

# Personal Process Description Graph for Describing and Querying Personal Processes

Jing Xu<sup>1</sup>, Hye-young Paik<sup>1</sup>, Anne H. H. Ngu<sup>2</sup>, and Liming Zhan<sup>1</sup>

<sup>1</sup> The University of New South Wales, Sydney, NSW, Australia  
{jxux494, hpaik, zhanl}@cse.unsw.edu.au

<sup>2</sup> Texas State University, Austin, Texas, USA  
angu@txstate.edu

**Abstract.** Personal processes are ad-hoc to the point where each process may have a unique structure and is certainly not as strictly defined as a workflow process. In order to describe, share and analyze personal processes more effectively, in this paper, we propose Personal Process Description Graph (PPDG) and a personal process query. The personal process query approach is developed to support different types of graph queries in a personal process graph repository. The approach follows a filtering and refinement framework to speed up the query computation. We conducted experiments to demonstrate the efficiency of our techniques.

## 1 Introduction

People are often confronted with tasks that are infrequent (even one-off) and not well-described. Examples are applying for jobs, buying a house, or filing a tax return. We refer to these as “personal processes” in which an individual performs a set of logically related tasks to accomplish some personal goals [7, 15].

To overcome their lack of familiarity, people typically seek out other people’s experiences with the task. It is difficult and time-consuming to capture and share personal processes, although we have many kinds of channels to share variety of information such as photos or video. Furthermore, searching and understanding already shared processes are limited to a simple keyword search via search engines. We see the following problems in existing publicly available personal process repositories such as how-to guides Web sites or Q/A forums.

- With regards to capturing a personal process, the current model of describing processes, which is based on free texts and bullet points, makes it difficult to (1) understand the process especially if there are many steps involved; (2) compare and contrast different paths/ways to accomplish the goal; and (3) understand the dependencies between data and actions.
- For searching and analyzing a personal process, the current model of keyword search over the textual descriptions is limited and is not able to provide exploratory answers to query. That is, the search always returns a set of Web pages containing the descriptions of the whole processes even if the user just requires a fragment of the process or a specific task in the process.

In order to describe, share and analyze personal processes more effectively, we initiate a novel framework for personal process management named ProcessVidere. Our ultimate aim is to provide a space for users to share their experiences, capture the process knowledge from people, analyze them effectively, and support users to re-use the whole or part of the processes for their own purposes. In this paper, as a concrete step towards building ProcessVidere, we present the following elements as the fundamental components of the system:

- Personal Process Description Graph (PPDG), a graph-based description language to present both control flow and data flow of a personal process.
- A template-based approach to perform structured queries over PPDG, as well as an implementation of the key template queries and preliminary performance evaluation results.

The paper is organized as follows: Section 2 introduces the PPDG language. Section 3 presents the basic design of PPDG query templates. Section 4 describes methods and algorithms for an efficient processing of the query templates. Then we present the experiment results in Section 5. The related work is discussed in Section 6 followed by a conclusion in Section 7.

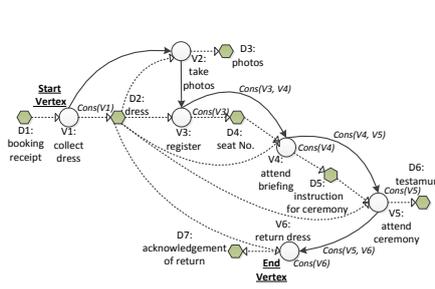
## 2 Personal Process Description Graph (PPDG)

Let us consider “attending a graduation ceremony at UNSW”. We interviewed six UNSW graduates and asked them to describe the process. A collective summary of the process is as follows:

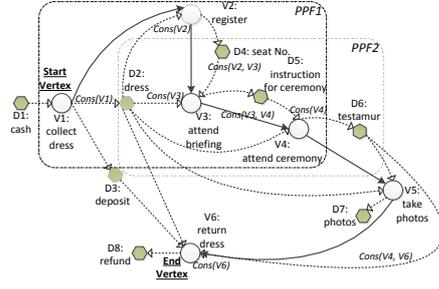
*Before the graduation day, a graduand may book a graduation gown online. The booking can be paid for by card (in which case a receipt is issued), or by cash when collected. On the day, the graduand collects the dress, take a photo and register for the ceremony to obtain a seat number. After a briefing session, the graduand attends the ceremony. The dress is returned after the ceremony.*

Not every step is strictly followed, in fact, six different variations of this personal processes are obtained. A typical workflow management or business process modelling (BPM) approaches would attempt to create a single reference model that describes the above process as complete as possible. However, building such a model for a flexible and ad-hoc process is difficult and often makes the model convoluted. As discussed in Section 6, existing workflow or BPM systems are not equipped to deal with ad-hoc processes very well [7, 15, 8].

In ProcessVidere, instead of developing and managing a reference model, we consider a personal process as process steps experienced and described by a single person. Each description of the process becomes a feasible process, the so-called “model”, that leads to the intended goal. PPDG described below is designed to capture the description of an individual experience of a process. Later in the paper, we will explain how queries are written over these descriptions to give a flexible view of the process.



**Fig. 1.** PPDG1: UNSW graduation ceremony process by Graduant A



**Fig. 2.** PPDG2: UNSW graduation ceremony process by Graduant B

**2.1 PPDG Syntax Overview**

PPDG represents a personal process  $P$  as a labelled directed graph. It describes the whole process of performing a personal process placing *equal emphasis* on both actions and input/output data related to each action in the process. Figure 1 depicts a PPDG of the graduation ceremony process experienced by Graduant A. An action is noted by a circular node. Actions can be divided into two categories: a simple action and composite action which contains another process. A simple action can be one-off action, repeated action, or duration action. In order to make the visualization of the graph simple, all types of actions are represented using the same notation. The details are stored in the schema associated with the process graph.

Data elements in a process  $P$  are represented by hexagonal nodes. A data element can be either basic or composite (i.e., composition of basic data). In PPDG, data could be any artifacts that is available somewhere: physical items such as paper documents, academic dress, or digital items such as digital photos, text messages and so on. A data element can be external (i.e., it is not produced by any action) or processed (i.e., there is at least one action produced it).

The data elements and actions are connected to represent ‘action flow’ or ‘data flow’. Action flows, represented by solid lines, describe temporal sequence of the actions. For example, in Figure 1, ‘ $V_1$ : collect dress’ takes places before ‘ $V_2$ : take photos’. Data flows, represented by dotted lines, keep track of data sources and denotes the relationships between the data and actions. For example, ‘ $V_4$ :attending briefing’ takes two data inputs ‘ $D_2$ : dress’ and ‘ $D_4$ : seat no.’ and produces one data output ‘ $D_5$ : instructions for ceremony’.

PPDG also stores constraints/conditions relating to action, data and the flows. If the constraints concern an action (referred to as ‘action constraints’), it may include conditions such as the location or the time the action takes place. If the constraints are on the connecting edges of two actions (referred to as ‘transition constraints’, it may specify the conditions that should be met for the flow to take place. It can also be used to enforce the sequence of the two actions. We define PPDG more formally as follows.

**Definition 1.** A personal process description graph PPDG is a tuple  $PPDG := (A, D, E_A, E_D, C, \phi, \lambda)$  where:

- $A$  is a finite set of nodes  $a_0, a_1, a_2, \dots$  depicting the starting action ( $a_0$ ) and actions ( $a_1, a_2, \dots$ ).
- $D$  is a finite set of nodes  $d_0, d_1, d_2, \dots$  depicting the data input/output of an action.
- $E_A$  is a finite set of directed action-flow edges  $ea_1, ea_2, \dots$ , where  $ea_i = (a_j, a_k)$  leading from  $a_j$  to  $a_k$  ( $a_j \neq a_k$ ) is an action-flow dependency. It reads  $a_j$  takes place before  $a_k$ . Each node can only be the source/target of at most one action-flow edge :  $ea = (a_i, a_j) \in E_A : ea' = (a_k, a_l) \in E_A \setminus ea : a_i \neq a_k$  and  $a_j \neq a_l$ .
- $E_D$  is a finite set of directed data-flow edges  $ed_1, ed_2, \dots$ , where  $ed_i = (a_j, d_k)$  leading from  $a_j$  to  $d_k$  is a data-flow dependency. It reads  $a_j$  produces  $d_k$ .  $ed_l = (d_m, a_n)$  leading from  $d_m$  to  $a_n$  is a data-flow dependency. It reads  $a_n$  takes  $d_m$ .
- $C$  is a finite set of conditions  $c_1, c_2, \dots$  with  $c_i = (< \text{name}, \text{descr} >, x_j)$  being associated to  $x_j \in \{A, D, E_A, E_D\}$  and having name and description of the condition.
- $\phi$ : a function that maps Label  $L(A_i)$  to action nodes.
- $\lambda$ : a function that maps Label  $L(D_i)$  to data nodes.

### 3 Query Templates and Their Basic Constructs

A repository of PPDG is organized by categories and domains. The idea in ProcessVidere is that this repository can be simply keyword searched or browsed, but more importantly, it can be used to perform structured queries for sophisticated analysis. One might pose a query “How to attend a graduation ceremony at UNSW” to understand the whole process. Others might want some answers to particular aspects of the process, such as “What happens after attending the briefing session?”, “How do I get a seat no.?”, “I have collected the dress, what can I do with it?”.

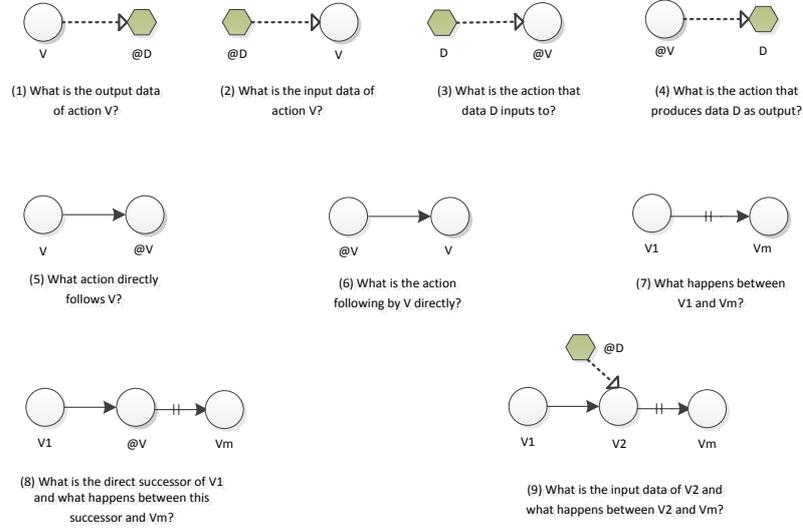
In answering these queries, we will develop various graph-based query processing techniques. In this section, as the first step, we present the basic design of the single match approach. We envisage that this basic set of query constructs will allow us to build more complex query approaches in our immediate future work to implement the similarity (e.g., the semantic mismatch issues in the text labels) or graph composition/aggregation.

For simplicity, we remove  $C$  (conditions) from PPDG here on. We propose a template-based approach in which three types of query template constructs are defined: *atomic*, *path* and *complex* query templates. We also assume that the user can enter one of the query templates directly through ProcessVidere, using an editor not dissimilar to BPMN-Q query editor [12].

Figure 2 introduces another PPDG on graduation ceremony. Although it looks similar to Graduand A process (Figure 1), there are a few differences. For example, ‘ $V_1$ : collect dress’ takes ‘ $D_1$ : cash’ and produces ‘ $D_3$ : deposit’. ‘ $V_6$ : return dress’ takes ‘ $D_3$ : deposit’ as well as the dress. Also, ‘take photo’ appears after ‘attend ceremony’ (rather than before ‘register’).

**Atomic Query Templates.** These will match an action-flow edge ( $\in E_A$ ) or a data-flow edge ( $\in E_D$ ) and the nodes ( $\in A, D$ ) directly connected by it.

$Q_1$  to  $Q_6$  in Figure 3 represent atomic query templates. The text label shown in each template describes the query contained in it. The prefix symbol “@” in the node label indicates a variable node (i.e., “@D” for a variable data node, “@V” for a variable action node).



**Fig. 3.** Atomic Query, Path Query and Complex Query Templates over PPDG

*Example 1.* Let us assume that we issue  $Q_2$  with  $v$  as  $V_1$ : *collect dress*. The results (processed over PPDG1 and PPDG2) will be  $\{PPDG1.ed_1 = (D_1: \textit{booking receipt}, V_1: \textit{collect dress}), PPDG2.ed_1 = (D_1: \textit{cash}, V_1: \textit{collect dress})\}$ . So the answer shows that *collect dress* can either take cash or a booking receipt.

*Example 2.* Similarly, we issue  $Q_5$  with  $v$  as  $V_1$ : *collect dress*. The results (processed over PPDG1 and PPDG2) will be  $\{PPDG1.ea_1 = (V_1: \textit{collect dress}, V_2: \textit{take photos}), PPDG2.ea_1 = (V_1: \textit{collect dress}, V_2: \textit{register})\}$ . So the answer shows that *collect dress* can be followed by either *taking photos* or *registering*.

**Path Query Templates.** Intuitively, a path query template is designed to match a fragment of PPDG contained by two action nodes. Visually, we denote the template as shown in Figure 3 ( $Q_7$ ). The symbol “||” is used to represent a path query between action nodes specified by  $V_n$  and  $V_m$  (where  $n, m$  are the action node numbers). To explain, we first define a path in PPDG.

**Definition 2.** A path of PPDG is a tuple  $P_{ath} = (A', D', E'_A, E'_D, \phi, \lambda)$  where:

- $P_{ath} \subseteq PPDG$ .
- $x \in A'$  iff:
  - $x = \textit{source}$ .
  - $x = \textit{destination}$ .
  - $x$  lies on a path from source to destination in  $ppdg \in PPDG$ .
- $\forall x, y \in A', e(x, y) \in E_A \rightarrow e(x, y) \in E'_A$
- $d \in D'$  iff:
  - $d$  is output data of source action in a path.
  - $d$  is input data of destination action in a path.
  - $d$  is input/output data of action  $x$  which lies on a path from source to destination in  $ppdg \in PPDG$ .

- $\forall d \in D', e(d, x) \in E_D \rightarrow e(d, x) \in E'_D$  and  $e(x, d) \in E_D \rightarrow e(x, d) \in E'_D$

A path query template will return a single matching path from each PPDG considered for processing.

*Example 3.* Let us assume that we set  $n = 1$  and  $m = 4$  in  $Q_7$ , so that  $Q_7$  is a path query to find personal process fragment between ‘ $V_1$ : collect dress’ and ‘ $V_4$ : attend ceremony’. Therefore, the returned result of  $Q_7$  (processed over PPDG2) is a path shown as  $PPF1$  (dotted box) in Figure 2.

**Complex Query Template.** A complex query template is a composite of atomic and/or path query templates.  $Q_8$  and  $Q_9$  in Figure 3 are the samples of complex queries that are composed of  $Q_5$  and  $Q_7$  and a constant action,  $Q_2$  and  $Q_7$ , respectively. The complex query of PPDG can consist of  $Q_1$  to  $Q_6$  or  $Q_1$  to  $Q_6$  and  $Q_7$  or  $Q_1$  to  $Q_7$  with constant actions/data.

## 4 Query Processing

Here we formally introduce the definitions of PPDG query and fragment, the schema of PPDG with detailed algorithms to process query templates.

**Definition 3 (PPDG Query).** A query graph is a tuple  $PPDG - Q = (QA, QD, QAF, QADF, QP, T, S)$  where:

- $QA$  is a finite set of action nodes in a query.
- $QD$  is a finite set of data nodes in a query.
- $QAF \subseteq QA \times QA$  is the action flow relation between action nodes in a query.
- $QADF \subseteq QA \times QD$  is the data flow relation between action nodes and data nodes in a query.
- $QP$  is the path relation between action nodes which includes data nodes and data edges corresponding to each action node in query.
- $T: QA \rightarrow \{CONSTANT ACTION, VARIABLE ACTION\}$
- $S: QD \rightarrow \{CONSTANT DATA, VARIABLE DATA\}$

**Definition 4 (PPDG Fragment (PPF)).** A connected subgraph  $(A', D', E'_D)$ , or  $(A', E'_A)$ , or  $(A', D', E'_A, E'_D)$  of a personal process description graph  $(A, D, E_A, E_D)$ , where  $A' \subseteq A$ ,  $D' \subseteq D$ ,  $E'_A \subseteq E_A$ ,  $E'_D \subseteq E_D$ , is a fragment of the personal process description graph.

A query template can return 0, 1 or more PPDG Fragments. Figure 2 shows examples of PPF in  $PPF1$  (dotted box) and  $PPF2$  (dotted box).

**PPDG data schema** The schema for our PPDG has the following tables:

- Graph(ID, Name, Description)
- Action(ActionID, GraphID, ActionLabel, Type1, Type2, SubProcess, Description)
- Data(DataID, GraphID, DataLabel, Type1, Type2, SubProcess, Artifacts)
- Edge\_Action(sActionID, dActionID, GraphID)
- Edge\_Data(DataID, ActionID, GraphID, Direction)

We store the descriptions of graphs, actions, and data in three tables, respectively. The relationships between actions and data are stored in two tables. Specially, the *Direction* item in table *Edge\_Data* indicates whether the data is the input or output of an action.

**Atomic Queries ( $Q_1 \sim Q_6$ ).** Using the label of an action (data), we can get the *ActionID* (*DataID*) by matching the label in table *Action* (*Data*). Then the *IDs* of related actions and data can be derived from table *Edge\_Action* and *Edge\_Data*. After that, the label of the related actions and data can be found in table *Action* and *Data*. Each atomic query can be implemented by a single SQL statement. For instance, the results of  $Q_5$  are obtained by the following SQL statement when giving  $Q_5(\textit{ActionLabel})$ .

```
SELECT da.ActionID, da.GraphID, da.ActionLabel
FROM query q, Action sa, Action da, Edge_Action e
WHERE q.ActionLabel=sa.ActionLabel
and e.sActionID=sa.ActionID and e.GraphID=sa.GraphID
and e.dActionID=da.ActionID and e.GraphID=da.GraphID;
```

The other five atomic query templates are straightforwardly implemented with minor modifications of the SQL statement above.

**Path Query ( $Q_7$ ).** For the implementation of the path query template, intuitively, we can use the six atomic queries ( $Q_1 \sim Q_6$ ) above to get the path step by step: (1) First, launch atomic query  $Q_5$  from the first action node  $sA$ , and get a set  $\mathcal{S}$  of action nodes that are direct successors of  $sA$  from different processes. (2) Then we perform the same atomic query for each action node in  $\mathcal{S}$ . (3) The same procedure is executed iteratively until we reach the  $dA$  or end of the process. The results are a set  $\mathcal{A}$  of action paths. (4) For each action path in  $\mathcal{A}$ , we launch  $Q_1$  and  $Q_2$  to find the data nodes related to every action nodes. After that, the final results are obtained.

The cost of this initial algorithm is very high due to querying redundant processes which do not contain the end action node  $dA$  of the path. Therefore, we filter all the processes to obtain a small set  $\mathcal{G}$  of processes which contains both  $sA$  and  $dA$  nodes before the above steps (1 to 4) are performed.

The Algorithm 1 illustrates the details of path query processing. To enable computing the path in an iterative fashion, we use a tuple  $T$  to process the path query.  $T$  is employed to maintain the nodes information used for the path computation of a set of action and data nodes in a graph. Particularly,  $T.graph$  indicates which graph the path locates in,  $T.queue$  is a queue of action nodes on the path, and  $T.data$  is a set of data nodes on the path. First, we filter the graphs by the start and end actions of the path in Line 1. Then, Line 3-5 initialize  $T$  for each related graph and store them in a set  $\mathcal{T}$ . From Line 6 to Line 15, we find the actions on the path using the iterative method. Particularly, Line 8 launches  $Q_5$  to find the next action node  $r$  of current action node in the tail of  $T.queue$ . If there is no result from  $Q_5$ , the  $T$  is pruned. Otherwise, we push  $r$  into  $T.queue$  (Line 10). If  $r$  is  $dA$ , we get one exact result and put it into result set  $\mathcal{R}$  (Line 12). If  $r$  is not  $dA$ , we continue to search for the next action (Line 14, 15). Finally, Line 16-17 launch  $Q_1$  and  $Q_2$  for each actions in  $T \in \mathcal{R}$  to fill the data nodes for each path.

**Algorithm 1: Path Query( $sA, dA$ )**


---

**Input** :  $sA, dA$ : The start and end actions of the path  
**Output**:  $\mathcal{R}$ : The result set of  $P_{ath}$

- 1  $\mathcal{G} \leftarrow$  all the processes contain both  $sA$  and  $dA$ ;
- 2  $\mathcal{T} = \phi; \mathcal{T}' = \phi$ ;
- 3 **for** each  $graph \in \mathcal{G}$  **do**
- 4      $T.graph = graph; T.queue.push(sA)$ ;
- 5      $\mathcal{T} \leftarrow T$ ;
- 6 **while**  $\mathcal{T} \neq \emptyset$  **do**
- 7     **for** each  $T \in \mathcal{T}$  **do**
- 8          $r = Q_5(T.queue.back())$  on  $T.graphID$ ;
- 9         **if**  $r \neq NULL$  **then**
- 10              $T.queue.push(r)$ ;
- 11             **if**  $r == dA$  **then**
- 12                  $\mathcal{R} \leftarrow T$ ;
- 13             **else**
- 14                  $\mathcal{T}' \leftarrow T$ ;
- 15      $\mathcal{T} = \mathcal{T}'; \mathcal{T}' = \emptyset$ ;
- 16 **for** each  $T \in \mathcal{R}$  **do**
- 17     Find related data nodes of all action nodes in  $T$  by  $Q_1$  and  $Q_2 \rightarrow T.data$ ;
- 18 **return**  $\mathcal{R}$ ;

---

**Complex Query.** We decompose the complex query first, and then utilize methods of atomic and path queries mentioned above to find matched fragments/processes. The filtering and verification methods are used to improve the performance of the algorithm. To begin with, we split the complex query into one *node set* and four *pair sets*: Constant node set  $S_n$ , Constant pair set  $P_{con}$ , Variable pair set  $P_v$ , Path pair set  $P_{path}$  and Preprocessing pair set  $P_{pre}$ .  $S_n$  contains all constant nodes appearing in the query. In any of the four *pair sets*, a *pair* is described as  $\{n, n'\}$  -  $n$  and  $n'$  are nodes linked by one edge or path in a graph.  $n$  or  $n'$  could be either a constant node or a variable node. We store the pairs with two constant action/data nodes linked by an edge in  $P_{con}$ . The pairs which can be processed directly by using atomic queries stored in  $P_v$  and those that can be processed by path query stored in  $P_{path}$ , respectively. And the pairs which need to be preprocessed before using the methods mentioned above are put into  $P_{pre}$ . That is, all the pairs in  $P_{pre}$  depend on the results from  $P_v$  to identify one node of each pair before using the methods of atomic queries or path query directly. We use  $S_n$  and  $P_{con}$  to find a candidate set  $\mathcal{C}$  of graphs, and then prune and verify  $\mathcal{C}$  by  $P_v, P_{path}$  and  $P_{pre}$ .

The details of the complex query processing is illustrated in Algorithm 2. Similar to Algorithm 1, we also use a tuple  $T$  to store the intermediate result. Particularly,  $T.graph$  indicates which graph the path locates in,  $T.R$  stores the known action/data nodes, and  $T.Pair$  stores the pairs that exist in  $T.graph$ . Line 1 splits query  $Q$  into one nodes set and four pair sets. Then, we use  $S_n$  and  $P_{con}$  to filter all the processes to get a set  $\mathcal{G}$  of processes in Line 2. Based

---

**Algorithm 2: Complex Query( $Q$ )**


---

**Input** :  $Q$ : The complex query  
**Output**:  $\mathcal{R}$ : The result set of the complex query

- 1 Decompose  $Q$  into nodes set  $S_n$  and pair sets  $P_{con}, P_v, P_{path}, P_{pre}$ ;
- 2  $\mathcal{G} \leftarrow$  all the processes containing the nodes in  $S_n$  and matching the pairs in  $P_{con}$ ;
- 3  $\mathcal{C} = \phi$ ;  $\mathcal{C}' = \phi$ ;
- 4 **for** each  $graph \in \mathcal{G}$  **do**
- 5      $T.graph = graph$ ;  $T.R \leftarrow$  all nodes in  $S_n$ ;
- 6      $\mathcal{C} \leftarrow T$ ;
- 7 **for** each pair  $\{n, n'\} \in P_v$  **do**
- 8      $\mathcal{U} \leftarrow$  results of query  $Q(n, n')$  on  $\mathcal{G}$ ;
- 9     Join  $\mathcal{U}$  and  $\mathcal{C}$  on  $graph$  attribute; update  $\mathcal{G}$ ;
- 10    **for** each  $(U, T) \in \mathcal{U} \times \mathcal{C}$  **do**
- 11        $T.R \leftarrow U.r$ ;
- 12       **if** variable  $v \in P_{pre}$  is identified by  $U.r$  **then**
- 13            $T.Pair \leftarrow \{u, v'\}$  or  $\{v', u\}$ ;
- 14        $\mathcal{C}' \leftarrow T$ ;
- 15     $\mathcal{C} = \mathcal{C}'$ ;  $\mathcal{C}' = \phi$ ;
- 16 **for** each pair  $\{n, n'\} \in P_{path}$  **do**
- 17      $\mathcal{V} \leftarrow$  results of path query  $Q(n, n')$  on  $\mathcal{G}$ ;
- 18     Join  $\mathcal{V}$  and  $\mathcal{C}$  on  $graph$  attribute; update  $\mathcal{G}$ ;
- 19     **for** each  $(V, T) \in \mathcal{V} \times \mathcal{C}$  **do**
- 20        $T.R \leftarrow V.R$ ;
- 21        $\mathcal{C}' \leftarrow T$ ;
- 22      $\mathcal{C} = \mathcal{C}'$ ;  $\mathcal{C}' = \phi$ ;
- 23 **for** each  $T \in \mathcal{C}$  **do**
- 24     Process queries on all pairs in  $T.Pair$ ;
- 25      $\mathcal{R} \leftarrow T$  if all queries have results on  $T$  ;
- 26 **return**  $\mathcal{R}$ ;

---

on  $\mathcal{G}$  and  $S_n$ , we get a candidate set  $\mathcal{C}$  containing intermediate results  $T$  in Line 4-6. From Line 7 to 15, we process atomic queries on each pair in  $P_v$ , and then use the results to prune and verify the candidate set. Then, we get the result set  $\mathcal{U}$  (Line 8) for each query. For each result  $U \in \mathcal{U}$ ,  $U.r$  is a result node on graph  $U.graph$  for atomic query. Next,  $\mathcal{U}$  and  $\mathcal{C}$  are joined on their  $graph$  attribute (Line 9), so that the unjoined tuples in  $\mathcal{C}$  can be pruned safely. If one variable node  $v$  of any pair in  $P_{pre}$  is identified by  $U.r$ , we replace the variable node  $v$  with  $U.r$ , and store the corresponding pair in  $T.Pair$ , because the pair only appears in  $T.graph$  (Line 10-13). From Line 16 to 22, we process path query on each pair in  $P_{path}$ , which can be used for further pruning. Finally, we process atomic query or path query on each pair in  $T.Pair$  to obtain the complex query result set  $\mathcal{R}$  (Line 23-25).

*Example 4.* To process query  $Q_8$  in Figure 3 (set  $m=5$ ), we split the query as  $S_n = \{V_1, V_5\}$ ,  $P_{con} = \{\phi\}$ ,  $P_v = \{\{V_1, @V\}\}$ ,  $P_{path} = \{\phi\}$ , and  $P_{pre} = \{\{@V, V_5\}\}$ . Filtering by  $S_n$  and  $P_{con}$ , suppose we get the process in Figure 2, then we process the query on  $P_v$  by using  $Q_5(V_1, @V)$  to get the result  $\{V_1, V_2\}$ . Note that  $V_2$

matches the node in pair  $\{@V, V_5\} \in P_{pre}$ , so the new pair  $\{V_2, V_5\}$  is stored in  $T.pair$ .  $P_{path}$  is empty and no further processing is necessary. Finally, we process the query on the pair  $\in T.pair$  and assemble all results to get the fragment of graph.

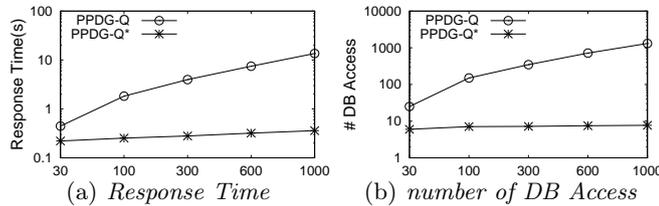
## 5 Performance Evaluations

In this section, we present results of a performance study to evaluate the efficiency and scalability of the proposed techniques in the paper.

**Datasets** We have evaluated our query processing techniques on both real and synthetic datasets. We invited 14 volunteers to describe the procedures they went through at Graduation Research School of UNSW, and generated 30 different processes with 295 action nodes and 441 data nodes. The synthetic datasets were generated by randomization techniques. By giving 30 action labels and 30 data labels, we randomly choose  $n$  action labels with  $n + 2$  data labels to assemble  $p$  processes. The number  $p$  varies from 100 to 1000 (default value = 300). The number  $n$  of action nodes in each process varies from 10 to 25 (default value = 10). By the default setting, the total number of action nodes and data nodes were 3000 and 3600 respectively in our experiment. We had a total of 10 queries in the experiment. Each query was generated randomly. The average response time and I/O cost of the 10 queries on each dataset represent the performance of our query processing mechanism.

All algorithms have been implemented in Java and compiled on JRE 1.6. We used PostgreSQL to store the data. The experiments were run on a PC with Intel Xeon 2.40GHz dual CPU and 6G memory on Debian Linux.

We have measured the I/O performance of the algorithms by measuring the number of database access. Query response times were recorded to evaluate the efficiency of the algorithms, which contained the CPU time and the I/O latency.



**Fig. 4.** Number of Process ( $p$ )

**Impact of the number of processes ( $p$ ).** We varied the value of  $p$  and evaluated the performance of our algorithm against the real datasets where  $p$  is 30 and the synthetic datasets where  $p$  varies from 100 to 1000 in Figure 4. With a larger number of processes, more processes are involved in the computation, thus incurring higher computation cost. From Figure 4(a) and Figure 4(b), we found the result of the response time and number of database access had the same increasing trend as the number of processes increased. That means I/O costs is the main contribution to the query processing cost. Due to the high cost of database access, we can optimize the technique of PPDG query (denoted PPDG-Q here) proposed in Section 4 by storing the actions node of  $graph \in \mathcal{G}$

in cache. This optimization is represented as PPDG-Q\* in our experiments. As expected, PPDG-Q\* significantly outperforms PPDG-Q and is less sensitive to the growth of  $p$ .

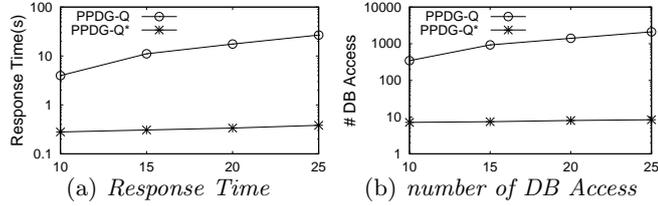


Fig. 5. Action Nodes of Process ( $n$ )

**Impact of the number of action nodes in each process ( $n$ ).** Figure 5 investigates the impact of the number of action nodes in each process on the algorithms where  $n$  grows from 10 to 25. With the growth of  $n$ , more action/data nodes are involved in the query computation, thus the response time and the number of database accesses increase. The results show that the scalability of PPDG-Q\* is better than that of PPDG-Q regarding the growth of  $n$ .

## 6 Related Work

We discuss related work broadly in two categories: business process modelling languages and process query approaches.

**Modelling Business Processes.** There are many models developed to represent business processes, Petri Net, Pi-Calculus and recently, Business Process Model and Notation (BPMN) [1, 11] to name a few. In all of these models, however, the main focus is in formalizing action flows in a process and data flows are mostly ignored. They also tend to provide highly expressive and sophisticated constructs which make them a complicated tool to use for personal processes. With increasing interests in management of personal workflows, many argue that a process modelling approach that reduces expressive power and simplifying notations for personal processes is necessary [5, 15, 9]. There are some work that takes the simplification view. For example, in [8], a personal workflow management system is proposed using a model called *Metagraph*. It places the data flow at the center of the process. However, doing so makes it difficult to comprehend the temporal execution sequence of the actions. The cooking graph, proposed in CookRecipe [14] system, not only captures individual actions but also the diverse relationship between the actions. However, it is highly customized for summarizing a cooking recipe. [5] proposes a model that simplifies BPMN constructs to only include actions and sequential and parallel action flows. This model does not consider data items in the process.

Our work shares the same view that personal processes do not require complicated expressive power, for example, a parallel execution has no practical meaning when a process is viewed from a single person (i.e., single executor) view point. We also argue that describing data items explicitly is just as important as describing actions to support a useful set of analysis techniques over the personal process model. Therefore our work puts equal emphasis on both. This

is a point that is often overlooked in all modelling languages, even the ones that are designed for personal processes such as the work mentioned above.

**Querying Processes.** Querying business process models represented as graphs is one of graph processing applications that has attracted significant attention (e.g. [4], [2], [3], [13], [6]). The Business Process Query Language BPQL in [4] works on an abstract representation of BPEL<sup>1</sup> files, which focuses on querying actions only. The BPMN-Q is a visual language to query repositories of process models [2]. It processes the queries by matching a process model graph, converted from BPMN, to a query graph [12]. Our PPDG describes personal processes directly as graph (although mapping from/to BPMN is possible) and uses a query paradigm which takes both actions and data into consideration. The approach presented in [3] is based on the notion of partial process models which describe a desired model through a combination of process model fragments and process model queries. Complex query of PPDG has similar structure as partial process models, but the approach in [3] can not be used to perform PPDG queries because PPDG queries need to process both control and data flows. In [10], authors propose the exact subgraph matching approach of assembling graphs to provide answers to a given query graph if no single candidate graph is isomorphic with the query. Another aggregated graph search paper [13] introduces a novel approach for querying and reusing knowledge contained in business process models repositories, which presents the solution for the similar subgraph matching. Due to the structure of PPDG and flexible attribute of personal processes, the above two approaches are not suitable for applying directly to query PPDG graph repositories.

## 7 Conclusion

In this paper, we present Personal Process Description Graphs (PPDG) for describing personally experienced processes. PPDG is the first step towards providing a solution to sharing the process knowledge through a personal process repository, querying and analyzing personal processes and reusing the processes (either as a whole or fragments). Our immediate future work includes improving the query processing algorithms by introducing more punning rules, and utilizing other types of DBMS, e.g. graph database, further developing the algorithms to perform similarity and aggregation matches in PPDG repository.

## References

1. <http://www.bpmn.org/>.
2. A. Awad. BPMN-Q: a language to query business processes. In *EMISA*, pages 115–128, 2007.
3. A. Awad, S. Sakr, M. Kunze, and M. Weske. Design by selection: A reuse-based approach for business process modeling. In *ER*, pages 332–345, 2011.
4. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB*, pages 343–354, 2006.

<sup>1</sup> <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>

5. M. Brambilla. Application and simplification of BPM techniques for personal process management. In *BPM Workshops*, pages 227–233, 2012.
6. R. M. Dijkman, M. Dumas, B. F. van Dongen, R. Käärik, and J. Mendling. Similarity of business process models. *Inf. Syst.*, 36(2):498–516, 2011.
7. S. A. Hajimirsadeghi, H.-Y. Paik, and J. Shepherd. Processbook: Towards social network-based personal process management. In *BPM Workshops*, pages 268–279, 2012.
8. S.-Y. Hwang and Y.-F. Chen. Personal workflows: Modeling and management. In *Mobile Data Management*, pages 141–152, 2003.
9. S. W. Kim, H. Paik, and I. Weber. Automating form-based processes through annotation. In *ICSOC*, pages 558–565, 2012.
10. T. Le, H. Elghazel, and M. Hacid. A relational-based approach for aggregated search in graph databases. In *DASFAA*, pages 33–47, 2012.
11. R. Lu and S. W. Sadiq. A survey of comparative business process modeling approaches. In *BIS*, pages 82–94, 2007.
12. S. Sakr and A. Awad. A framework for querying graph-based business process models. In *WWW*, pages 1297–1300, 2010.
13. S. Sakr, A. Awad, and M. Kunze. Querying process models repositories by aggregated graph search. In *BPM Workshops*, pages 573–585, 2012.
14. L. Wang. *CookRecipe: towards a versatile and fully-fledged recipe analysis and learning system*. PhD thesis, City University of Hong Kong, 2008.
15. I. Weber, H. Paik, and B. Benatallah. Form-based web service composition for domain experts. *TWEB*, 8(1):2, 2013.