Computing Degree of Parallelism for BPMN Processes*

Yutian Sun and Jianwen Su

Department of Computer Science,
University of California, Santa Barbara
{sun,su}@cs.ucsb.edu

Abstract. For sequential processes and workflows (i.e., pipelined tasks), each enactment (process instance) only has one task being performed at each time instant. When a process allows tasks to be performed in parallel, an enactment may have a number of tasks being performed concurrently and this number may change in time. We define the "degree of parallelism" of a process as the maximum number of tasks to be performed concurrently during an execution of the process. This paper initiates a study on computing degree of parallelism for three classes of BPMN processes, which are defined based on the use of BPMN gateways. For each class, an algorithm for computing degree of parallelism is presented. In particular, the algorithms for "homogeneous" and acyclic "choice-less" processes (respectively) have polynomial time complexity, while the algorithm for "asynchronous" processes runs in exponential time.

1 Introduction

There has been an increasing interest in developing techniques for supporting business processes in research communities (e.g., recent conferences/workshops including BPM, COOPIS, ICSOC, ...). A business process is an assembly of tasks (performed by human or systems) to accomplish a business goal such as handling a loan application, approving a permit or treating a patient. The emergence of data management tools in the early 1980's brought the concept of workflow systems to assist execution of business processes in an ad hoc manner. IT innovations in the last decade have been exerting a growing pressure to increase automation in design, operation, and management of business processes. Recent research in this area focused on modeling approaches (e.g., [12,3,21,22,10,1]), verifying properties of business processes and workflow (e.g., [19,20,7]), etc. In this paper, we study the problem of computing the maximum number of tasks that are to be performed in parallel, which can provide useful information to execution planning for processes or workflow [14,24].

Performing business tasks requires resources [16,17] including data, software systems, devices, and in particular human. Resource planning is essential in business process (and workflow) execution management. For processes (workflow) with sequentially arranged tasks (i.e., pipelined tasks), each process instance has at most one task to be performed at one time; the amount of resources needed can be roughly determined

* Work supported in part by NSF grant IIS-0812578 and a grant from IBM.

G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (Eds.): ICSOC 2011, LNCS 7084, pp. 1–15, 2011.
© Springer-Verlag Berlin Heidelberg 2011

by the process initiation rate, the number of tasks in the process, and the amount of work needed for each task. In this paper we focus on calculating the number of tasks that are performed simultaneously, this information provides a needed input to resource estimation.

When a process allows tasks to be performed in parallel, an enactment (process instance) may have a number of tasks to be performed concurrently and this number may change in time. We introduce a new concept "degree of parallelism" as the maximum number of tasks to be performed concurrently during a process execution, i.e., the peak demand on tasks to be performed. This paper initiates a study on computing degree of parallelism for business processes specified in BPMN [4].

Degree of parallelism is a worst case metric for business processes and can provide useful guidance to process modeling and execution planning. For example, some processes may have unbounded degrees, i.e., their peak use exceeds any fixed number. It is quite likely that such processes are results of modeling mistakes. More importantly, the peak time information on tasks could help in planning the needed resources (including human) for the execution of defined business processes.

Technically, this paper defines a formal model for processes (or workflows). The core building blocks in the model are adopted and/or generalized from BPMN constructs; in addition, our model also incorporates an expected duration for each task. The semantics resembles the Petri nets based semantics presented in [8]. The modeling of task durations makes the model closer to real world processes, e.g., healthcare treatment protocols (processes). We formally define the notion of degree of parallelism on this model, and present the following new technical results on three new subclasses of processes based on different combinations of BPMN gateways:

1. For "homogeneous" processes (that use only one type of gateways) a polynomial time algorithm is developed that computes the degree of parallelism.
2. For acyclic "choice-less" processes that does not allow choice gateways nor cycles, we present a polynomial time algorithm for processes in this subclass (the time complexity is linear in the sum of all task durations in the process).
3. We also consider asynchronous processes that use only two types of BPMN gateways: exclusive-merge and parallel-split. By mapping such processes to "process graphs", we show an algorithm to compute the degree of parallelism. The complexity of the algorithm is exponential time in general, but quadratic if the process contains at most one cycle. The general case solution answers an open problem in [13], and the quadratic result improves the cubic time result in [13].

This paper is organized as follows. The formal model and the key notion of degree of parallelism for processes are presented in Section 2. Sections 3, 4, and 5 focus on homogeneous, acyclic choice-less, and asynchronous processes, respectively. Related work is discussed in Section 6, and conclusions are included in Section 7.

2 A Formal Model for Processes

In this section, we introduce a formal model for BPMN processes (or workflows), the key notions include "process", "(pre-)snapshot", "derivation", and "reduction".

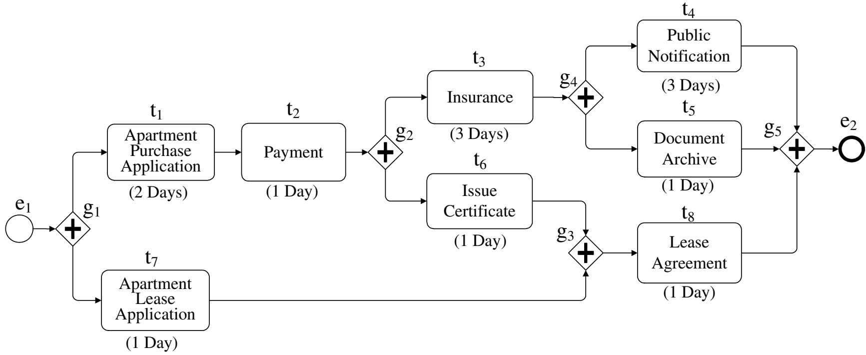


Fig. 1. A BPMN process

The semantics resembles the Petri net semantics for BPMN are presented in [8]. We also define the central notion of "degree of parallelism" used in this paper.

In BPMN [4], a process is modeled as a graph whose nodes and edges are of different types. In this paper, we focus on one type of edges corresponding to "sequence flow" in BPMN, and three types of nodes: "event", "task", and "gateway".

We consider two classes of *events* in BPMN: *start* and *end* events. A start (event) node initializes a process by sending out a "flow front" (or an active point of execution) to the next node. When all flow fronts reach their end (event) nodes, the process ends. A *task* (node) represents an atomic unit of work.

A *gateway* node in a process alters the current execution path (e.g., by choosing an alternative path or proceeding on all paths). There are four kinds of frequently used gateways in BPMN: (exclusive-)*choice*, (exclusive-)*merge*, (parallel-)*split* and (parallel-) *join*. Choice and merge gateways allow a flow front in a process to follow one of several alternatives (choice) or choose only one flow front from possibly several incoming edges to continue (merge). Split and join gateways, on the other hand, forward a flow front to every outgoing edge for parallel execution (split) or synchronize flow fronts from all incoming edges and combine them into one (join).

Example 2.1. Fig. 1 shows an example BPMN process, which combines purchasing an apartment and putting it out for lease (in China where leasing arrangements need an approval from the city office for real estate management). The process begins from a start event (e_1) and immediately forks into two paths by a split gateway g_1. The upper and lower paths represent the purchase and lease sub-processes respectively. The applicant files the purchase (t_1) and lease (t_7) applications. The expected duration of each task is shown below the task node in the figure, e.g., t_1 would take 2 days while t_7 only 1 day. After paying a purchasing transaction fee (t_2), the applicant obtains the certificate (t_6). Together with the leasing application, the applicant can finish a lease contract with the tenant (t_8). For the other branch, the applicant also needs to spend 3 days on getting an insurance policy (t_3). Once this is done, the housing office will make a public notification (t_4) for additional 3 days as required by law before archiving all the documents (t_5). Finally, all flow fronts will synchronize at the join gateway g_5; the process will end when reaching end event e_2. ∎

In our formal model, a "process" is a graph with edges corresponding to control-flow transitions and nodes representing start/end events, tasks, or gateways. Similarly, a gateway node alters the execution path (choosing an alternative path or following parallel paths). Our model includes two kinds of gateways that are more general than BPMN gateways: *exclusive* (denoted as ⊗) and *parallel* (denoted as ⊕). A ⊗-gateway essentially combines a merge gateway and a choice gateway, and a ⊕-gateway is a join gateway followed by a split gateway. Specifically, a ⊗-gateway node passes an incoming flow front on an incoming edge immediately to one of the outgoing edges to continue the flow front. A ⊕-gateway node, on the other hand, waits until one flow front from each incoming edge arrives, merges them into one flow front, and then split it again to pass a flow front to each of its outgoing edges. Note that when the number of incoming/outgoing edges is 1, ⊗- and ⊕-gateways degenerate to BPMN gateways.

Our model associates a duration to each node to indicate the typical length for the node to complete. Without loss of generality, the duration of each gateway or event node is always 0 (it takes no time to complete). A task node takes some time (> 0) to perform before finishing. In Fig. 1, durations are shown in parentheses below task nodes.

For the technical development, we use *indeg* and *outdeg* to denote the number of incoming edges and outgoing edges of a node respectively. Let **T** be the set consisting of the following types: ○ (start), ● (end), □ (task), ⊗ (exclusive gateway), and ⊕ (parallel gateway). Let $\mathbb{N}$ be the set of natural numbers.

Definition: A *process* (*with durations*) is a tuple $P = (V, s, F, E, \tau, \delta)$, where

- V is a (finite) set of *nodes*,
- $s \in V$ and $F \subseteq (V - \{s\})$ are the *start* node and a set of *end* nodes (resp.),
- $\tau : V \to \mathbf{T}$ is a mapping that assigns each node in V a type such that $\tau(s) =$ "○", $\tau(v) =$ "●" for each $v \in F$, and $\tau(v)$ is not "○" nor "●" for each $v \in V - F - \{s\}$,
- $\delta : V \to \mathbb{N}$ is a mapping that assigns each node a *duration* such that for each $v \in V$, $\delta(v) > 0$ iff $\tau(v) =$ "□" (v is a task node), and
- $E \subseteq (V - F) \times (V - \{s\})$ is a set of *transitions* satisfying all conditions listed below:
  1. For the start node s, $outdeg(s) = 1$ and $indeg(s) = 0$,
  2. For each end node $v \in F$, $indeg(v) = 1$ and $outdeg(v) = 0$, and
  3. For each task node v, $indeg(v) = outdeg(v) = 1$.

Given a process $P = (V, s, F, E, \tau, \delta)$, a *cycle* (*of size* $n \in \mathbb{N}$) is a sequence $v_1, v_2, ..., v_n$ such that for each $i \in [1..n]$, v_i is a node in V, and $(v_i, v_{(i \bmod n)+1}) \in E$. A process is *acyclic* if it contains no cycles.

The graph shown in Fig. 1 can also be viewed as a process in our model, where e_1 and e_2 are the start and end nodes (resp.), g_i's ($1 \leqslant i \leqslant 5$) are ⊕-gateway nodes, and t_i's ($1 \leqslant i \leqslant 8$) are task nodes with non-0 durations.

In general, a process can be nondeterministic and/or have tasks performing in parallel. For example, if a flow front goes into a ⊗-gateway, the gateway can choose nondeterministically an outgoing edge to route the flow front. Also, a process can spawn several flow fronts during the execution due to ⊕-gateway nodes. The goal of this paper is to compute the maximum number of tasks that may run in parallel.

In order to define the notions precisely for algorithm development, we need to provide a semantics for processes. We introduce a pair of notions "pre-snapshots" and "snapshots" below that are used to formulate the semantics.

In the remainder of this section, let $P = (V, s, F, E, \tau, \delta)$ be some process. A *flow front* is a triple (u, v, n), where $(u, v) \in E$ is an edge (transition) in P and n is a (possibly negative) integer such that $n \leqslant \delta(u)$. Intuitively, a positive number n denotes the remaining time needed to complete the node u or "time-to-live" for u. When $n \leqslant 0$, u is completed and the flow front is ready to move forward through v in the process. Since the duraton of a non-task node is always 0, a flow front (u, v, n) originating at a non-task node u can proceed unless v is a $\oplus$-gateway node.

A *pre-snapshot* of the process P is a multiset of flow fronts. Note that duplicates are allowed in a pre-snapshot. The singleton multiset $\{(s, u, 0)\}$ is an *initial* pre-snapshot where s is the start node. Given a pre-snapshot S of P, a node v in P is *ready* (*to activate*) in S if one of the following holds:

- v is an end/task/$\otimes$-gateway node and a flow front (u, v, n) is in S for some $n \leqslant 0$, or
- v is a $\oplus$-gateway node and for each incoming edge (u, v) into v, (u, v, n) is a flow front in S for some $n \leqslant 0$.

Example 2.2. Consider the process shown in Fig. 1. The triples $(e_1, g_1, 0)$, $(t_1, t_2, 2)$ are flow fronts of the process, $(e_1, g_1, 1)$ is not a flow front since the duration of e_1 is $0 < 1$, nor is $(t_1, t_3, 2)$ since (t_1, t_3) is not an edge in the process. The following are pre-snapshots: $\{(e_1, g_1, 0)\}$, $\{(g_1, t_1, -2)\}$, $\{(t_1, t_2, 2), (t_7, g_3, 1)\}$, and also $\{(t_1, t_2, 2),$ $(t_2, g_2, -1), (t_7, g_3, 1)\}$. In the pre-snapshot $\{(t_1, t_2, 2), (t_2, g_2, -1), (t_7, g_3, 1)\}$, task t_2 has completed, tasks t_1 and t_7 are still running in parallel, and node g_2 is ready. ∎

If a node v is ready in a pre-snapshot S, we can proceed a (or more) flow front(s) to the node v to *derive* a new pre-snapshot S' as follows.

- If v is an end node and (u, v, n) is in S where $n \leqslant 0$, then $S' = S - \{(u, v, n)\}$.
- If v is a task or $\otimes$-gateway node and (u, v, n) is in S where $n \leqslant 0$, then $S' = (S - \{(u, v, n)\}) \cup \{(v, w, \delta(v))\}$ where $\delta(v)$ is the duration of v and (v, w) is an edge leaving v in P. (When v is $\otimes$-gateway, w is nondeterministically selected.)
- If v is a $\oplus$-gateway node with all incoming edges from $u_1, ..., u_\ell$ and for each $i \in [1..\ell]$, (u_i, v, n_i) is in S where $n_i \leqslant 0$, then $S' = (S - \{(u_i, v, n_i) \mid 1 \leqslant i \leqslant \ell\}) \cup \{(v, w_i, \delta(v)) \mid (v, w_i)$ is an outgoing edge of v in $P\}$.

Example 2.3. Consider the process in Fig. 1. Since e_1 is the start node, the initial pre-snapshot is $\{(e_1, g_1, 0)\}$. Clearly, g_1 is ready and we can derive the pre-snapshot $\{(g_1, t_1, 0), (g_1, t_7, 0)\}$ since g_1 is a $\oplus$-gateway. Now both t_1 and t_7 become ready. We may derive the pre-snapshots $\{(t_1, t_2, 2), (g_1, t_7, 0)\}$, and then $\{(t_1, t_2, 2), (t_7, g_3, 1)\}$. At this point, no nodes are ready. ∎

We call a pre-snapshot of process P in which no nodes are ready a *snapshot*. In Example 2.3, $\{(t_1, t_2, 2), (t_7, g_3, 1)\}$ is a snapshot. In general, a pre-snapshot can always derive in a finite number of steps into a snapshot. We call the procedure of a pre-snapshot eventually deriving a snapshot a *reduction*.

From a snapshot, derivations cannot be made since no nodes are ready. At this time we can advance process operations by one time unit. Technically, let S be a snapshot and S' a pre-snapshot. S *task-derives* S' if $S' = \{(u, v, n-1) \mid (u, v, n) \in S\}$.

Note that if a flow front has a positive time-to-live the time is decremented by 1, if the time-to-live is zero or negative, the resulting time may be negative. While this may be a useful information for measuring performance, we do not use the negative amounts in this paper. Also, as the task-derivation indicates, the scheduling algorithm for process tasks is an eager one—it performs the task immediately when the task becomes ready. It is interesting to examine alternative scheduling policies and explore their impact on, e.g., the degree of parallelism. But this is beyond the scope of the present paper.

Example 2.4. Continuing with Example 2.3, the snapshot $\{(t_1, t_2, 2), (t_7, g_3, 1)\}$ task-derives the pre-snapshot $\{(t_1, t_2, 1), (t_7, g_3, 0)\}$. The latter indicates that t_7 is completed but g_3 is not ready since it is a ⊕-gateway and the other incoming edge does not have a flow front. Therefore, $\{(t_1, t_2, 1), (t_7, g_3, 0)\}$ is also a snapshot, which further task-derives $\{(t_1, t_2, 0), (t_7, g_3, -1)\}$. Now task t_2 becomes ready. ∎

Definition: Let $n \in \mathbb{N}$ and $P = (V, s, F, E, \tau, \delta)$ be a process. An *enactment* of length n of P is a sequence $p = S_1 S_2 ... S_n$ such that for each $i \in [1..n]$, S_i is a snapshot, S_1 is a reduction from the initial pre-snapshot and for each $i \in [2..n]$, S_i is obtained from S_{i-1} by first applying task-derivation on S_{i-1} followed by a reduction. The enactment p is *complete* if $S_n = \emptyset$. The *semantics* of a process P is a set of all complete enactments.

Let S be a snapshot, the *active cardinality* of S, denoted as $|S|_{\text{active}}$, is the cardinality of the multiset $\{(u, v, n) \mid (u, v, n) \in S \text{ and } n \geqslant 1\}$. The active cardinality of S indicates the number of (currently) running tasks at the time of the snapshot.

Definition: The *degree (of parallelism)* of a process P, denoted as $\text{DP}(P)$, is the maximum active cardinality of a snapshot in some enactment of P.

The degree of process P reflects how much parallelism the execution of P allows, i.e., the maximum number of (active) flow fronts that can appear during the execution of P. Suppose the total amount of "work" in P is fixed. The greater $\text{DP}(P)$ is, the more resources operations of P will need. On the other hand, the availability of these resources will mean the total time to complete an enactment is shorter. This, however, does not mean the throughput of the business managing process P is automatically higher. To achieve operational efficiency under resource limitation, it may be possible to plan tasks in P in a way to lower the degree of parallelism while maintaining the throughput. The study on the degree of parallelism is an initial step towards understanding the issue of resource needs and constraints on tasks as specified in a process.

3 Homogeneous Processes

In this section, we focus on a subclass of processes, called "homogeneous processes", and present a polynomial time algorithm to compute the degree of parallelism.

A process is *homogeneous* if its gateway nodes only use one kind of gateway, either ⊗-gateway or ⊕-gateway, but not both. There are two flavors of homogeneous processes. A *parallel*-(or ⊕-)homogeneous process uses only ⊕-gateway while a *choice*-(or ⊗-)homogeneous process uses only ⊗-gateway.

Lemma 3.1. The degree of parallelism for each $\otimes$-homogeneous process is always 1.

From the semantics, it is easy to see that derivation and task-derivation from a pre-snapshot will not increase the cardinality, since at most one outgoing edge (transition) can be invoked for each node. Since the initial pre-snapshot only contains one element, the cardinality of each snapshot of each arbitrary $\otimes$-homogeneous process is always one, which bounds the degree of parallelism. The proof can be done by an induction.

Obviously, Lemma 3.1 fails for $\oplus$-homogeneous processes. In the remainder of this section, we only focus on the calculation of the degree of parallelism of $\oplus$-homogeneous processes. The process in Fig. 1 is a $\oplus$-homogeneous process.

Given a process P, a node v is *reachable* in P, if there exists an enactment $S_1S_2...S_k$, such that v is ready either in the snapshot S_k, or in a pre-snapshot that can be derived from S_{k-1} and reduced to S_k.

Lemma 3.2. If every node in a $\oplus$-homogeneous process P is reachable, P is acyclic.

Proof: (Sketch) Let P be a $\oplus$-homogeneous process that contains a cycle C. Consider a sequence of pre-snapshots $S_1, ..., S_m$ such that (1) S_1 is initial, and for each $i \in [2..m]$, S_i is derived or task-derived from S_{i-1} in one step, (2) for some node v in C, v is ready in S_m, and (3) no other nodes in C that is ready in S_j for $j < m$. Since each node in P is reachable, it is possible to find a v and pre-snapshot sequence that satisfy (1)-(3). Clearly, v must have at least two incoming edges (one from the path and the other on the cycle) and thus a $\oplus$-gateway node. By the definition of derivition/task-derivition, some node on C must be ready in S_j for some $j < m$, a contradiction. ∎

Since a $\oplus$-homogeneous process is acyclic according to Lemma 3.2, each node will be added into a snapshot or pre-snapshot at most once. Thus, the degree of parallelism is finite and less than the total number of task nodes in the process.

Theorem 3.3. Given a $\oplus$-homogeneous process $P = (V, s, F, E, \tau, \delta)$, the degree of parallelism of P can be computed in $O(|V| \log |V|)$ time.

To establish Theorem 3.3, we develop an algorithm that simulates the execution of a process $P = (V, s, F, E, \tau, \delta)$ in computing its degree of parallelism. The simulation uses a priority queue to store all nodes that are currently running. When a node finishes, it is popped from the queue. Thus, the degree of the $\oplus$-homogeneous process is the maximum number of tasks that appear in the queue at some point during the simulation.

We use $[v, t]$ to denote an element in Q where t is the completion time for a node v. Entries of form $[v, t]$ in Q, are sorted according to the completion time t in the ascending order. The algorithm (Alg. 1) uses an array $RN(v)$ to record the number of remaining incoming nodes that haven't finished but are necessary for v to be performed.

Alg. 1 starts by placing $[s, 0]$ in Q which is analogous to the initial pre-snapshot. The array $RN(v)$ is initialized to $indeg(v)$ for each v. Every time an element $[v, t]$ is popped from Q indicates the completion of v at time t. (Since elements in Q are sorted by their end times, the one at the front of Q always has the earliest end time.) Once a node v finishes, the algorithm checks if there is an edge connecting from v to u, and if so $RN(u)$ is decremented by 1. If $RN(u)$ becomes 0, u starts to execute and therefore will

Algorithm 1. Compute Degree of a Parallel-Homogeneous Process

Input: A process $P = (V, s, F, E, \tau, \delta)$
Output: degree of parallelism $\text{DP}(P)$

```
 1: Initialize a priority queue Q to be empty;
 2: Q.enque([s, 0]);
 3: for each v ∈ V − {s} do
 4:     RN(v) := indeg(v);
 5: end for
 6: deg := 0;
 7: t_old := 0;
 8: while Q is not empty do
 9:     [v1, t1] := Q.deque();
10:     for each v2 ∈ {v | (v1, v) ∈ E} do
11:         RN(v2) := RN(v2) − 1;
12:         if RN(v2) = 0 then
13:             Q.enque([v2, t1 + δ(v2)]);
14:         end if
15:     end for
16:     if t_old ≠ t1 then
17:         #T := |{[v, t] | [v, t] ∈ Q ∧ v is a task node }|;
18:         deg :=max(deg, #T);
19:         t_old := t1;
20:     end if
21: end while
22: return DP(P) = deg;
```

Step	Q	Updated RN	$\#T$	DP	Step	Q	Updated RN	$\#T$	DP
1	$[e_1, 0]$		0	0	9	$[t_8, 5], [t_3, 6]$	$RN(t_8) = 0$	2	2
2	$[g_1, 0]$	$RN(g_1) = 0$	0	0	10	$[t_3, 6]$	$RN(g_5) = 2$	1	2
3	$[t_7, 1], [t_1, 2]$	$RN(t_7) = RN(t_1) = 0$	2	2	11	$[g_4, 6]$	$RN(g_4) = 0$	0	2
4	$[t_1, 2]$	$RN(g_3) = 1$	1	2	12	$[t_5, 7], [t_4, 9]$	$RN(t_5) = RN(t_4) = 0$	2	2
5	$[t_2, 3]$	$RN(t_2) = 0$	1	2	13	$[t_4, 9]$	$RN(g_5) = 1$	1	2
6	$[g_2, 3]$	$RN(g_2) = 0$	0	2	14	$[g_5, 9]$	$RN(g_5) = 0$	0	2
7	$[t_6, 4], [t_3, 6]$	$RN(t_6) = RN(t_3) = 0$	2	2	15	$[e_2, 9]$	$RN(e_2) = 2$	0	2
8	$[g_3, 4], [t_3, 6]$	$RN(g_3) = 0$	1	2	16	$\varnothing$		0	2

Fig. 2. Simulating the process in Example 2.1

be pushed into Q. During the simulation, let $\#T$ be the number of task nodes in Q. In all, the degree $\text{DP}(P)$ is the highest $\#T$ that appears during the entire simulation.

Fig. 2 illustrates the details of simulating the process in Example 2.1. It turns out that the degree of parallelism is 2 even though the process has 3 parallel branches (Fig. 1).

The complexity of Alg. 1 depends on the time to maintain the priority queue. Since the size of the queue can be at most $|V|$, the complexity of this algorithm is $O(|V| \log |V|)$.

4 Acyclic Choice-Less Processes

In this section, we introduce another subclass of BPMN processes, acyclic "choice-less processes" and focus on the computation of degree of parallelism for such processes. We present a polynomial time algorithm to compute the degree. Note that for acyclic processes, the degree is always finite.

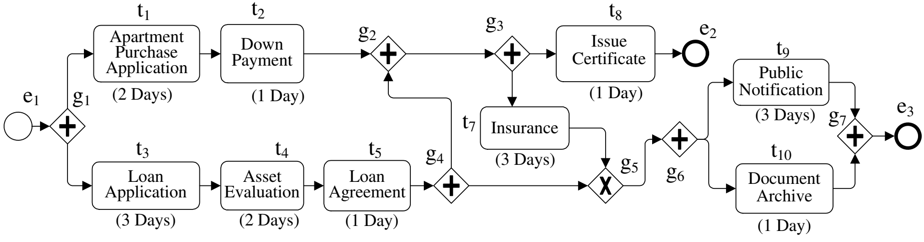


Fig. 3. An acyclic choice-less process

Definition: A process $P = (V, s, F, E, \tau, \delta)$ is *choice-less* if for each $v \in V$, $outdeg(v) = 1$ whenever $\tau(v) =$ "⊗".

Intuitively, a choice-less process contains no exclusive-decision gateway nodes. But it may contain exclusive-merge gateway (with one outgoing edge). These processes are used frequently in scientific workflows [2], where the focus is on computations that involve large amounts of datasets. Knowing the degree of parallelism of a scientific workflow would potentially help scheduling computations (i.e. tasks), especially for a cloud computation setting [11].

Example 4.1. Fig. 3 shows an example of acyclic choice-less process for purchasing an apartment with loan. The process begins with two branches: to apply for the apartment purchase (t_1) and pay the down payment (t_2), and to apply for the loan (t_3). After assessing the apartment (t_4), the bank decides to pay the rest of the balance (t_5) to complete the purchase (g_2). Once the loan is settled, the housing office will archive the documents (t_{10}) and make a public notification (t_9). Also, the office will give the certificate to the buyer (t_8) for the new ownership. After the customer purchases the insurance (t_7), the housing office will again archive the documents (t_{10}) and make a public notification (t_9). Note that t_{10} is invoked twice due to the presence of a ⊗-gateway (g_5). ∎

Theorem 4.2. Given an acyclic choice-less process $P = (V, s, F, E, \tau, \delta)$, the degree of parallelism of P can be computed in $O(|E| \log |V| + |E|L)$ time where L is the sum of durations of all task nodes in P.

In the remainder of this section, we discuss key ideas for proving Theorem 4.2. More details are provided in the online appendix [18].

The key idea to compute the degree of an acyclic choice-less process is to partition the process into smaller pieces, analyze the pieces, and then aggregate them together. We view each ⊕-gateway node as a pair of BPMN split and join gateways, all ⊗-gateway nodes are actually merge gateways due to the choice-less restriction. The following are the 3 main steps:

1. Decompose the process into segments according to join and merge gateway nodes.
2. For each segment, a list is computed to capture the parallelism information.
3. Combine all such lists and compute the degree for the input process.

In the ***first*** step, a process is chopped into segments. Each segment is separated by join and merge gateway nodes. To generate a segment, a depth-first search is used. We create

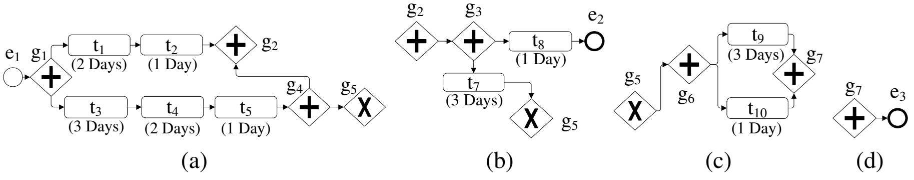
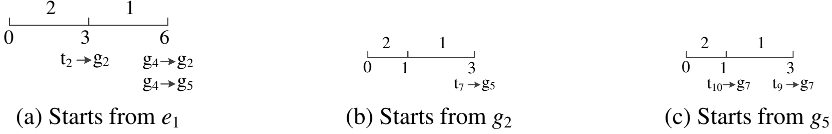


Fig. 4. Four segments of the process in Fig. 3



(a) Starts from e_1 (b) Starts from g_2 (c) Starts from g_5

Fig. 5. Event point list

a new segment by traversing from the start node. when a join or merge gateway node is visited, it is marked as an *exit node* for the current segment, and starts a new segment. The node also plays the role of the *entry node* of the new segment. Since a join or merge gateway node has only one outgoing edge, each segment has only one entry node and may have several exit nodes.

Example 4.3. Fig. 4 shows all four segments of the process in Fig. 3. Fig. 4(a) has the entry node e_1 and two exit nodes g_2, g_5. Fig. 4(b) starts from g_2 and ends at e_2, g_5. Fig. 4(c) enters at g_5 and has one exit node g_7. Fig. 4(d) starts from g_7 and ends at e_3. ▮

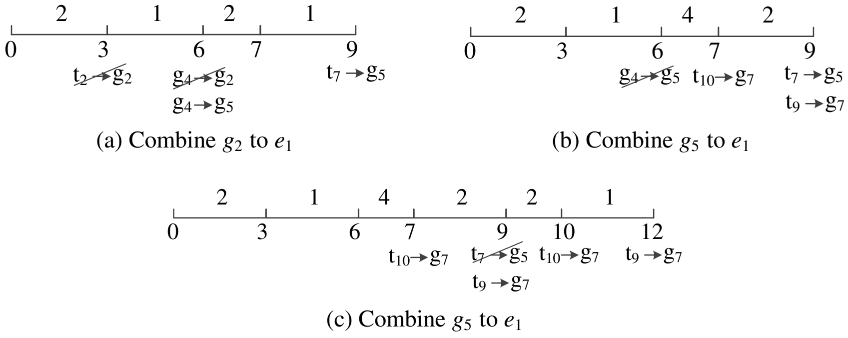
In the ***second*** step, we compute the "parallelism" information of each segment, with a data structure *event point list*. An event point list contains two basic pieces of information: (1) the cardinality of the corresponding segment's enactment between two timestamps, and (2) the time the segment will reach its exit nodes and through which edge the segment will reach each exit node.

Example 4.4. Fig. 5 shows three event point lists generated according to the segments in Example 4.3. Fig. 5(a) corresponds to Fig. 4(a). From time 0 to 3, the degree is 2, then t_2 completes and invokes g_2. From time 3 to 6, only one flow front exists, and at timestamp 6, g_4 invokes g_2 and g_5. Fig. 5(b)(c) provide the similar event point lists corresponding to Fig. 4(b)(c), resp. The event point list of Fig. 4(d) is an empty list. ▮

Constructing event point lists is similar to the algorithm in Section 3. By mapping each entry node to an start node and each exit node to an end node, each segment is in fact a homogeneous process. Similar to Alg. 1, a priority queue can be used to simulate each segment. And the event point list can be derived according to $\#T$. When an exit node is popped out from the queue, this node, together with its incoming edge, will be recorded in the event point list.

In the remainder of this section, we may use term "event point list" and "segment" interchangeably to refer to the same object according to the context.

Once all event point lists are constructed, the ***third*** step combines the lists. Since the choice-less process is acyclic, a key observation is that all segments follow a topological order, i.e., a segment can only be invoked by its preceding segments.



(a) Combine g_2 to e_1

(b) Combine g_5 to e_1

(c) Combine g_5 to e_1

Fig. 6. Combination of event point lists

With the sorted segment sequence, we remove the second segment and combine its event point list into the first event point list. Note that this guarantees that the second segment can only be invoked by one segment, i.e., the first. This procedure repeats until only one event point list left in the end.

There are two types of event point lists to be combined. One starts from a join gateway (node) and the other from a merge gateway. The combination of these two types of event point lists to the first event point list need be handled differently.

If the second event point list's entry node is a join gateway, we first mark where this segment is invoked in the first event point list according to each different incoming edges from left to right. once all different incoming edges are marked, we combine the second event point list to the first one at the last timestamp where an edge is marked. Then we repeat the above steps until the second event point list cannot be combined any more. The reason to mark incoming edges is to simulate the synchronization property of join gateway. A join gateway can only continue once all its incoming edges are ready.

Example 4.5. The segment order for Example 4.4 is e_1, g_2, g_5, g_7. Now consider the second event point list that starts from join gateway g_2. Since g_2 has two incoming edges, (t_2, g_2) and (g_4, g_2), in the first event point list, we mark $t_2 \rightarrow g_2$ and $g_4 \rightarrow g_2$ and then combine the second event point list at time 6. Fig. 6(a) is the new event point list starts from e_1 and g_2 (segment) should be removed from the segment sequence. ∎

If the second event point list's entry node is a merge gateway, the combination is simpler. Since for each incoming edge of this kind of node, once a flow front arrives, the node immediately routes it to its outgoing edge. Thus when scanning the first event point list, once at some timestamp, the second segment is invoked, we can simply do the combination.

Example 4.6. After merging g_2 to e_1 in Example 4.5, the segment sequence is e_1, g_5, g_7. Now the second event point list starts from the merge gateway g_5. In the first event point list (Fig. 6(a)), there are two places that call g_5. Hence, two combinations are needed. Fig. 6(b) and (c) show the first and second combination respectively.

Now the only event point list left is the one with entry node g_7. Since g_7 leads an empty event point list, the final event point list is the same as the one in Fig. 6(c). ∎

The algorithm details are provided in the online appendix [18].

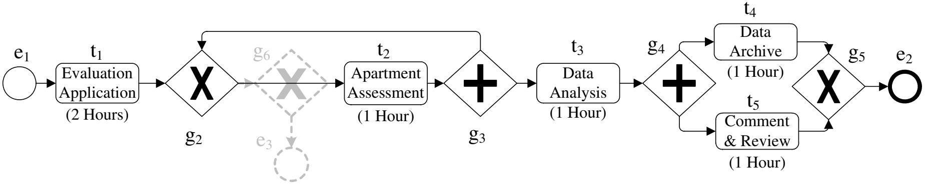


Fig. 7. Pre-sell permit approval process

5 Asynchronous Processes

In this section, we introduce the third subclass of BPMN processes, called "asynchronous processes", and present an algorithm to compute their degrees. Intuitively, an asynchronous process only includes split and merge gateways, i.e., it cannot do synchronization nor choices. It turns out that computing degree of parallelism for such processes is rather intricate, the time complexity of the algorithm is exponential.

Definition: A process $P = (V, s, F, E, \tau, \delta)$ is *asynchronous* if for each node $v \in V$, *outdeg*$(v) = 1$ whenever $\tau(v) =$ "$\otimes$" and *indeg*$(v) = 1$ whenever $\tau(v) =$ "$\oplus$".

From the definition, an asynchronous process includes only gateways nodes that are split gateway or merge gateway.

Example 5.1. Fig. 7 shows a process for apartment evaluation. If a developer is building apartments and plans to sell them, she needs a "pre-sell" permit from the city housing office. The office checks if the apartments are in good quality. An apartment quality evaluation process will start when an application (t_1) is received. Then the office staff will assess each apartment. If there is no more apartment to check, the process will end at g_6 and exit to e_3. Otherwise, evaluation modes to the next apartment (t_2). Once an apartment is assessed, the data will be send to the housing office asynchronously for analysis (t_3). After that, comment will be drawn (t_5) and data will be archived (t_4). ∎

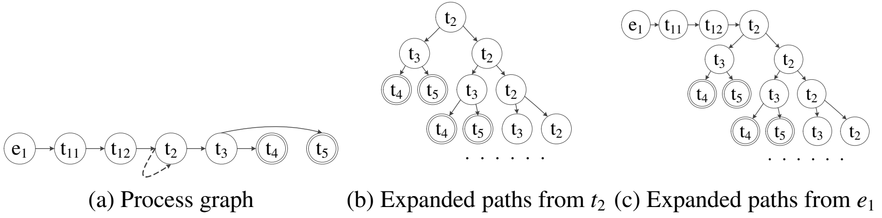
Technically, the process in the above example is not asynchronous due to the decision gateway (g_6). In order to simplify the analysis, we hide it from the process, link an edge directly from g_2 to t_2, and remove e_3 as well.

In the technical development, we use simplified graphs for asynchronous processes.

Definition: A (*process*) *graph* is a tuple (V, E, s, F) where V is a set of nodes containing the initial node s and a set F of final nodes, and $E \subseteq (V-F)\times V$ is a set of edges.

A *path* of size n of an process graph $G = (V, E, s, F)$ is a sequence of nodes $v_1 v_2 ... v_n$, where for each $i \in [1..n]$, $v_i \in V$, $v_1 = s$, and for each $i \in [1..(n-1)]$, $(v_i, v_{i+1}) \in E$. A path denotes one possible execution of the given process graph. However, in order to take all the possible executions into consideration, we pursue all paths in parallel. Let $D_n(G)$ denote the number of distinct paths of G with length n. We define the *degree* of G to be the max $D_n(G)$ for all $n \in \mathbb{N}$.

Lemma 5.2. Each asynchronous process P can be translated into a process graph G, such that the degree of P is the same as the degree of G.



(a) Process graph (b) Expanded paths from t_2 (c) Expanded paths from e_1

Fig. 8. Process graph and its expansion

Fig. 8(a) shows the process graph translated from the process in Fig. 7. We now can focus on process graphs and compute degree of an asynchronous process by computing degree of its corresponding graph.

A process graph G is said to be *bounded* if the degree of G is finite. The key results of the section are now stated below.

Theorem 5.3. Let P be an asynchronous process whose process graph has n nodes and m edges. Boundedness of degree of P can be decided in $O(m+n)$ time; if the degree is bounded, the degree can be computed in exponential time, and in $O(mn)$ time if P is acyclic or contains only one cycle.

Theorem 5.3 follows from the following two lemmas (Lemmas 5.4 and 5.5).

Lemma 5.4. (1) The degree of a process graph is bounded iff it does not contain two distinct cycles such that one connects to the other. (2) Given a process graph $G = (V, E, s, F)$, its boundedness can be determined in $O(|V| + |E|)$ time.

Lemma 5.4 is a slight variant of a result in [13] (the models are slightly different). Furthermore, given a process graph with at most one cycle (always bounded), an algorithm was presented in [13] to compute the degree in cubic time complexity. However, the general case was left open.

In the remainder of this section, we discuss a new algorithm that makes two improvements over the result in [13]: (1) it computes the degree for the general case, thus solves the open problem from [13], (2) when applying to acyclic and one-cycle graphs, the time complexity is quadratic, which improves the cubic result in [13].

Lemma 5.5. Given a bounded process graph $G = (V, E, s, F)$, the degree of G can be computed in exponential time, and in $O(|V||E|)$ time if G contains at most one cycle.

To compute the degree of a bounded process graph, we use the following steps:

1. Eliminate all cycles of the given process graph. For each node in the new graph, compute the numbers of reachable nodes in different depths and store these numbers in a list, called "*child list*". The method to compute each child list is according to a reversed topological order.
2. Add cycles back to the graph, with the result from step 1, compute the numbers of reachable nodes in different depths for those nodes that are inside cycles. Store these numbers in a list, called "*cycle child list*".

3. Remove all the cycles once more and compute the cycle child list for the source node. The degree of the corresponding process graph is the largest number of this cycle child list.

The detailed algorithm is rather involved and sketched in the online appendix [18].

6 Related Work

Our work is an extension of the work in [13] that focused on non-determinism of a simple graph model. Their model can be mapped to asynchronous processes with their degree of non-determinism coincides with degree of parallelism. The results reported in Section 5 extended their results and solve an open problem.

There were a stream of papers related to degree of non-determinism of finite state machines. These addressed the problems of boundedness [23,15], computing the degree [23,9,15], estimating the upper bound of the degree [23], and complexity bounded on this problem [15,5]. Although the problems are different from ours, it remains to explore whether these techniques can be used in solving our problem.

Our work is also related to workflow execution management. The work in [14] proposed a set of resource patterns for task allocation. While a language for specifying the resource allocation constraints was described in [16,17]. The study in [6] focused on authorization constraints and determining if a workflow can finish under such constraints. Static scheduling issues were studied in [24], where the authors developed an adaptive rescheduling strategy for grid workflow. Finally, while our problem seems relevant to parallel computing, it was not studied in the literature to the best of the authors' knowledge.

7 Conclusions

We focus on a subset of BPMN and examine the worst case number of parallel tasks and develop a set of preliminary results. It is still not clear how one would extend the algorithm to the full set of BPMN. This work also spawns many interesting questions related to planning business process execution. For example, given the resource requirements and cost functions, how can these algorithms be augmented to produce sufficient information for execution planning. Such problems are key to many business processes, e.g., in healthcare delivery. Clearly, this paper merely peeks into a broader topic concerning business operations planning and optimization.

References

1. Abiteboul, S., Segoufin, L., Vianu, V.: Modeling and verifying active xml artifacts. Data Engineering Bulletin 32(3), 10–15 (2009)
2. Barker, A., van Hemert, J.: Scientific Workflow: A Survey and Research Directions. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 746–753. Springer, Heidelberg (2008)

3. Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Towards Formal Analysis of Artifact-Centric Business Process Models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 288–304. Springer, Heidelberg (2007)
4. Business Process Model and Notation (BPMN), version 2.0 (January 2011), `http://www.omg.org/spec/BPMN/2.0/PDF`
5. Chan, T., Ibarra, O.H.: On the finite-valuedness problem for sequential machines. Theoretical Computer Science 23(1), 95–101 (1983)
6. Crampton, J.: A reference monitor for workflow systems with constrained task execution. In: Proc. 10th ACM Symp. on Access Control Models and Technologies, SACMAT (2005)
7. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proc. Int. Conf. on Database Theory (ICDT), pp. 252–267 (2009)
8. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in bpmn. Inf. Softw. Technol. 50, 1281–1294 (2008)
9. Gurari, E.M., Ibarra, O.H.: A note on finite-valued and finitely ambiguous transducers. Theory of Computing Systems 16(1), 61–66 (1983)
10. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath III, F., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculín, R.: Introducing the guard-stage-milestone approach to specifying business entity lifecycles. In: Proc. Workshop on Web Services and Formal Methods (WS-FM). Springer, Heidelberg (2010)
11. Juve, G., Deelman, E.: Scientific workflows and clouds. Crossroads 16(3) (March 2010)
12. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
13. Potapova, A., Su, J.: On nondeterministic workflow executions. In: Proc. Workshop on Web Services and Formal Methods, WSFM (2010)
14. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Workflow Resource Patterns: Identification, Representation and Tool Support. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 216–232. Springer, Heidelberg (2005)
15. Sakarovitch, J., de Souza, R.: On the Decidability of Bounded Valuedness for Transducers. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 588–600. Springer, Heidelberg (2008)
16. Senkul, P., Kifer, M., Toroslu, I.H.: A logical framework for scheduling workflows under resource allocation constraints. In: Proc. 28th Int. Conf. on Very Large Data Bases (2002)
17. Senkul, P., Toroslu, I.H.: An architecture for workflow scheduling under resource allocation constraints. Information Systems 30, 399–422 (2005)
18. Sun, Y., Su, J.: On-line Appendix to the Paper "Computing Degree of Parallelism for BPMN Processes" (2011), `http://www.cs.ucsb.edu/~su/papers/2011/AppendixICSOC2011.pdf`
19. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248. Springer, Heidelberg (1997)
20. van der Aalst, W.M.P.: Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, p. 161. Springer, Heidelberg (2000)
21. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
22. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
23. Weber, A.: On the valuedness of finite transducers. Acta Inf. 27, 749–780 (1990)
24. Yu, Z., Shi, W.: An adaptive rescheduling strategy for grid workflow applications. In: Proc. IPDPS (2007)