $q := \langle A_{p'}, E_{p'} \cup (a,b) \rangle$;
    if $\mathcal{WS} \models q$ then $S := S \cup \{q\}$;
  end
  return $S$
Function $addFrequentArc(p = \langle A_p, E_p \rangle)$: pattern;
  $S := \emptyset$
  forall frequent $(a,b) \in E^{\subseteq}_f - E_p$ s.t. $a \in A_p$, $b \in A_p$ do begin
    $p' := \langle A_p, E_p \cup (a,b) \rangle$
    if $\mathcal{WS} \models p'$ then $S := S \cup \{p'\}$;
  end
  return $S$
```

Fig. 4. **Algorithm** w-find($\mathcal{F}$,$\mathcal{WS}$)

the weak pattern obtained from p by deleting edges and nodes in ep_0 which do no occur in any other ep_i, with $0 < i \leq m$. By construction $ep_0 \in Compl(q)$, and hence $q \prec p$. As in the other case, since $|q| < |p|$, by induction there exists a chain of weak patterns $\perp \prec q_1 \prec \ldots \prec q_m = q$. ■

It turns out that the space of all connected weak patterns is a lower semi-lattice w.r.t. the precedence relation $\prec$. And, in fact, the algorithm w-find, reported in Figure 4, exploits an apriori-like exploration of this lower semi-lattice. Specifically, at each stage, the computation of L_{k+1} (steps 5-14) is carried out by extending any pattern p generated at the previous stage ($p \in L_k$), in two ways: (i) by adding frequent edges in $E^{\subseteq}$ (*addFrequentArc* function), and (ii) by adding an elementary weak pattern (*addEWFrequentPattern* function). Each pattern p', generated by the functions above, is an *admissible subgraph* of $\mathcal{WS}$ (cf. $\mathcal{WS} \models p'$), i.e., for each $a \in A_{p'}$, $OutDegree_{p'}(a) \leq \mathtt{OUT}_{max}(a)$. The properties of the

w-find algorithm are reported below.

Lemma IV.10: In the w-find algorithm, the following propositions hold:

(a) addFrequentArc adds to U connected patterns, which are not necessarily $\mathcal{F}$-patterns;

(b) addFrequentEWPattern add to U connected w-patterns, (not necessarily patterns);

(c) For each k, L_k contains only frequent connected $\mathcal{F}$-patterns.

Proof: We shall prove the above statements by induction on k. The proof is structured as follows. First, observe that L_0 contains a set of frequent connected $\mathcal{F}$-patterns. Indeed, by definition each elementary weak pattern is connected. Next, assuming that for a given $k \geq 0$, L_k contains only a set of frequent connected $\mathcal{F}$-patterns, observe that:

- Statement *(a)* holds. Indeed, since the input graph p is a connected $\mathcal{F}$-pattern, each graph p' obtained from p by adding a frequent arc (a, b) is connected as well. Notice now that, if $\mathcal{WS} \models p'$, then for each instance $\{I\} \models p$, the graph $I' = I \cup \{(a, b)\}$ is an admissible instance, i.e., $I' \in \mathcal{I}(\mathcal{WS})$. Indeed, for each execution e_I associated to I, an execution $e_{I'}$ can be obtained by adding (a, b) to $\delta Marked_{t+1}$ whenever $a \in Ready_t$. Moreover, $\{I'\} \models p'$ thus entailing that p' is a pattern. Finally, notice that, in principle, $\mathcal{F}$ may contain no I' satisfying the above condition.
- Statement *(b)* holds. Indeed, notice that *addEWFrequentPattern* returns any admissible connected subgraph p obtained from the union of a $\mathcal{F}$-pattern p' with an elementary pattern p''. If p is not a w-pattern, then either there exists an *and-join* $a \in A_p$, and $(b, a) \in E$ such that $(b, a) \notin E_p$, or there exists a *deterministic fork* $a \in A_p$ and $(a, b) \in E$, with b *or-join*, such that $(a, b) \notin E_p$. In both cases, there exists a node a and an edge e (containing a) such that $e \notin E_p$. But this cannot happen, since a is contained either in p' or in p'': indeed, since p' and p'' are w-patterns, e is contained in any of them, and consequently in E_p. Finally, notice that p'' is not necessarily a pattern, and consequently p is not necessarily a pattern as well.
- Statement *(c)* holds. Indeed, it follows from statements a and b that the set of candidate graphs U contains either connected patterns or connected $\mathcal{F}$-patterns. The consequence is trivial, by noticing that step 12 of the algorithm adds to L_{k+1} the only patterns in U which are frequent w.r.t. $\mathcal{F}$. ■

We next show that all the weak patterns are actually computed by the algorithm.

Proposition IV.11: **(Correctness)** *The algorithm of Figure 4 terminates and computes all and only the frequent connected weak patterns.*

Proof: The algorithm w-find computes all the elements in the lower semi-lattice induced by the operator $\prec$ over w-patterns. The correctness follows from Lemma IV.9, stating that any weak pattern is represented by a chain in this lower semi-lattice, and by the observation that we also prune the chains that will lead to unfrequent pattern. The latter is done by replacing the function *addEWPattern* in the definition of the relation $\prec$ with *addFrequentEWPattern*. ■

As conclusion of the presentation of w-find, we again remark that focusing on weak patterns is an efficient way for computing frequent patterns. In fact, Proposition IV.5 states that (i) for each frequent $\mathcal{F}$-pattern p', there exists a frequent weak pattern p (hence, computed by w-find) containing p' and (ii) any subgraph of any frequent weak pattern (again, computed by w-find) is a frequent $\mathcal{F}$-pattern as well.

B. Mining by connecting components

The algorithm w-find, proposed in Figure 4, performs a smart levelwise exploration of the lower semilattice, analyzed in Lemma IV.9. However, a different strategy can be exploited by observing that, in general, any connected pattern can be obtained by either composing two connected subgraphs, or by extending a subgraph by means of an edge.

Lemma IV.12: Let $p \in (2^{\mathcal{WS}} - \mathrm{EW})$ *be a connected* $\mathcal{F}$*-pattern. Then, there exist two* $\mathcal{F}$*-patterns* p_1 *and* p_2 *(not necessarily distinct) such that* $p = p_1 \cup p_2 \cup X$*, where* X *can be either the empty set or the graph* $\langle\{a,b\},\{(a,b)\}\rangle$ *with* $a \in p_1$ *and* $b \in p_2$*.*

Proof: Let p be a connected pattern not in EW. Then, due to Proposition IV.5, $q = \textit{ws-closure}(p)$ is a weak pattern. Due to Lemma IV.9, there exists a chain of connected w-patterns, such that $q_0 = \perp \prec q_1 \prec ... q_{n-1} \prec q_n = q$. Moreover, each q_{i+1} can be derived from q_i by either adding an edge in p_i or by connecting an elementary weak pattern to q_i. By denoting with Δq_i the graph that we compose with q_i, in order to derive q_{i+1}, we derive the following relationship: $\Delta q_0 \prec \Delta q_0 \cup \Delta q_1 \prec ... \prec \bigcup_{i=0,n-1} \Delta q_i = q$. W.l.o.g. we can assume that there exists $0 \leq j < n$ such that $p'_1 = \bigcup_{i=0,j} \Delta q_i$ and $p'_2 = \bigcup_{i=j+1,n-1} \Delta q_i$ are connected (it trivially holds for $j = n-1$). Then, by the definition of the relation of precedence $\prec$, we have that p'_1 and p'_2 are either connected or can be connected by means of an edge. The result follows by letting $p_1 = p'_1 \cap p$, and $p_2 = p'_2 \cap p$. ■

The above lemma, states that candidates can be generated by iteratively connecting components. In fact, we can generate a candidate at the n-th level of the lattice by merging two components at the j-th and (n-j) -th level, respectively. It is clear that in the worst case, for $j = n$-1, we degenerate to the levelwise search described in the previous subsection; nonetheless, in the best case, this approach converges in exponentially fewer iterations. Obviously, we also need an additional effort for identifying the components that can be merged. Roughly speaking, these components must be such that their boundaries can match, where the boundary of a graph in $2^{\mathcal{WS}}$ is the set of nodes that (according to a workflow schema $\mathcal{WS}$) admit either an input or an output edge.

In order to formalize the above intuitions, given a graph $p = \langle A_p, E_p \rangle \in 2^{\mathcal{WS}}$, we denote by $\texttt{INBORDER}(p) = \{a \in A_p \mid \mathit{InDegree}_p(a) < \mathit{InDegree}(a)\}$ the set of all the nodes in p which admit a further incoming edge, and by $\texttt{OUTBORDER}(p) = \{a \in A_p \mid \mathit{OutDegree}_p(a) < \texttt{OUT}_{max}(a)\}$ the set of all nodes in p which admit an outgoing edge. The sets $\texttt{INBORDER}(p)$ and $\texttt{OUTBORDER}(p)$ represent the input and output boundaries of p, i.e., the set of nodes inside p, which can reach (resp. can be reached by) other nodes outside p. Notice that, by construction, the input boundary of a w-pattern cannot contain *and-join* activities. Similarly, the output boundary of a w-pattern cannot contain *deterministic forks*.

The boundaries can be exploited to connect components. Since an arc connects the boundaries of two components, it suffices to concentrate on frequent arcs and iteratively generate new candidates by merging the frequent components whose boundaries are connected by means of those arcs.

Based on this ideas, we have developed an other algorithm (c-find), whose details are reported in Figure 5. It starts by computing frequent elementary patterns (step 1). Then, the core of the algorithm is a main loop (steps 3-24), in which the following operations are performed. For each node $a \in \mathcal{WS}$, the set $\texttt{INF}(a)$ (resp. $\texttt{OUTF}(a)$) of $\mathcal{F}$-patterns containing a in the input (resp. output) boundary is computed (steps 5-6). In the step 8 and 9, the variables *FA* and *PF* are used to store frequent arcs that may connect patterns, and candidates that may be generated by composing “compatible” patterns.

```
Input: A workflow Graph $\mathcal{WS} = (A, E)$, a set $\mathcal{F} = \{I_1, \ldots, I_N\}$ of instances of $\mathcal{WS}$.
Output: A set of frequent $\mathcal{F}$-patterns.
Method: Perform the following steps:
1   $R := \{\, e \mid e \in EW,\ e$ is frequent w.r.t. $\mathcal{F}\,\}$; $\Delta R := R$;
2   forall $(a, b) \in E$ do $connected\_by(a, b) = \emptyset$;
3   repeat
4     forall $a \in A$ do begin
5       $\mathtt{INF}(a) := \{\, p \in R \mid a \in \mathtt{INBORDER}(p)\,\}$, $\mathtt{INFP}(a) = \emptyset$;
6       $\mathtt{OUTF}(a) := \{\, p \in R \mid a \in \mathtt{OUTBORDER}(p)\,\}$, $\mathtt{OUTFP}(a) = \emptyset$;
7     end
8     $FA := \{\, (a, b) \mid (a, b)$ is frequent w.r.t. $\mathcal{F}, \mathtt{OUTF}(a) \neq \emptyset, \mathtt{INF}(b) \neq \emptyset\,\}$
9     $FP := \{\, p \cup q \mid p \cap q \neq \emptyset, p \in R,\ q \in \Delta R, \mathcal{WS} \models p \cup q\}$;
10    forall $(a, b) \in FA$ do
11      forall $p_1 \in \mathtt{OUTF}(a), p_2 \in \mathtt{INF}(b)$ s.t. $(a, b) \notin p_1 \cup p_2$ and $(p_1, p_2) \notin connected\_by(a, b)$ do begin
12        $q := p_1 \cup p_2 \cup \{(a, b)\}$;
13        if $\mathcal{WS} \models q$ then begin
14          $FP := FP \cup \{q\}$;
15          $\mathtt{INBORDER}(q) := ComputeInBorder(b, p_1, p_2)$;
16          $\mathtt{OUTBORDER}(q) := ComputeOutBorder(a, p_1, p_2)$;
17          forall $a \in \mathtt{INBORDER}(q)$ do $\mathtt{INFP}(a) := \mathtt{INFP}(a) \cup \{q\}$;
18          forall $a \in \mathtt{OUTBORDER}(q)$ do $\mathtt{OUTFP}(a) := \mathtt{OUTFP}(a) \cup \{q\}$;
19          $connected\_by(a, b) := connected\_by(a, b) \cup \{(p_1, p_2)\}$;
20        end
21      end
22    $\Delta R := \{\, p \in FP \mid p$ is frequent w.r.t. $\mathcal{F}\,\}$;
23    $R := R \cup \Delta R$;
24  until $\Delta R = \emptyset$;
25  return $R$;

Function $ComputeInBorder(b, p_1, p_2)$;
  if $|InDegree_{p_1 \cup p_2}(b)| < InDegree(b) - 1$ then $\mathtt{INBORDER} := \{b\}$ else $\mathtt{INBORDER} := \emptyset$;
  forall $c \in (\mathtt{INBORDER}(p_1) \cup \mathtt{INBORDER}(p_2)) - \{b\}$ do
      if $|InDegree_{p_1 \cup p_2}(c)| < InDegree(b)$ then $\mathtt{INBORDER} := \mathtt{INBORDER} \cup \{c\}$;
  return $\mathtt{INBORDER}$;

Function $ComputeOutBorder(a, p_1, p_2)$;
  if $|OutDegree_{p_1 \cup p_2}(a)| < \mathtt{OUT}_{max}(a) - 1$ then $\mathtt{OUTBORDER} := \{a\}$ else $\mathtt{OUTBORDER} := \emptyset$;
  forall $c \in (\mathtt{OUTBORDER}(p_1) \cup \mathtt{OUTBORDER}(p_2)) - \{a\}$ do
      if $(|OutDegree_{p_1 \cup p_2}(c)| < \mathtt{OUT}_{max}(b))$ then $\mathtt{OUTBORDER} := \mathtt{OUTBORDER} \cup \{c\}$;
  return $\mathtt{OUTBORDER}$;
```

Fig. 5. **Algorithm** c-find($\mathcal{F}$,$\mathcal{WS}$)

Then, boundaries are recomputed for the new candidate $\mathcal{F}$-patterns (steps 11-21), and frequent $\mathcal{F}$-patterns are detected by computing the frequency of each candidate (step 22). Notice that boundaries for candidate $\mathcal{F}$-patterns can be incrementally computed by extending the boundaries of the connected components, and that new candidates can be generated also by merging $\mathcal{F}$-patterns sharing some nodes. The algorithm terminates when no further candidates can be found, i.e., when the computed patterns have empty input-output boundaries.

Theorem IV.13: **(Correctness)** *The c-find algorithm terminates and computes all and only the frequent connected weak patterns.*

Proof: Correctness trivially holds by step 22 of the algorithm: indeed, no pattern is included in R unless it is not a $\mathcal{F}$-pattern. As for completeness, let p be a $\mathcal{F}$-pattern.

We prove by induction on $|p|$ that $p \in R$. The case $p \in \text{EW}$ statement trivially holds as a consequence of step 1 of the algorithm. Let us consider the case $|p| > 1$. By lemma IV.12, there exist p_1, p_2 such that $p = p_1 \cup p_2 \cup X$, where X can be the empty set or an edge connecting p_1 and p_2. By induction, both p_1 and p_2 are in R. Let us assume w.l.o.g. that p_1 is added to R at iteration k_1 and that p_2 is added to R at iteration $k_2 > k_1$. Two situations may occur.

1. $p_1 \cap p_2 \neq \emptyset$ and $p = p_1 \cup p_2$. In such a case, p is added to PF at the iteration $k_2 + 1$, and consequently it is added to R.
2. $p = p_1 \cup p_2 \cup \{(a, b)\}$. Again, without loss of generality we can assume that $a \in \mathtt{OUTBORDER}(p_1)$ and $b \in \mathtt{INBORDER}(p_2)$. In such a case, by step 8 of the algorithm, $(a, b) \in FA$ at iteration $k_2 + 1$. By steps 11 and 12 of the algorithm, $p \in PF$ at iteration $k_2 + 1$, and hence $p \in R$.

Observe finally that each candidate pattern p is considered at most k times, where k is the number of connected patterns contained in p. Since the number of candidate patterns is finite, the algorithm must terminate. ■

In comparing the performance of the c-find algorithm with the w-find algorithm proposed in the previous section, it is interesting to notice that the c-find algorithm can generate more candidates than w-find, but in general reaches convergence more quickly (number of iterations).

Proposition IV.14: Let $\mathcal{C}$ be the set of candidate patterns generated by c-find and let N_c be the steps required for its execution. Let $\mathcal{W}$ the set of candidate patterns generated by w-find and let N_w the steps required for its execution. Then, $\mathcal{W} \subseteq \mathcal{C}$, and $N_c \leq N_w$.

Proof: It is easy to see that c-find considers all the candidate patterns considered by w-find. This is a trivial consequence of Lemma IV.12, and of the observation that c-find degenerates in w-find each time it considers elementary patterns in R. This also entails $N_c \leq N_w$. However, in general, $\mathcal{C} = \mathcal{W}$ does not hold. Indeed, let us consider the situation in which there are four patterns p_1, p_2, p_3 and p_4, and the patterns $p_1 \cup p_2$ and $p_2 \cup p_4$ are frequent, but the pattern $p_1 \cup p_3$ is not. Assume also that patterns $p_1 \cup p_2$ and $p_3 \cup p_4$ are connected by means of an edge (a, b). In such a case, *c-find* would generate the (unfrequent) candidate pattern $p_1 \cup p_2 \cup p_3 \cup p_4 \cup \{(a, b)\}$, but *w-find* would not. ■

As a consequence of the above statements, the two algorithms can be considered as viable alternatives, and a preference can be carried out only by considering the particular structure of the workflow schema that had generated the instances. The next section will also provide a discussion and a comparison between the two algorithms.

V. Experiments and Discussion

In this section we study the behavior of the algorithms w-find and c-find, by evaluating both their performance and their scalability. As shown in the previous section, the algorithms are sound and complete w.r.t. the set of frequent w-patterns. Nevertheless, in principle the number of candidate w-patterns generated could be prohibitively high, thus making the algorithms unfeasible on complex workflow schemas. Moreover, we also compare the performance of our implementations w.r.t. several existing techniques for computing frequent itemsets adapted to the particular applicative domain.

In our experiments, we mainly use synthetic data. Synthetic data generation can be tuned according to: i) the size of $\mathcal{WS}$, ii) the size of $\mathcal{F}$, iii) the size $|L|$ of the frequent weak patterns in $\mathcal{F}$, and iv) the probability $p_{\subseteq}$ of choosing a $E^{\subseteq}$-arc. The ideas adopted in building the generator for synthetic data are essentially inspired by [2].

A. Performance of w-find

In a first set of experiments, we tested the w-find algorithm by first considering some fixed workflow schemas, and generating synthesized workflow instances. In particular, the nondeterministic choices in the executions are performed according to a binomial distribution with mean $p_{\subseteq}$. Frequent instances are forced into the system by replicating some instances (in which some variations were randomly performed) according to $|L|$. Figure 6 reports on the left the number of operations (matching of a pattern with an instance), for increasing values of $|\mathcal{F}|$. The figure shows that the algorithm scales linearly in the size of the input (for different supports).

In a second set of experiments, we randomly generate the workflow schemas to test the efficiency of the approach w.r.t. the structure of the workflow. To this aim, we fix $|\mathcal{F}|$ and generate workflow instances according to the randomly generated schema. The actual number of nodes and arcs is chosen by picking from a Poisson distribution with fixed mean

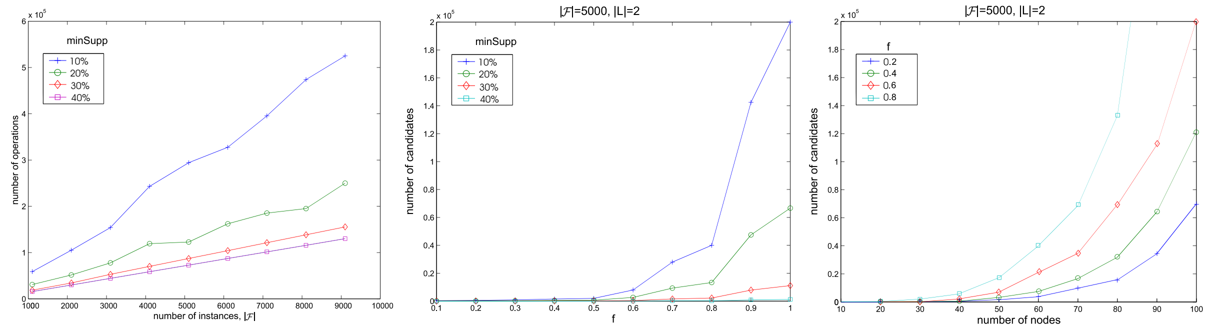


Fig. 6. w-find performance. Left: Number of operations w.r.t. $|\mathcal{F}|$. Center: Number of candidates w.r.t. f, for different *minSupp* values. Right: Number of candidates w.r.t. number of nodes, for randomly generated workflow schemas.

value. In order to evaluate the contribution of the complexity of workflow schemas, we exploit the factor $f = \frac{|E^{\subseteq}|}{|E|}$, which represents the degree of potential nondeterminism within a workflow schema. Intuitively, workflow schemas exhibiting $f \simeq 0$ produce instances with a small number of candidate w-patterns. Conversely, workflow schemas exhibiting $f \simeq 1$ produce instances with a large number of candidate w-patterns. Figure 6 shows on the center the behavior of w-find when f ranges between 0 (no nondeterminsm) and 1 (full nondeterminism), for different values of *minSupp* values. It is interesting to observe that even for significantly higher values of f (real workflow schema are expected to have a degree of non-determinism less then 0.5), the smart way of searching the search space reduces drastically the number of candidates being generated.

Finally, on the right, we report the number of candidates at the varying of the number of nodes, for different values of f. It is worth noting the exponential behavior, due to the combinatorial explosion of the search space.

B. Comparing w-find and Apriori

We consider an implementation of the *Apriori* algorithm which only computes frequent itemsets of edges in $E^{\subseteq}$. Such an approach is significant for analyzing the performance overhead suffered by traditional frequent-pattern mining methods, which typically can be easily adapted to mine workflow instances but are not tuned to take into account domain information about the workflow schema. We perform several experiments comparing the performance of the *Apriori* approach with the ones of w-find on increasing values of $|\mathcal{F}|$ and *minSupp*. For a dataset of instances generated as said before w.r.t. the workflow

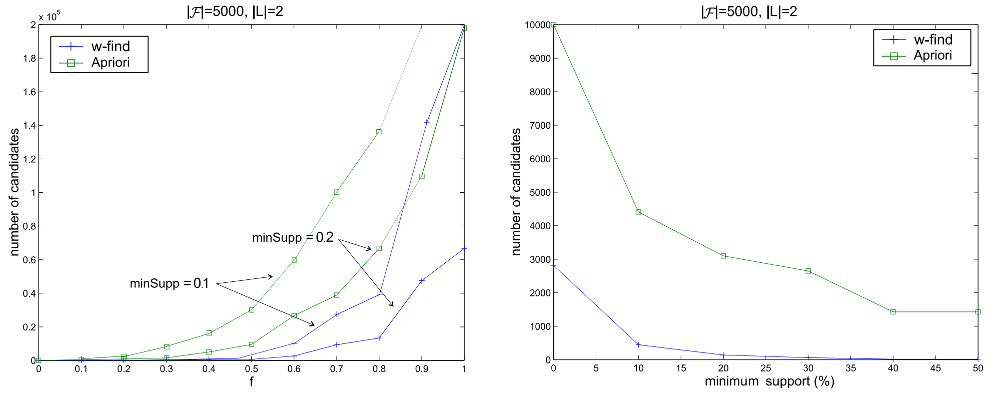


Fig. 7. w-find vs Apriori. Left: Number of candidates w.r.t. f. Right: Number of candidates for different *minSupp* values.

schema of Figure 1, the comparison is reported in Figure 7 (on the right).

As expected, w-find outperforms *Apriori* by an order of magnitude. This is mainly due to the fact that, contrarily to w-find, in the *Apriori* implementation arcs in $E^{\subseteq}$ are combined without taking into account the information provided by the workflow schema.

Figure 7 shows instead on the left the behavior of both *Apriori* and w-find when f ranges between 0 (no nondeterminsm) and 1 (full nondeterminism). Again, *Apriori* is outperformed by w-find. Notice that, for small values of f, both the algorithms produce a small number of candidates; however, in this situation, w-find still performs significantly better than *Apriori* for small *minSupp* values (e.g., 0.1). In fact, for lower values of of *minSupp*, the number of candidates increments significantly, and, hence, the focused strategy of w-find leads to a significant gain. However, we point out that the *adaptation* of *Apriori* tested here might be a viable solution for the mining of "nearly deterministic" workflows, if we are, moreover, interested in very frequent patterns (*minSupp*$>$ 0.2).

C. Comparing w-find with WARMR and Prefix-Span

A possible further approach to consider is the *WARMR* algorithm devised in [7] which allows an explicit formalization of domain knowledge (like, for example, the connectivity information provided by the workflow schema) which can be directly exploited by the algorithm. The setting file that we have used is reported in Figure 8 (predicates to be mined are on the right).

The results of the comparison are shown in Figure 9 (on the left), where we report the

```
warmode_key(instance(−I)).   rmode(startarc(+I, #N)).      type(startnode(pic, obj)).
talking(3).                  rmode(endnode(+I, #N).        type(endnode(pic, obj)).
use_packs(0).                rmode(andarc(+I, #S, #D)).    type(andarc(pic, obj, obj)).
minfreq(0.2).                rmode(xorarc(+I, #S, #D)).    type(xorarc(pic, obj, obj)).
typed_language(yes).         rmode(optarc(+I, #S, #D)).    type(optarc(pic, obj, obj)).
                             rmode(arc(+I, #S, #D)).       type(arc(pic, obj, obj)).
                             rmode(node(+I, #N)).          type(node(pic, obj)).
```

Fig. 8. The setting file used in the *WARMR* approach.

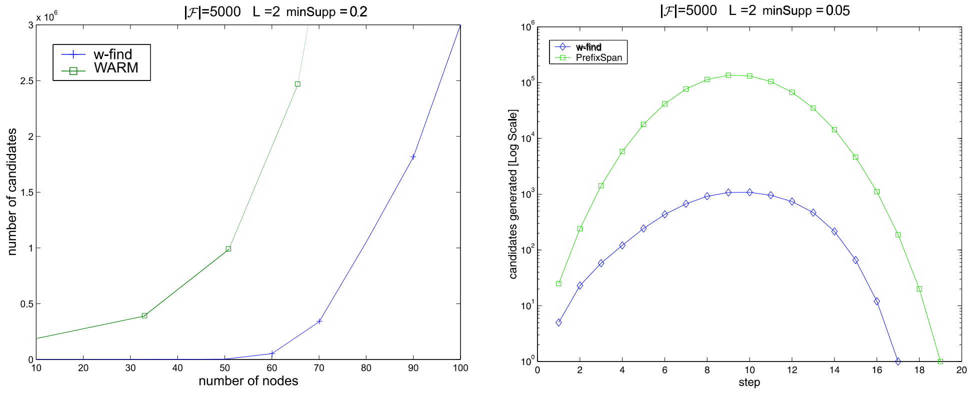


Fig. 9. Left: Comparison of *w*-find with *WARM*, over a fixed workflow schema. Right: Comparison of *w*-find with *PrefixSpan*.

correlation between the number of candidate patterns and the number of the nodes in the workflow schema, at the varying of f.

In the evaluation of the algorithm, we also have made some comparison w.r.t. methods for mining sequential pattern. However, a workflow is not a sequence; nonetheless, we can assume to represent each instance as a sequential pattern by considering the ordering of execution of each activity. For example, the instance reported in Figure 3 can be described by the sequence $s_1 = \langle S, c, (eg), (e_2l), (4, 5), j_2, j, l, (mno), A\rangle$, if we assume that each activity requires the same amount of time to be executed. Note that we grouped all the activities that we assume to be executed at the same time. Conversely, if the activity g requires more time, a possible ordering of executions associated to the same instance is $s_2 = \langle S, c, e, (e_2), (4, 5), j_2, j, g, l, (mno), A\rangle$. Thus, s_1 and s_2 are distinct sequences associated to the same instance. It follows that any sequential pattern algorithm can be used for extracting frequent instances, but it cannot be complete, in the sense that some frequent instances will not be mined since the sequences associated to the executions are possibly quite different (and hence infrequent). In our testing, we compared *w*-find with the *PrefixSpan* algorithm [16], but, in order to achieve a finer analysis, we assume each activity to require the same time to be executed; essentially, *PrefixSpan* has been applied

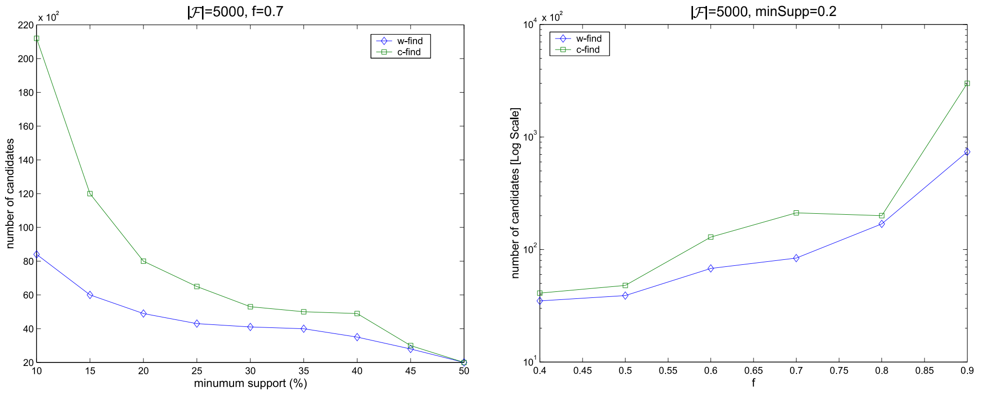


Fig. 10. Comparison of w-find with c-find. Left: Number of candidates for different *minSupp* values. Right: Number of candidates at the varying of the nondeterminism.

on the sequences constructed from each instance by performing a breadth-first search, starting from the initial activity.

The results are reported in Figure 9, where on the right we report the number of candidates generated at each stage of the computation, for a fixed workflow schema. Again, this experiment is significative only for understanding the advantage of a more focused method, and is not a comparison on "pure" sequences where *PrefixSpan* is expected to outperform both w-find and c-find. In fact, we can see that the more elaborate and domain dependent way of searching patterns in the lattice leads to a smaller number of patterns to be generated.

D. Comparing w-find and c-find

Finally, we report the experimental results of the comparisons between w-find and c-find. In a first set of experiments, we fixed a value of f (0.7), and generated 5000 random instances. In Figure 10, we report on the left the number of candidates generated over such instances at the varying of the minimum support. It is interesting to observe that w-find performs better than c-find especially for lower values of *minSupp*.

For a second set of experiments we fixed *minSupp*=0.2, and we made the comparison at the varying of f. This second set of experiments, whose results are reported in the right of Figure 10 confirmed the quality of w-find of generating fewer candidates, for every type of workflow (regardless of the degree of non-determinism).

The factor that may instead lead to a preference of c-find is in the number of steps

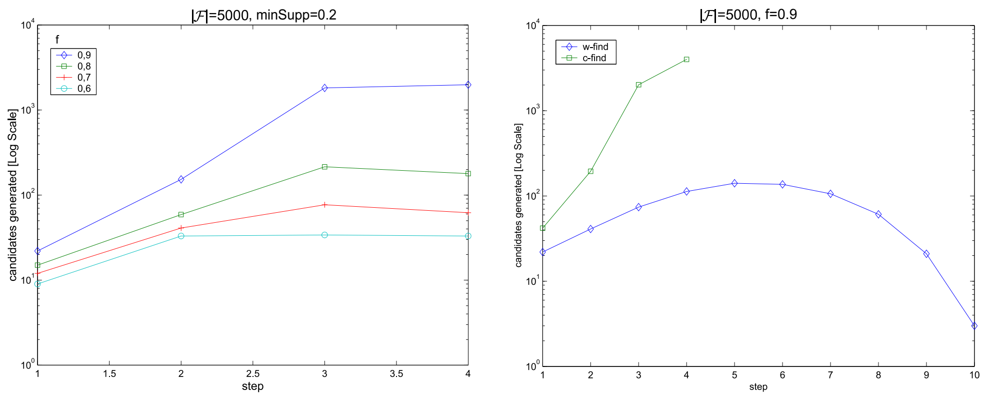


Fig. 11. Left: Number of candidates generated at the different steps. Right: Comparison of w-find with c-find.

performed. Let us consider Figure 11 which reports the number of candidates generated at the different steps of the algorithm (the scale is logarithmic). Here, the behavior of c-find is somehow dual to that of w-find (as reported on the right of Figure 9). Indeed, c-find at each successive step generates more candidates than in the previous one, and this leads the process to converge quickly. Conversely, w-find after a certain number of steps reduces dramatically the number of new frequent patterns discovered, and, hence, it requires more iterations. A more direct comparison is reported on the right of Figure 11, from which it is evident that the faster rate of convergence of c-find is payed with a bigger number of candidates generated. Since the number of steps coincides with the number of scans of the database, in the case of huge databases c-find may be convenient.

More generally, c-find is expected to exhibit better performance than w-find with *dense* workflow databases. More specifically, a set of workflow instances is dense if the number of expected frequent patterns is large w.r.t. the size of the workflow. If a database of instances exhibits this peculiarity, the number of candidate patterns to be generated is likely to be of the same order of magnitude as the number of frequent patterns (that is, the number of unfrequent patterns is small w.r.t. the set of frequent ones). In such a case, both c-find and w-find are expected to compute (almost) the same set of candidates. However, the look-ahead strategy of the c-find algorithm guarantees a faster convergence rate. Clearly, more efficient extensions could be devised to the proposed algorithms for dense databases, in order to avoid candidate generation (e.g., in the style of [10]).

VI. Conclusions

We have introduced the problem of mining frequent instances of workflow schemas, motivated by the aim of providing facilities for the system administrator to monitor the actual behavior of the workflow system in order to predict the "most probable" workflow executions. We have shown that the use of mining techniques is justified by the fact that even "simple" reachability problems are intractable.

We have proposed two novel graph mining algorithms specialized to deal with constraints imposed by the structures of workflow schemes and instances, and we have studied their properties both theoretically and experimentally, by showing that they represent an effective means of investigating some inherent properties of the executions of a given schema.

Following our approach, future research might develop more elaborated algorithms that are able to deal with more expressive modelling features, which have been not considered in our formal framework. For instance, a valuable on-going extension is dealing with supporting cyclic instances, by integrating our techniques with well known approaches for mining periodic patterns (see, e.g., [30]).

References

[1] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *Proc. 6th Int. Conf. on Extending Database Technology (EDBT'98)*, 1998, pp. 469–483.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.

[3] R.Agrawal and R.Srikant, "Mining Sequential Patterns," in *Proc. 11th Int. Conf. on Data Engineering (ICDE95)*, pp. 3–14, 1995.

[4] A. Bonner, "Workflow, transactions, and datalog," in *Proc. of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*, 1999, pp. 294–305.

[5] J.E. Cook and A.L. Wolf, "Automating process discovery through event-data analysis," in *Proc. 17th Int. Conf. on Software Engineering (ICSE'95)*, 1995, pp. 73–82.

[6] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan, "Logic based modeling and analysis of workflows," in *Proc. of the 17th ACM Symposium on Principles of Database Systems*, pp. 25–33, 19998.

[7] L. Dehaspe and H. Toivonen, "Discovery of Frequent DATALOG Patterns," *Data Mining and Knowledge Discovery*, vol. 3, no. 1, pp. 7–36, 1999.

[8] M.R. Garey and D.S. Johnson, *Computers and Intractability. A Guide to the Theory of* **NP***-completeness*, Freeman and Comp., NY, USA, 1979.

[9] D.Georgakopoulos, M. Hornick, and A. Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure," *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.

[10] J. Han, J. Pei, and Y. Yi, "Mining frequent patterns without candidate generation," in *Proc. Int. ACM Conf. on Management of Data (SIGMOD'00)*, pp. 1–12, 2000.

[11] A. Inokuchi, T. Washi, and H. Motoda, "An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data," in *Proc. 4th Conf. on Principles of Data Mining and Knowledge Discovery*, pp. 13–23, 2000.

[12] N.D. Jones and T. Laaser, "Complete problems for deterministic polynomial time," *Theoretical Computer Science*, vol. 3, pp. 105–117, 1977.

[13] G. Kappel, P. Lang, S. Rausch-Schott, and W. Retschitzagger, "Workflow management based on object, rules, and roles," *IEEE Data Engineering Bulletin*, vol. 18, no. 1, pp. 11–18, 1995.

[14] P. Koksal, S.N. Arpinar, and A. Dogac, "Workflow history management," *SIGMOD Recod*, vol. 27, no. 1, pp. 67–75, 1998.

[15] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Proc. IEEE Int. Conf. on Data Mining (ICDM'01)*, 2001, pp. 313–320.

[16] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns by prefix-projected growth," in *Proc. Int. Conf. on Data Engineering (ICDE'01)*, pp. 215–224, 2001.

[17] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-structure mining of frequent patterns in large databases," in *Proc. IEEE Int. Conf. on Data Mining (ICDM'01)*, 2001, pp. 441–448.

[18] H. Schuldt, G. Alonso, C. Beeri, and H. Schek, "Atomicity and isolation for transactional processes," *ACM Trans. Database Syst.*, vol. 27, no. 1, pp. 63–116, 2002.

[19] P. Senkul, M. Kifer, and I.H. Toroslu, "A logical framework for scheduling workflows under resource allocation constraints," in *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB'02)*, 2002, pp. 694–702.

[20] M.P. Sing, "Semantical considerations on workflows: An algebra for intertask dependencies," in *Proc. Int. Workshop on Database Programming Languages (DBPL'95*, 1995, pp. 6–8.

[21] W.M.P. van der Aalst, "The application of petri nets to worflow management," *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[22] W.M.P. van der Aalst, A. Hirnschall, and H.M.W. Verbeek, "An alternative way to analyze workflow graphs," in *Proc. 14th Int. Conf. on Advanced Information Systems Engineering*, 2002, pp. 534–552.

[23] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G.Schimm, and A.J.M.M. Weijters, "Workflow mining: A survey of issues and approaches," *Data and Knowledge Engineering*, vol. 47, no. 3, pp. 237–267, 2003.

[24] W.M.P. van der Aalst and K.M. van Hee, *Workflow Management: Models, Methods, and Systems*, MIT Press, 2002.

[25] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Advanced workflow patterns," in *Proc. 7th Int. Conf. on Cooperative Information Systems (CoopIS 2000)*, 2000, pp. 18–29.

[26] D. Wodtke and G. Weikum, "A formal foundation for distributed workflow execution based on state charts," in *Proc. 6th Int. Conf. on Database Theory (ICDT'97)*, 1997, pp. 230–246.

[27] D. Wodtkeand J. Weissenfels, G. Weikum, and A. Dittrich, "The Mentor project: Steps towards enterprise-wide workflow management," in *Proc. IEEE Inter. Conf. on Data Engineering (ICDE'96)*, 1996, pp. 556–565.

[28] X. Yan and J. Han, "gSpan: Graph-based substructure pattern pining," in *Proc. IEEE Int. Conf. on Data Mining (ICDM'02)*, 2001, An extended version appeared as UIUC-CS Tech. Report: R-2002-2296.

[29] X. Yan and J. Han, "CloseGraph: Mining closed frequent graph patterns," in *Proc. ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD'03)*, 2003, pp. 286–295.

[30] J.Yang, W.Wang, and Philip S. Yu, "Mining asynchronous periodic patterns in time series data," in *Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, pp. 275–279, 2000.

[31] M. Zaki, "Efficiently Mining Frequent Trees in a Forest," in *Proc. 8th Int Conf. On Knowledge Discovery and Data Mining (SIGKDD'02)*, pp. 71–80, 2002.