Indexing Process Model Flow Dependencies for Similarity Search*

Ahmed Gater[1], Daniela Grigori[2], and Mokrane Bouzeghoub[1]

[1] Université de Versailles Saint-Quentin en Yvelines
45 avenue des Etats-Unis, 78035 Versailles Cedex, France
[2] Université Paris-Dauphine, Pl. M^{al} de Lattre de Tassigny 75775 Paris, France

Abstract. The importance gained by process models in modern information systems leaded to the proliferation of process model repositories. Retrieving process models within such repositories is a critical functionality. Recent works propose metrics that rank process models of a repository according to their similarity to a given query. However, these methods sequentially browse all the processes of the repository and compare each one against the query, which is computationally expensive. This paper presents a technique for quickly retrieving process models similar to a given query that relies on an index built on behavioral characteristics of process models.

Keywords: semantic process models, process similarity search, process indexing.

1 Introduction

The importance gained by process models in modern information systems and in service oriented architecture leaded to the proliferation of process model repositories. These repositories may store collections of hundreds of process models used by large enterprises, best practices processes (like SAP best practice processes[1]) or reference models provided by process management systems vendors.

Consequently, there is a critical need for tools and techniques to manage process model repositories, including techniques that allow retrieving process models fulfilling user needs. If the user need is formulated or available as a process model, the most similar processes must be retrieved in the repository. Solving this problem, called process similarity search, requires to (i) define a suitable similarity measure and (ii) propose methods that evaluate the similarity between a process query and a set of target processes in the repository. While the first problem received recently considerable attention ([10,3]), very few approaches addressed the second one.

Given an algorithm calculating a similarity measure for two processes, a naive approach to solve the retrieval problem is to traverse all the processes of the

* This work has received support from the National Agency for Research on the reference ANR-08-CORD-009.

[1] `http://www.sap.com/solutions/businessmaps/composer/index.epx`

repository and compare each one against the query, and rank these processes according to their similarity to the query. However, majority of existing process matching algorithms ([11,5]) are NP-complete and therefore they do not scale for large process model repositories.

We propose in this paper an effective and fast similarity search technique that allows retrieving the most similar processes to a user query within a process repository. To this end, we use an abstraction function that represents a process as a finite set of *flow dependencies* between its activities. Thus, the similarity of two processes is defined at the basis of the similarity of their *flow dependencies*. To speed up the comparison of the flow dependencies of the query and those of the repository processes, we define an index structure built on the *flow dependencies* and the activities of the processes. Furthermore, we address the case where a process query cannot be fulfilled by a single target process, but by the composition of several processes. To the best of our knowledge there is not other work allowing to propose the composition of a set of processes as an answer of a query.

The remainder of the paper is organized as follows. The next section presents basic definitions and notations. Section 3 presents the abstraction function we used to represent processes. Section 4 explains the index structures and section 5 shows how they are used for query answering. In section 6 we present an experimental study of our technique. Section 7 discusses related works. Finally section 8 draws conclusions and presents ongoing work.

2 Background and Definitions

A business process model consists of a set of related activities that are combined using control flow operators. In this paper, a process model is formalized as a directed attributed graph (A, C, E), called process graph (p-graph for short), where A is a set of activity nodes, C is a set of connector nodes and E is a set of edges. An activity node represents an atomic task, while connectors represent control flow constraints between activities. An activity $Act = (N, In, Out)$ is described by its name (N), a set of inputs (In), and a set of outputs (Out). The inputs and outputs of activities are annotated with concepts taken from a domain ontology. Connector nodes represent *Split* and *Join* operators of types *XOR* or *AND*. *Split* connectors have multiple outgoing edges, while *Join* connectors have multiple incoming edges.

The processes we handle are block-structured, i.e. sequences, alternative and parallel branchings, and loops are specified with well defined *entry* and *exit* nodes. A block in a process can be an atomic activity, a well-delimited sub-process, or even the process itself. There are five types of blocks: atomic block (a single atomic activity), sequence of blocks $SEQ < B_1, ..., B_n >$, parallel execution of blocks $AND < B_1, ..., B_n >$, alternative execution of blocks $XOR < B_1, ..., B_n >$, and loop through a block $LOOP < B >$. The blocks may be nested, but never overlap, i.e. two blocks are either nested or disjoint (if a node belongs to two blocks then they are nested).

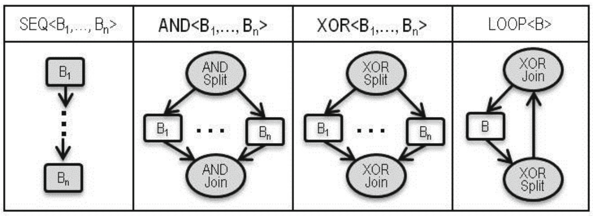

Fig. 1. Block structures

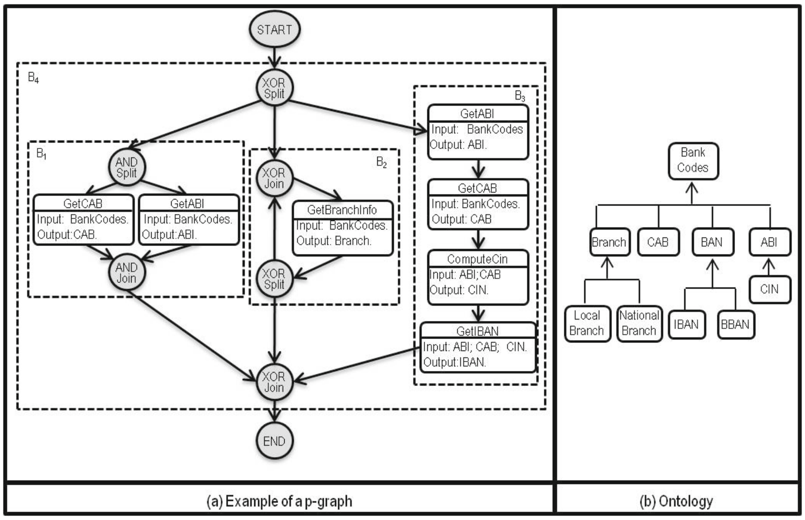

Fig. 2. Running Example

The structures of these blocks are shown by Figure 1, where the highlighted nodes are the *entry* and *exit* of each block. Square boxes represent activity nodes, and the oval ones represent connectors. An example of such block-structured processes is depicted in the left part of Figure 2, where its blocks are shown in the form of dashed boxes (B_1, B_2, B_3 and B_4). The inputs/outputs of its activities are annotated by the ontology depicted in the right part of the same figure. Notice that the assumption that processes are structured into blocks is not strong, since recent studies have shown that most of the unstructured processes can be transformed to equivalent structured processes [14].

In the remainder, we use the notions of smallest block containing a set of activities and loop-free path. Their descriptions are given in Definitions 1 and 2.

Definition 1. The smallest block containing a set of activities
Let (A, C, E) *be a p-graph and* $acts = \{a_1, ..., a_k\} \subseteq A$ *a set of activities. The smallest block containing the activities of acts, denoted* $\delta(acts)$*, is the block B such that:*

– *B contains the activities of acts, and*
– *Every block $B_i \neq B$ which contains the activities of acts contains also B.*

Definition 2. Path and loop-free path. *Let (A, C, E) be a p-graph and a and b be two nodes. A path $a \rightarrow b$ refers to the existence of a sequence of edges $(n_1, n_2), (n_2, n_3), ..., (n_{k-1}, n_k) \in E$, with $k > 1$, $n_1 = a$ and $n_k = b$. A path that does not contain an edge (n_{i-1}, n_i), with n_{i-1} is a connector node of type XOR-Split and n_i is a connector node of type XOR-Join, denoted $a \overset{lf}{\rightarrow} b$, is called loop-free path.*

3 Process Model Representation for Fast Retrieval

While graph indexing algorithms exist in the literature [15], these algorithms can not be directly applied to process graphs. Firstly, p-graphs have two kind of nodes (activities and control nodes) and specific attributes capturing the semantics of the process. Most importantly, p-graphs capture the behavioral semantics of the processes, which is not taken into account by the existing graph indexing techniques that are mainly structural.

The idea is then to transform the p-graphs into another representation which is the most faithfully representative of the p-graphs and, at the same time, makes the evaluation of their similarity faster. This representation must be enough representative to ensure that the similarity of two p-graphs is strongly correlated by the similarity of their new representation.

To this end, we use an abstraction function that captures the essential behavioral characteristics specified by a p-graph. This abstraction function inspired by [6] and called *process type*, represents a p-graph as a finite set of *flow dependencies* that occur between each pair of its activities. A *flow dependency* type specifies how two activities relate to each other, and it is defined as the type of the smallest block containing the two activities as formalized by Definition 3. This definition states that there are four types of *flow dependencies* that may occur between a pair of activities: *"SEQ"* when one of them is always executed after the end of the execution of the other one, *"PATH"* if there exists a loop-free execution path between them, *"AND"* when they are executed in parallel, *"XOR"* when the activities are never executed in the same run.

Definition 3. Flow dependency type. *Let (A, C, E) be a p-graph and $a_i, a_j \in A$ two activities. The type $T_{i,j}$ of the flow dependency that may occur between a_i and a_j is one of the followings:*

– **Sequence flow dependency:** *$T_{i,j}$ = SEQ iff $\delta(\{a_i, a_j\})$ is of type SEQ and $(a_i, a_j) \in E$. The SEQ flow dependency is not commutative, thus, if $T_{i,j}$ = SEQ, then $T_{j,i}$ is not defined.*
– **Path flow dependency:** *$T_{i,j}$ = PATH iff $\delta(\{a_i, a_j\})$ is of type SEQ and $a_i \overset{lf}{\rightarrow} a_j$ occurs. The PATH flow dependency is not commutative, thus, if $T_{i,j}$ = PATH, then $T_{j,i}$ is not defined.*

- **XOR flow dependency:** *$T_{i,j} = XOR$ iff $\delta(\{a_i, a_j\})$ is of type XOR. The XOR flow dependency is commutative, thus, if $T_{i,j} = XOR$, then $T_{j,i} = XOR$.*
- **AND flow dependency:** *$T_{i,j} = AND$ iff $\delta(\{a_i, a_j\})$ is of type AND. The AND flow dependency is commutative, thus, if $T_{i,j} = AND$, then $T_{j,i} = AND$.*

The type of the *flow dependency* of two activities is unique since it is defined on the basis of the type of the smallest block containing the two activities.

In order to take into account loops specified in a p-graph, we define the notion of *flow dependency multiplicity*, emphasing the fact that some flow dependencies involve activities situated in a loop (and thus possible executed several times in a run).

Definition 4. Flow dependency multiplicity. *Let (A, C, E) be a p-graph, $a_i, a_j \in A$ be two activities, $T_{i,j}$ be the type of their flow dependency. The multiplicity of a flow dependency between a_i, a_j, denoted $M_{i,j}$ is:*

- *$M_{i,j} = *$ iff one of the blocks containing the smallest block containing a_i and a_j ($\delta(\{a_i, a_j\})$) is of type LOOP.*
- *$M_{i,j} = 1$ otherwise.*

The *process type* of a p-graph is defined below as the set of the pairs of activities, with their *flow dependencies types* and multiplicities. Notice that for each pair of activities only one *flow dependency* is added to the *process type*. When the *flow dependency type* between two activities is commutative (*XOR* and *AND*), the pair of activities to be added is, by convention, (a_i, a_j) such that the the name of a_i is lexicographically prior to the name of a_j.

Definition 5. Flow dependency and Process type. *Let $P = (A, C, E)$ be a p-graph and $a_i, a_j \in A$ be two activities.*

- *The flow dependency between a_i and a_j is defined as a tuple $fd_{i,j} = (a_i, a_j, T_{i,j}, M_{i,j})$, where $T_{i,j}$ and $M_{i,j}$ are respectively its type and multiplicity.*
- *The process type PT_P of P is the set of all flow dependencies that occur between the pairs of activities of P, such that for any pair of flow dependencies $(a_i, a_j, T_{i,j}, M_{i,j})$ and $(a_k, a_l, T_{k,l}, M_{k,l}) \in PT_P$, $a_i \neq a_k \vee a_j \neq a_l$, $a_i \neq a_l \vee a_j \neq a_k$.*

In the following we use $fd(a_i, a_j)$, $fd(a_i, a_j).Type$, and $fd(a_i, a_j).Multiplicity$ to denote, respectively, the *flow dependency* occurring between activities a_i and a_j, its type, and its multiplicity.

4 Indexing Process Models

Two p-graphs are considered as likely similar when they have similar *process types*, i.e. the more two p-graphs share *flow dependencies*, the more they are similar. The goal is then to speed up the comparison of the *process type* of the query with those of the p-graphs registered in the repository.

To identify p-graphs having similar process types to a query, we first need to identify among registered activities those that are similar to the query activities. Thus, we need an efficient way to find all the activities registered in the repository that are similar to the activities of the query.

Accordingly, we define two index structures, constructed on an off-line pre-processing step of the p-graphs registered in the repository that speed up the evaluation of the queries. The first structure indexes the activities stored in the repository, and is built on the concepts annotating their inputs and outputs. The second one consists on hash tables that store the *process types* of p-graphs as signatures of their *flow dependencies*. We assume hereafter that each p-graph of the repository has a unique identifier, and each activity of each p-graph has also a unique identifier. We assume also that all the p-graphs registered in the repository are annotated using the same ontology. This assumption is not strong since there are many works on ontology alignment and merging (see [7] for a survey in ontology alignment). In the following, we show how these index structures are built.

4.1 Indexing the Activities of the Repository

An important issue in our p-graph retrieval technique is its ability to retrieve among the activities of the repository those that are similar to the activities of the query. The idea is then to define efficient mechanisms that allow finding quickly potential matches of a query activity. Based on the observation stating that activities having similar inputs/outputs are considered more likely to be similar [9,13], we built an index that allows quickly retrieving, among the activities of the repository, those having inputs/outputs similar to those of the query activity.

The index consists of two sets, attached to each concept of the ontology, that record the activities where this concept appears as an input or an output. Precisely, each concept c of the ontology is attached to two sets of annotations In_c and Out_c that record the identifiers of the activities in which this concept appears, respectively, as an input or an output. For instance, let us consider the p-graph example and the piece of the ontology annotating its activities of Figure 2. The attached input and output sets of the concept *"ABI"* are respectively $In_{ABI} = \{ComputeCIN, GetIBAN\}$ and $Out_{ABI} = \{GetABI-1, GetABI-2\}$.

In this work, we are interested in retrieving inexact matches when exact ones do not exist. In other words, an activity in a target process is considered as a potential match for a activity of the query process when it satisfies the constraints (inputs and outputs) of the query activity at a given level, i.e. some mismatches between the inputs/outputs of a query activity and those of a target activity can be tolerated.

For instance, there is no activity of the p-graph depicted in Figure 2 that can strictly fulfill a query activity requiring the concept *"NationalBranch"* as an output. Accordingly, we have to look for activities having as output the concepts the most similar to *"NationalBranch"* on the basis of the relationships between the concepts of the ontology. That way, by considering, for example, the parent concept of *"NationalBranch"* which is the concept *"Branch"*, the activity *"GetBranchInfo"* can be considered as a potential match of this query activity.

Given a concept c annotating an input (resp. output) of a target activity, the idea is to find a reduced set of concepts (to avoid overloaded answers), called *relaxers*, such that, when a query activity requires as input (resp. output) one of the *relaxers* of c, this latter can be considered as a match at a given degree of this activity input (resp. output). To get the set of relaxers of a given concept c, we use three relaxation rules that are formalized in Definition 6. These rules give the possible ways to ralaxe a concept annotating an input or an output. The relaxation rules that have to be applied and the distance (parameter ξ in definition 6, called relaxation degree) between a concept and its relaxers are application dependent reflecting the mismatches that a user accepts for finding activity matches. Notice that the higher is the value of ξ, less accurate are the matches. Setting $\xi = 0$ means that no relaxation is allowed, and thus only exact matches are retrieved. In the remaining, we use $c \overset{\eta}{\hookrightarrow} c'$, where $\eta \in \{desc, asc, cous\}$ to denote that the concept c' is a relaxer of type η of the concept c. Notice that each concept is the *relaxer* of itself, and we note that by $c \overset{origin}{\hookrightarrow} c$.

Definition 6. Concept relaxation rules. *Let c_1 and c_2 be two concepts of the same ontology O, c_{sc} be their least common superconcept, and a natural number $\xi \geq 1$. c_1 can relax c_2 as follows:*

- **Descendant relaxer:** *c_1 is a descendant relaxer of generation ξ (denoted $Desc_\xi$) of c_2 iff c_1 is a sub-concept of c_2, and the length of the path between c_1 and c_2 (in number of intermediate edges) is less than or equal to ξ.*
- **Ascendant relaxer:** *c_1 is an ascendant relaxer of generation ξ (denoted Asc_ξ) of c_2 iff c_1 is a super-concept of c_2, and the length of the path between c_1 and c_2 (in number of intermediate edges) is less than or equal to ξ.*
- **Cousin relaxer:** *c_1 is a cousin relaxer of generation ξ (denoted $Cous_\xi$) of c_2 iff the length of the paths between c_{sc} and c_1 and c_2 are less than or equal to ξ.*

Consequently, the set of annotations attached to the input set In_c (resp. output Out_c) of a concept c records also the identifiers of the activities that have as input (resp. output) one of the concepts that it can relax.

For instance, let us consider the concept *ABI* of the ontology depicted by Figure 2 and the activities of the p-graph of the same figure. By considering the aforementioned relaxation rules and relaxers of generation 1 ($\xi = 1$), the output set attached to concept *"ABI"* contains activity identifiers: *"GetABI-1"* and *"GetABI-2"* (because *"ABI"* is an output of these activities), *"ComputeCIN"* (because this activity has concept *"CIN"* as an output which is a descendant relaxer of *"ABI"* of generation 1), *"GetCAB-1"* and *"GetCAB-2"* (because these activities has concept *"CAB"* as an output which is a cousin relaxer of *"ABI"* of generation 1). The formal description of the attached input and output sets of a concept c are given in Definition 7.

Definition 7. Input and Output annotation sets. *Let AIds and GIds be, respectively, the sets of the identifiers of all the activities and the p-graphs registered in the repository. Let $\eta \in \{desc, asc, cous, origin\}$ be a relaxation rule. Let c be a concept belonging to an ontology O.*

- $In_c = \{(Id_A, Id_g, \eta, ss) | Id_A \in AIds, Id_g \in GIds, \exists c' \in In(Id_A), c' \overset{\eta}{\hookrightarrow} c, ss = InputSim(c', c)\}$
- $Out_c = \{(Id_A, Id_g, \eta, ss) | Id_A \in AIds, Id_g \in GIds, \exists c' \in In(Id_A), c' \overset{\eta}{\hookrightarrow} c, ss = OutputSim(c', c)\}$

The calculation of the similarity between a target concept and its relaxers differs depending on whether it annotates an input or an output. For the case of inputs, the similarity between a concept and its descendant relaxer is 1 since all the attributes required by a target activity input c can be provided by a query activity input which is a descendant of c. Following the same reasoning, the similarity between a target activity output and its ascendants is 1 since the target activity output c can provide all the attributes of a query activity output which is its ascendant. In other cases, any similarity measure [16] can be used to evaluate the similarity between a concept and its relaxer.

The construction of the sets of input and output annotations attached to the concepts of the ontology is done incrementally. Thus, when adding a new activity, only the input and output annotation sets of concepts appearing in this activity
are updated by adding the new annotations. In another hand, when an activity is removed from the repository, only the annotations generated by this activity are deleted. Therefore, adding and deleting an activity do not require the reconstruction of the input and output annotations from scratch, this makes the updating time short.

4.2 Indexing Process Types

As mentioned previously, the p-graphs registered in the repository are compiled into their respective *process types* that are indexed using three hash tables: *Processes*, *Activities* and *FlowDependencies*.

The *Processes* hash table contains the descriptions of the p-graphs registered in the repository, such as the names, the number of their activities, and their storage path. It is indexed by the identifier assigned to the p-graphs. This identifier is auto-generated by incrementing a counter every time that a new p-graph is added to the repository.

The *Activities* hash table stores the descriptions of the activities of the p-graphs registered in the repository. It is indexed by the identifiers assigned to activities that are built by concatenating the identifier of the p-graph to which they belong and a unique identifier distinguishing each activity within the p-graph to which it belongs (generated in the same way as the identifier of the p-graphs). Each entry of this table stores the name, inputs, and outputs of an activity.

The *FlowDependencies* hash table contains the *flow dependencies* specified by the p-graphs registered in the repository. It is indexed by the signatures of these *flow dependencies* that are built as follows.

Let us consider a *flow dependency* $fd_{(i,j)} = (a_i, a_j, T_{(i,j)}, M_{(i,j)})$ occurring in a p-graph, and Aid_i and Aid_j be the identifiers assigned to the activities a_i and a_j. The signature of $fd_{(i,j)}$ is built by concatenating the identifiers of the activities (Aid_i and Aid_j) and the type of the flow dependency ($T_{(i,j)}$): "$Aid_i.Aid_j.T_{(i,j)}$". The signature of each *flow dependency* is unique since there is only one *dependency flow* between each pair of activities. Each entry of the *FlowDependencies* hash table records the multiplicity and the identifier of the p-graph to which it belongs.

As the activity index, the construction of the index for *process types* is done incrementally, by inserting the description of the p-graph and its activities, and its *flow dependencies* in *Processes*, *Activities*, and *flow dependency* hash tables.

5 Process Retrieval

Given a query p-graph $Q = (A_Q, C_Q, E_Q)$ and a set of indexed p-graphs annotated using the same ontology O, the evaluation of Q operates in four steps as shown by Figure 3. First, the process type PT_Q of Q is established (step *Process Type Generator*) following the procedure presented in section 3. Second, the *Activity Matches Searcher* retrieves within the repository the match-candidates of each activity of Q. Next, *Process Matches Searcher* examines the match-candidates of the activities of Q and determines the set of p-graphs that potentially match Q. The result is a list of p-graphs containing at least one pair of activities similar to a pair of activities of Q and having the same *flow dependency type.* These p-graphs are ranked based on the similarity of their *process types* with PT_Q, i.e. p-graphs sharing more *flow dependencies* with Q are better ranked than those sharing less *flow dependencies.* Based on these p-graph match-candidates, the *Process Composition Searcher* tries to discover complex match candidates by composing the p-graph match-candidates. The similarities of these compositions are then evaluated. This finally leads to a ranked list of target p-graphs. Hereafter, we detail these steps.

5.1 Activity Matches Searcher

Given a query activity $A_q = (N_q, In_q, Out_q)$, the *Activity Matches Searcher* retrieves the set of activities registered in the repository that match A_q.

As mentioned above, an activity $A_t = (N_t, In_t, Out_t)$ registered in the repository is a potential match of A_q if it shares at least one direct or relaxed input/output with A_q, i.e. if it exists at least one input (resp. output) of A_t which is an input (resp. output) of A_q or one input (resp. output) of A_q is a relaxer of an input (resp. output) of A_t. The relaxation rules to be considered to generate these match candidates are defined by the user.

To select these activities, the searcher refers to the input/output annotations attached to the ontology concepts. Specifically, given the set of allowed input

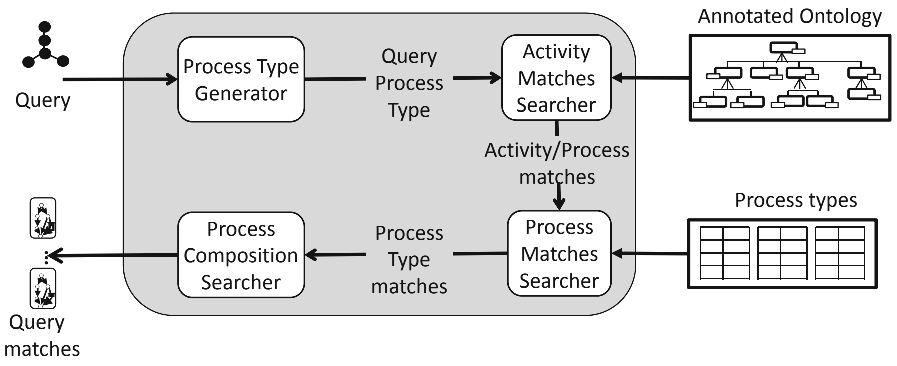

Fig. 3. P-graph retrieval steps

and output relaxation rules specified by a user denoted respectively R_I and R_O, the set of match candidates of A_q is formally described by:

$$ActCandid(A_q, R_I, R_O) = InCandid(A_q, R_I) \cup OutCandid(A_q, R_O).$$

InCandid and *OutCandid* are the functions that, respectively, compute the sets of the activities that share with A_q, at least, one direct or relaxed input and output. They are defined as follows:

$$InCandid(A_q, R_I) = \bigcup_{c \in In_q} \{Id_{a_t} | (Id_{a_t}, Id_g, \eta, ss) \in InAnnot(c), \eta \in R_I\}$$

$$OutCandid(A_q, R_O) = \bigcup_{c \in Out_q} \{Id_{a_t} | (Id_{a_t}, Id_g, \eta, ss) \in OutAnnot(c), \eta \in R_O\}$$

$InAnnot(c)$ and $OutAnnot(c)$ are the functions that retrieve respectively the input and output annotations attached to the concept c using the index built on the p-graph activities of the repository.

Once the set of match candidates is established, the similarity between A_q and each candidate has to be calculated. The similarity between A_q and a target activity A_t identified by Id_{a_t} is defined on the basis of the similarity between their inputs and outputs as stated by the following formula:

$$ActSim(A_q, Id_{a_t}) = \tfrac{1}{2}(\frac{1}{|In_t|} \sum_{c \in In_q} InSim(Id_{a_t}, c) +$$
$$\frac{1}{|Out_q|} \sum_{c \in Out_q} OutSim(Id_{a_t}, c)),$$

where, $|Out_q|$ (resp. $|In_t|$) is the number of outputs (resp. inputs) of A_q (resp. A_t).

The function $InSim(Id_{a_t}, c)$ (resp. $OutSim(Id_{a_t}, c)$) returns the similarity of the input (resp. output) annotation attached to the concept c whose activity identifier is Id_{a_t} when it exists, zero otherwise. This is shown by the following formulas:

$$InSim(Id_{a_t}, c)) = \begin{cases} ss \;\; if\,(Id_{a_t}, Id_g, \eta, ss) \in InAnnot(c) \\ 0 \qquad\quad otherwise \end{cases}$$
$$OutSim(Id_{a_t}, c) = \begin{cases} ss \;\; if\,(Id_{a_t}, Id_g, \eta, ss) \in OutAnnot(c) \\ 0 \qquad\quad otherwise \end{cases}$$

Doing so, some registered activities may be retrieved even if they have a low similarity with the query activity. To keep only promising candidates, we set a similarity threshold that match-candidates have to meet to be considered as potential matches. Therefore, considering a similarity threshold ρ, the set of matches of an activity A_q is: $ActMatches(A_q, R_I, R_O, \rho) = \{Id_{a_t} | Id_{a_t} \in ActCandid(A_q, R_I, R_O), ActSim(A_q, Id_{a_t}) > \rho\}$

Other threshold functions, such as the number of fulfilled inputs and/or outputs, and/or the number of activity match candidates to select can be considered.

Once the activity matches of each activity of Q are established and ordered according to their similarity with this activity, they are passed as input to the *Process Matches Searcher.*

5.2 Process Matches Searcher

Given the sets of activity matches of a query, the *process matches searcher* evaluates the similarity between each p-graph match-candidate and Q, this leads to a ranking of these p-graphs.

The set of p-graph match-candidates of Q is defined as the set of p-graphs with which it shares at least one activity as formalized by the following formula:

$$ProCand(Q, R_I, R_O, \rho) = \bigcup_{A_q \in A_Q} \; \bigcup_{\substack{Id_{a_t} \in \\ ActMatches(A_q, R_I, R_O, \rho)}} PId(Id_{a_t}),$$

where $PId(Id_{a_t})$ gives the identifier of the p-graph to which the activity identified by Id_{a_t} belongs.

Subsequently, a mapping between the activities of each match-candidate T (identified by Id_T) and the activities of Q is established. This mapping contains all the correspondences found between their activities. It is formally defined as follows:

$M_{Q \leftrightarrow T} = \{(a_q, Id_{a_t}, ss) | a_q \in A_Q, Id_T = PId(Id_{a_t}), Id_{a_t} \in ActMatches(a_q, R_I, R_O, \rho), ss = ActSim(a_q, Id_{a_t})\}$, such that for any pair $(a_q^1, Id_{a_t}^1, ss_1)$ and $(a_q^2, Id_{a_t}^2, ss_2) \in M_{Q \leftrightarrow T}$, $a_q^1 \neq a_q^2$ and $Id_{a_t}^1 \neq Id_{a_t}^2$.

The similarity of this mapping is defined as follows:

$$Sim_{M_{Q \leftrightarrow T}} = \frac{1}{|A_Q|} \sum_{(a_q, Id_{a_t}, ss) \in M_{Q \leftrightarrow T}} ss.$$

As stated above, the similarity between Q and its match-candidate T is estimated in the basis of the similarity of their process types PT_Q and PT_T.

Let us consider $\Pi_{Q \leftrightarrow T}$ be the set of fully matched *flow dependencies* between PT_Q and PT_T, that is defined as follows: $\Pi_{Q \leftrightarrow T} = \{(fd_q, fd_t) | fd_q =$

$(a_1^q, a_2^q, \rho_q, m_q) \in PT_Q, fd_t = (a_1^t, a_2^t, \rho_t, m_t) \in PT_T, (a_1^q, a_1^t, ss_1) \in M_{Q \leftrightarrow P}, (a_2^q, a_2^t, ss_2) \in M_{Q \leftrightarrow P}, \rho_q = \rho_t, m_q = m_t\}$.

Let us also consider $\Pi'_{Q \leftrightarrow T}$ be the set of partially matched *flow dependencies* between PT_Q and PT_T, that includes the *flow dependencies* having the same type and having their respective activities matched but having different multiplicities. Its formal description is defined as follows: $\Pi'_{Q \leftrightarrow T} = \{(fd_q, fd_t) | fd_q = (a_1^q, a_2^q, \rho_q, m_q) \in PT_Q, fd_t = (a_1^t, a_2^t, \rho_t, m_t) \in PT_T, (a_1^q, a_1^t, ss_1) \in M_{Q \leftrightarrow P}, (a_2^q, a_2^t, ss_2) \in M_{Q \leftrightarrow P}, \rho_q = \rho_t, m_q \neq m_t\}$.

Finally, we define the similarity between Q and T as follows.

$$SimPro(Q, T) = \omega_m * Sim_{M_{Q \leftrightarrow T}} + \omega_f * \frac{|\Pi_{Q \leftrightarrow T}|}{|PT_Q|} + \omega_p * \frac{|\Pi'_{Q \leftrightarrow T}|}{|PT_Q|}$$

Weights $0 \leq \omega_m \leq 1$, $0 \leq \omega_f \leq 1$, and $0 \leq \omega_p \leq 1$ ($\omega_m + \omega_f + \omega_p = 1$) indicate the contribution of respectively, the matched activities between Q and T, fully and partially matched *flow dependencies* to establish this similarity.

It should be noted that the computation of the shared flow dependencies between a query and a target p-graphs is enhanced using the indice built on the flow dependencies of the p-graphs of the repository.

5.3 Process Composition Searcher

As mentioned before, in some cases a query cannot be satisfied by a single target p-graph, but by a set of p-graphs composed using control structures (sequence, parallelism, alternative branches).The idea is then to develop a technique that is able to propose the composition of several p-graphs as an answer. Such kind of responses are very helpful since it relieves the user of a very tedious composition task.

Informally, two p-graphs can be composed when their respective activities are matched against subgraphs that are disjoint. Figure 4 shows a query and a set of its match candidates. The match candidates T_1 and T_2 are good candidates for composition since their respective activities are matched against subgraphs that are disjoint, while the composition of T_3 and T_4 is not possible since the subgraphs of the query covered by these matches overlap. In other words, a set of p-graphs can be composed if they are mapped to subgraphs of the query that are composable in sequence or using connector nodes of types XOR or AND. This is formalized by Definition 8. This definition states that two p-graphs are composable when the type of the *flow dependencies* occurring between each pair of their respective match activities in the query are of the same type.

There is an exception to this rule that occurs when the two p-graphs are matched against two parts of the query that are in sequence. The exception states that if the *flow dependencies* occurring between the activities of the query matched to a p-graph G_1 and the activities matched to another p-graph G_2 is of type *PATH*, it may exist at most one pair of activities $a_1 \in G_1$ and $a_2 \in G_2$ having a *flow dependency* of type *SEQ*.

Definition 8. Process composition. *Let us consider a query p-graph* $Q = (A_Q, C_Q, E_Q)$ *and a set of its match-candidates:* $T_1 = (A_1, C_1, E_1)$, ..., $T_k = (A_k, C_k, E_k)$. *Let* $M_{Q\leftrightarrow T_k}$, ..., $M_{Q\leftrightarrow T_k}$ *be the mappings found between* Q *and respectively* T_1, ..., T_k. *Let* $M_{Q\rightarrow T_1} = \{a_q | (a_q, a_{T_1}, ss_1) \in M_{Q\leftrightarrow T_1}\}$, ..., $M_{Q\rightarrow T_k} = \{a_q | (a_q, a_{T_k}, s_2) \in M_{Q\leftrightarrow T_k}\}$ *be the sets of the mapped nodes of* Q *following respectively the mappings* $M_{Q\leftrightarrow T_1}$, ..., *and* $M_{Q\leftrightarrow T_k}$. *P-graphs* T_1, ..., *and* T_k *are composable to answer the query* Q *iff:*

- $M_{Q\rightarrow T_1} \cap ... \cap M_{Q\rightarrow T_k} = \emptyset$
- *for each pair* $M_{Q\rightarrow T_i}$ *and* $M_{Q\rightarrow T_j}$, *and each pair of activities* $a_i^1, a_i^2 \in M_{Q\rightarrow T_i}$, *and each pair of activities* $a_j^1, a_j^2 \in M_{Q\rightarrow T_j}$: $fd(a_i^1, a_j^1).Type = fd(a_i^2, a_j^2).Type$

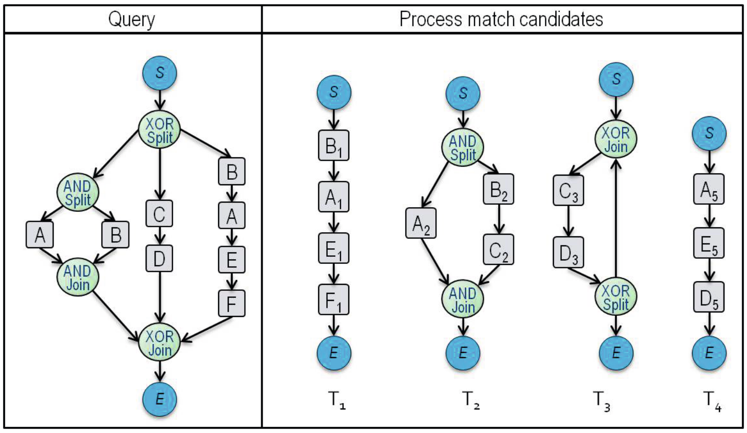

Fig. 4. Process composition example

The composition of a set of p-graphs results in the fulfillment of other *flow dependencies* specified by the query that are not satisfied by one p-graph taken alone. Therefore, the set of fulfilled flow dependencies of the composition of two p-graphs T_i and T_j is defined as follows: $I(T_i, T_j) = \{(a_1, a_2, \gamma, m) | a_1 \in M_{Q\rightarrow T_i}, a_2 \in M_{Q\rightarrow T_j}\}$, where, γ is the type of the *flow dependency* between each pair (a_1, a_2) (it is unique since T_1 and T_2 are composable), m is its multiplicity which is also the same of all the pairs of activities because the composition rule allows only the composition of p-graphs that are mapped to disjoint sub-blocks.

Let $T_s = \{T_1, ..., T_k\}$ be the set of p-graphs to be composed in order to answer the query Q, the set of *flow dependencies* of the query satisfied by the composition of the p-graphs of T_s is: $I_Q = \bigcup_{i=1}^{|T_s|} \bigcup_{j=i}^{|T_s|} (I(T_i, T_j))$. Therefore, the similarity between Q and T_s is defined as follows:

$$SimCompo(Q,T_s) = \sum_{i=1}^{|T_s|} SimPro(Q,T_i) + \omega_f * \frac{|I_Q|}{|PT_Q|},$$ where ω_f is as defined in section 5.2

6 Implementation and Experiments

We implemented our technique on top of a platform for matching p-graphs [4] and experimentally evaluated it. One of the problems we faced to conduct our experiments is the lack of a public benchmark over which we can test our technique, and against which we could compare its result. To overcome this, we built

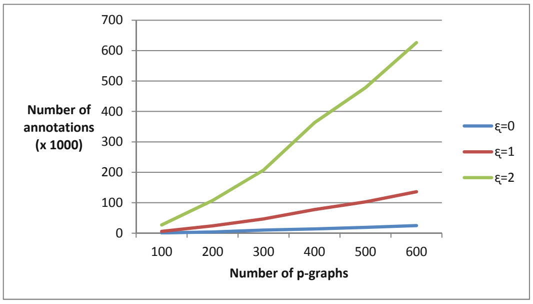

Fig. 5. Number of annotations of the ontology Vs the number of p-graphs and ξ

up our own test collection that we generated by considering the mismatches that most often occur in real life p-graphs and a set of p-graph characteristics that could impact the performances of our technique. The collection contains 623 p-graphs annotated using an ontology containing 500 concepts. On average each p-graph contained 19 activities with a minimum of 2 and a maximum of 83 activities. The average size of an activity name is 2 words with a minimum of 1 and a maximum 8 words.

We randomly extracted from this collection 30 query p-graphs and 300 p-graph targets. We then manually evaluated the similarity between each query and the targets in a 1-7 Likert scale and we subsequently ranked the targets according to their similarity with each query. Details about the methodology we followed to built this collection and its characteristics can be found in [8].

The first experiments were dedicated to the study of the size of the indexes according to the number of indexed p-graphs and the relaxation degree (ξ). From Figure 5, we can see that the number of annotations generated when adding p-graphs is approximatively linear with respect to the number of indexed p-graphs (the tendency is the same whatever the value of ξ). In addition, the number of generated annotations for the same number of indexed p-graphs increases, as expected, with the increasing of the value of ξ. Nevertheless, the number of annotations remains reasonable for a repository containing 600 p-graphs (630k

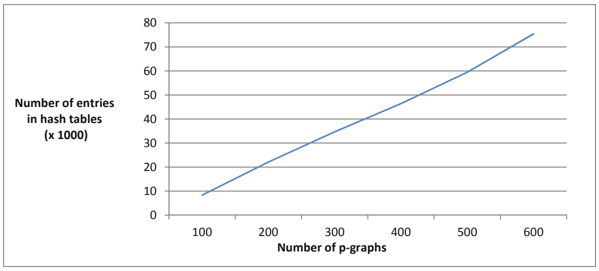

Fig. 6. Sizes of the hash tables Vs the number of p-graphs and ξ

when $\xi = 2$). From Figure 6, we can see that the size of the hashtable index is also approximatively linear with respect to the number of indexed p-graphs. For instance, the indexing of 600 p-graphs required 77k entries in the hashtables, and on average, the indexing of a p-graph required 128 entries. The size of the indexes could become prohibitive for repositories containing thousands of p-graphs, but it supports the indexing of current repositories that, in majority of cases, contain less than 1000 p-graphs. It should be noted that the time for indexing 600 p-graphs is about 300 seconds. Although the indexing of a large number of p-graphs at the same time may require considerable time, it is not penalizing since it is done in an off-line preprocessing step.

Figure 7 shows the effect of varying the number of indexed p-graphs and the relaxation degree ξ (from 0 to 2) on the average and maximum times spent for answering the queries of our benchmark. The results show that query answering is done within low times. For example, the maximum and average time for answering a query are respectively equal to 37 ms and 95 ms when the number of indexed p-graphs is equal to 600 and ξ is equal to 2 (the worst case in our experiments). These results also show that the time of query answering increases

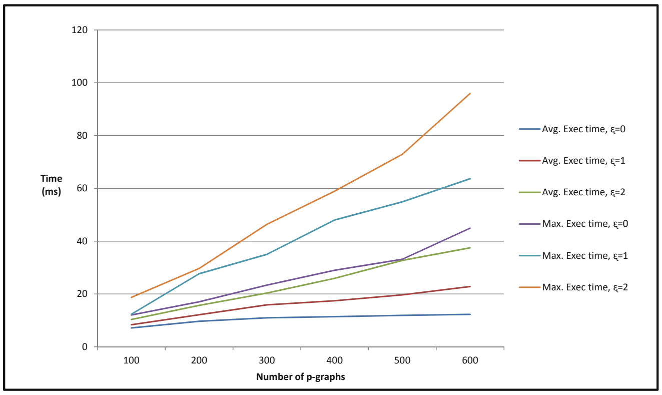

Fig. 7. Query evaluation times vs the number of p-graphs and ξ

with the increasing of the number of indexed p-graphs, and the tendency of this increasing is approximatively linear (whatever the value of ξ).

Finally, we evaluated the quality of the rankings found by our technique with respect to the number of the indexed p-graphs and the value of ξ. The indexes are built only on p-graphs for which a manual evaluation was made (300 p-graphs). The effectiveness of the rankings is evaluated using the well known NDCG formula formalized in Definition 9. The results of this experiment are shown by the graphic of Figure 8. As shown by this graphic, the results obtained by our technique are very satisfactory. For instance, the NDCG obtained when $\xi = 2$ and the number of indexed p-graphs is 300 is equal to 0.81. We also observed that the value of NDCG slightly decreases when the number of indexed p-graphs increases. However, the decreasing is it is negligible when $\xi = 2$.

Definition 9. ***NDCG measure.*** *Let $\Psi = [P_1, P_2, ..., P_n]$ be the ranking found by an algorithm for a given query Q_k, and $\delta^{(k,i)}$ be the similarity degree between Q_k and PM_i given by an expert. Let Z_n be the DCG corresponding to the manual (best) ranking. The effectiveness of the ranking Ψ is: $NDCG_n = \frac{1}{Z_n} \sum_{i=1}^{n} \frac{2^{\delta^{(k,i)}}-1}{log_2(i+1)}$.*

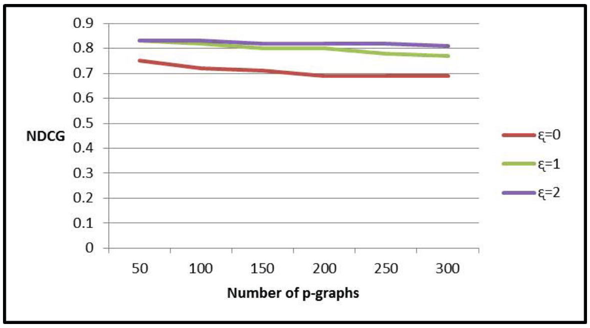

Fig. 8. NDCG Vs the number of indexed p-graphs and ξ

7 Related Work

To the best of our knowledge, the only works addressing the problem of process retrieval in repositories are [17,18,12,1].

In [17], an index build on top of a RDBMS relates a process model to a set of partial traces of length N. A feature-based filtering technique is proposed in [18] to search a collection of processes specified using process graphs. The target processes are ranked according to the number of shared features with the query. Features are small subgraphs that may be: the first and last activities, sequence of a given length, a split node and its successors, and a join node with its predecessors.

A visual query language extending BPMN is proposed in [1] to query graph-based process collection. The processes are stored in a relational database that stores process models nodes (activities and gateway), edges, and paths. The approach is extended in [2] to handle differences of vocabularies that may occur between the query and the stored processes.

Recently, another query language is proposed in [12] that allows specifying the behavioral relationships (two activities can be in sequence, parallel or exclusive branches) that must fulfill the processes to retrieve. The relationships that occur in the stored processes are indexed using an inverted list that relies each behavioral relationship to the set of processes where it occurs.

Many other works [3,10] addressed the problem of measuring the similarity between two processes. These methods sequentially traverse all the processes of the repository and compare each process against the query. The majority of these algorithms are NP-complete and therefore they do not suit for searching similar processes of a query within a repository of processes.

To summarize, we proposed a fast similarity search technique that allows retrieving within a repository the processes the more similar to the query, while the above techniques allow only exact structural matches. Other novel feature of our approach is that it allows proposing a composition of processes in the repository to answer a user query.

8 Conclusion

We propose in this paper an effective and fast similarity search technique that allows retrieving within a process repository, the processes the most similar to the query. Moreover, our approach allows proposing the composition of processes in the repository to answer his query. To do so, we use an abstraction function that represents a process model as a finite set of representative flow dependencies, which are indexed along with the process activities. We implemented our algorithms in a larger platform for matching process models [4] and experimentally evaluated it. Experiments show that our technique has very good execution times.

To reduce the size of indexes, mainly for managing very large repositories, we are currently investigating methods of ontology encoding that reduce the size of the annotated ontology. We also work on methods that allow a more compact representation of the process type relation.

References

1. Awad, A.: Bpmn-q: A language to query business processes. In: EMISA, pp. 115–128 (2007)
2. Awad, A., Polyvyanyy, A., Weske, M.: Semantic querying of business process models. In: EDOC, pp. 85–94 (2008)
3. Becker, M., Laue, R.: A comparative survey of business process similarity measures. Computers in Industry 63(2), 148–167 (2012)
4. Corrales, J.C., Grigori, D., Bouzeghoub, M., Burbano, J.E.: Bematch: a platform for matchmaking service behavior models. In: EDBT, pp. 695–699 (2008)

5. Dijkman, R.M., Dumas, M., van Dongen, B.F., Käärik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. Inf. Syst., 498–516 (2011)
6. Eshuis, R., Grefen, P.: Structural matching of bpel processes. In: Fifth European Conference on Web Services, Halle (Saale), Germany, pp. 171–180 (2007)
7. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (2007)
8. Gater, A.: Process matching and discovery. PhD thesis, University of Versailles (2012)
9. Gater, A., Grigori, D., Bouzeghoub, M.: Complex mapping discovery for semantic process model alignment. In: IIWAS (2010)
10. Gater, A., Grigori, D., Bouzeghoub, M.: A graph-based approach for semantic process model discovery. In: Sakr, S., Pardede, E. (eds.) Graph Data Management:Techniques and Applications, pp. 223–233. Information Science Reference (2011)
11. Grigori, D., Corrales, J.C., Bouzeghoub, M., Gater, A.: Ranking bpel processes for service discovery. IEEE T. Services Computing, 178–192 (2010)
12. Jin, T., Wang, J., Wen, L.: Querying Business Process Models Based on Semantics. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part II. LNCS, vol. 6588, pp. 164–178. Springer, Heidelberg (2011)
13. Klusch, M., Fries, B., Sycara, K.: Automated sematic web discovery with owls-mx. In: AAMAS 2006 (2006)
14. Polyvyanyy, A., García-Bañuelos, L., Dumas, M.: Structuring Acyclic Process Models. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 276–293. Springer, Heidelberg (2010)
15. Sakr, S., Al-Naymat, G.: Graph indexing and querying: a review. International Journal of Web Information Systems 6(2), 101–120 (2010)
16. Valery, C.: Fuzzy semantic distance measures between ontological concepts. In: NAFIPS, pp. 635–640 (2004)
17. Wombacher, A., Mahleko, B., Fankhauser, P.: A grammar-based index for matching business processes. In: ICWS 2005 (2005)
18. Yan, Z., Dijkman, R., Grefen, P.: Fast Business Process Similarity Search with Feature-Based Similarity Estimation. In: Meersman, R., Dillon, T.S., Herrero, P. (eds.) OTM 2010, Part I. LNCS, vol. 6426, pp. 60–77. Springer, Heidelberg (2010)