A General Business Process Model Query Language based on Execution Semantics

Liang Song[1,2,3,4], Jianmin Wang[1,3,4], Arthur H.M. ter Hofstede[5,6⋆], Marcello La Rosa[5], and Chun Ouyang[5]

[1] School of Software, Tsinghua University, Beijing, China
{songliang08}@mails.thu.edu.cn,jianmin@mail.thu.edu.cn
[2] Department of Computer Science and Technology, Tsinghua University, Beijing, China
[3] Key Lab for Information System Security, Ministry of Education, Beijing, China
[4] National Laboratory for Information Science and Technology, Beijing, China
[5] Queensland University of Technology, Brisbane, Australia
{a.terhofstede,m.larosa,c.ouyang}@qut.edu.au
[6] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. As business process management technology matures, organisations acquire more and more business process models. The resulting collections can consist of hundreds, even thousands of models and their management poses real challenges. One of these challenges concerns model retrieval where support should be provided for the formulation and efficient execution of business process model queries. As queries based on structural information only can not deal with all user requirements, there should be support for queries that require knowledge of process model semantics. In this paper we formally define a process model query language that is based on semantic relationships between tasks. This query language is independent from the particular process modelling notation used, but we will demonstrate how it can be used in the context of Petri nets by showing how the semantic relationships can be determined for these nets in such a way that state space explosion is avoided as much as possible. Feedback from industrial experts indicates that the language satisfies practical needs, while an experiment with two large process model repositories shows that queries expressed in our language can be evaluated efficiently.

1 Introduction

With the increasing maturity of business process management, more and more organisations need to manage large numbers of business process models, and among these may be models of high complexity. Business process models constitute valuable intellectual property capturing the way an organisation conducts its business. Processes may be defined along the entire value chain and over time a business may gather hundreds and even thousands of business process models. As an example consider China CNR Corporation Limited, a recently formed company, constituting a conglomerate of over 20 subsidiary companies, becoming one of the largest providers of material for high-speed trains. Before the merger of these companies, most of these subsidiaries used their own Enterprise Resource Planning (ERP) systems and together, these systems incorporated more than 200,000 business processes [1]. In this context, support for business process retrieval, e.g. for the purposes of process reuse or process standardization, is a challenging proposition.

While several query languages exist that can be used to retrieve process models from a repository, e.g. BPMN-Q [2] or BP-QL [3, 4], these languages are based on syntactic relationships between tasks and not on semantic relationships between them.

While in a process graph, a task A may follow a task B this does not mean that during execution task B will always follow sometime after task A. Let us consider for example the two process models in Figure 1. These models represent two variants of a business process for opening bank accounts in the BPMN notation [5]. Each variant consists of a number of tasks (represented by rectangles) and dependencies between these tasks. Arcs represent sequential dependencies, while diamonds represent decisions (if there is one incoming arc and multiple outgoing arcs) and simple merges (if there are multiple incoming arcs and one outgoing arc). These two variants could capture the way an account is opened in two different states in which the company operates, and could be part of a collection of various process models for all states in which the bank operates. Now let us assume that an analyst needs to find all states which require to assess the customer's

⋆ Senior Visiting Scholar of Tsinghua University.

credit history when opening an account. In this case, by only using the structural relationships between tasks, we cannot discern between the two variants, i.e. we would retrieve them both, since in both models there is at least a path from task “Receive customer request” to task “Analyse customer credit history”. However, if we consider semantic relationships, we can see that task “Analyse customer credit history” always follows task “Receive customer request” in all instances of the first process variant, but this is not the case for one instance of the second variant (the one where task “Open VIP account” is run). Thus we can correctly exclude the second variant from the results of our query, and return the first variant only.

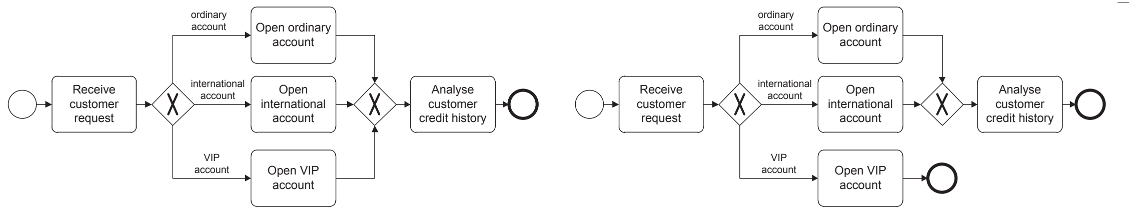


Fig. 1: Two variants of the a business process for opening bank accounts.

A process model retrieval language based on semantic relationships is indeed in line with process execution, e.g. through a workflow management system, and such a language may therefore be more intuitive to use by stakeholders who are not necessarily modelling experts. One challenge though, when it comes to determining semantic relationships between tasks, is how to determine these relationships in a feasible manner, i.e. without suffering from the well-known state space explosion problem.

In the light of the above, in this paper we aim to address the development of an *expressive* business process model query language. We do so by proposing a new query language for process model repositories, namely “A Process-model Query Language” (APQL). This language relies on a number of basic temporal relationships between tasks which can be composed to obtain complex relationships between tasks. These predicates allow us to express queries that can discriminate over single process instances or task instances.

Since the language relies on temporal relationships between tasks, it is independent of a specific business process modelling notation. However, in order to demonstrate its feasibility, we provide a concrete realization of this language in the context of Petri nets. To this end, we adapt the theory of *complete finite prefixes* [6, 7], and its improvements [8, 9], to compute temporal relationships between Petri net transitions.

To demonstrate the usefulness of the language, we conducted a set of structured interviews with practitioners of various Chinese organizations, based on a selected sample of queries. Moreover, we implemented a tool for evaluating APQL queries over repositories of Petri net models, and used this tool to evaluate the performance of the approach over two collections of Petri nets (one from practice, the other, a much larger one, artificially generated). The performance measurements show that indeed the approach can efficiently cope with complex queries issued over very large repositories.

The remainder of this paper is organized as follows. In Section 2.1 we formally define both syntax and semantics of APQL while in Section 3 we show how this language can be operationalized in Petri nets. Next, in Section 4 we present a tool implementation based on this realisation in Petri nets. In Section 5, we comment on the results of the interviews with industry experts and on the performance evaluation of our tool. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 The APQL Business Process Model Query Language

In this section we introduce *APQL* (A Process-model Query Language). First we provide an informal introduction to the language together with its abstract syntax (Section 2.1), then we provide a formal semantics (Section 2.3).

2.1 Syntax

In this section we look at the design rationale for APQL and provide an informal introduction to this language. The syntax of APQL is presented in the form of an abstract syntax, the advantages of which over a concrete syntax having been espoused by Meyer [10]. In essence, in an abstract syntax we can avoid committing ourselves prematurely to specific choices for keywords or to the order of various statements.

APQL is designed as a process model retrieval language that is independent of the actual process modelling language used. This is important as in practice a variety of modelling languages is used (e.g. BPMN, EPCs) and the language should be generally applicable. Another important starting point is the fact that process models have a semantics and it should be possible to exploit this semantics when querying. For example, while task t_1 may follow task t_2 in the process model (i.e. there is a directed path from task t_1 to task t_2), it may be the case that due to the presence of certain splits and joins, in an actual execution task t_2 cannot actually be executed after the execution of task t_1. Hence, a language based on the syntax (i.e. structure) of a process model only is not powerful enough. ... needs to check carefully if we can still make this argument

... do we include the following into a definition?

In order to achieve language independence we define a set of 19 basic predicates $\mathbb{BP}$ for tasks in business process models. In the following, the first three predicates capture the existence, exclusive and concurrent relationships of tasks, and the other 16 predicates capture the causal relationship between tasks. Also, these predicates are defined under the assumption that the tasks involved are part of the business process models being queried[7].

1. $exists\ t_1$: in every process instance, (at least one execution of) t_1 occurs.
2. $exclusive(t_1, t_2)$: in every process instance, it is never possible that both t_1 and t_2 are executed.
3. $concur(t_1, t_2)$: in every process instance, if t_1 is executed then t_2 is executed and vice versa, and t_1 and t_2 are not causally related.

Let Φ be one of the following intermediate predicates,

1. $succ_{any}(t_1, t_2)$: an execution of t_2 is succeeded by an execution of t_1 (e.g. ...t_2...t_1...).
2. $succ_{every}(t_1, t_2)$: *every* execution of t_2 is succeeded by an execution of t_1 (e.g. t_2...t_2...t_1).
3. $pred_{any}(t_1, t_2)$: an execution of t_2 is preceded by an execution of t_1 (e.g. ...t_1...t_2...).
4. $pred_{every}(t_1, t_2)$: *every* execution of t_2 is preceded by an execution of t_1 (e.g. t_1...t_2...t_2).
5. $isucc_{any}(t_1, t_2)$: an execution of t_2 is immediately succeeded by an execution of t_1 (e.g. ...$t_2 t_1$...).
6. $isucc_{every}(t_1, t_2)$: *every* execution of t_2 is immediately succeeded by an execution of t_1 (e.g. $t_2 t_1$...$t_2 t_1$).
7. $ipred_{any}(t_1, t_2)$: an execution of t_2 is immediately preceded by an execution of t_1 (e.g. ...$t_1 t_2$...).
8. $ipred_{every}(t_1, t_2)$: *every* execution of t_2 is immediately preceded by an execution of t_1 (e.g. $t_1 t_2$...$t_1 t_2$).

then

- $\Phi^{\forall}(t_1, t_2)$: $\Phi(t_1, t_2)$ holds *for all* process instances, and
- $\Phi^{\exists}(t_1, t_2)$: there *exists* a process instance where $\Phi(t_1, t_2)$ holds.

In this paper we will demonstrate how these relations can be computed for Petri nets, but when an implementation is provided for another language then APQL can also be used for querying process model collections where the process models are specified according to that language. Note that these predicates are all defined in terms of the execution of a process.

In APQL a query is a set of *Assignments* combined with a *Predicate*.

$$\begin{aligned} Query &\triangleq s : Assignments; p : Predicate \\ Assignments &\triangleq Assignment^* \end{aligned}$$

The result is those process models that satisfy the *Predicate*. An *Assignment* assigns a *TaskSet* to a variable and when evaluating the *Predicate* every variable is replaced by the corresponding *TaskSets*.

$$Assignment \triangleq v : Varname; ts : TaskSet$$

[7] We decide to apply this assumption to all rather than expressing in each of the predicates that an instance of t_1 or t_2 should eventually occur.

TaskSet can be enumerations of tasks or they can be defined in terms of other *TaskSet* either by *Construction* or by *Application*. A *TaskSet* can also be defined through a variable, a *TaskSetVar*.

$$\mathit{TaskSet} \quad \triangleq \quad \mathit{SetofTasks} \mid \mathit{Construction} \mid \mathit{Application} \mid \mathit{TaskSetVar}$$

A *Task* can be defined as a combination of a *TaskLabel* and a *SimDegree*. The idea is that one may be interested in *Tasks* of which the task label bears a certain degree of similarity to a given activity name. There are a number of definitions in the literature concerning label similarity and for a concrete implementation of the language one has to commit to one of these.

$$\begin{aligned}
\mathit{SetofTasks} \quad &\triangleq \quad \mathit{Task}^{+} \\
\mathit{Task} \quad &\triangleq \quad \mathit{TaskLabelExpr} \\
\mathit{TaskLabelExpr} \quad &\triangleq \quad l : \mathit{TaskLabel}; d : \mathit{SimDegree}
\end{aligned}$$

A *TaskSetVar* is simply a variable that may be used in assignments.

$$\mathit{TaskSetVar} \quad \triangleq \quad v : \mathit{Varname}$$

A *TaskSet* can be composed from other *TaskSets* through the application of the well-known set operators such as *union*, *difference*, and *intersection*. Another way to construct a *TaskSet* is by the application of a *TaskCompOp* (i.e. one of the basic predicates introduced earlier, but now interpreted as a function) on another *TaskSet*. In that case we have to specify whether we are interested in the tasks that have that particular relation with *all* or with *any* of the tasks in the *TaskSet*. For example, an expression with *TaskSet* S, *TaskCompOp* *PosSuccAny* and with *AnyAll* indicator *all*, should yield the tasks that succeed all the tasks in S in some process instances.

$$\begin{aligned}
\mathit{Construction} \quad &\triangleq \quad ts_1, ts_2 : \mathit{TaskSet}; o : \mathit{Set\_Op} \\
\mathit{Set\_Op} \quad &\triangleq \quad \mathit{Union} \mid \mathit{Difference} \mid \mathit{Intersection} \\
\mathit{Application} \quad &\triangleq \quad ts : \mathit{TaskSet}; o : \mathit{TaskCompOp}; a : \mathit{AnyAll} \\
\mathit{TaskCompOp} \quad &\triangleq \quad \mathit{Exists} \mid \mathit{Exclusive} \mid \mathit{Concur} \mid \\
&\qquad \mathit{AlwSuccAny} \mid \mathit{AlwSuccEvery} \mid \mathit{AlwPredAny} \mid \mathit{AlwPredEvery} \mid \\
&\qquad \mathit{PosSuccAny} \mid \mathit{PosSuccEvery} \mid \mathit{PosPredAny} \mid \mathit{PosPredEvery} \mid \\
&\qquad \mathit{AlwISuccAny} \mid \mathit{AlwISuccEvery} \mid \mathit{PosISuccAny} \mid \mathit{PosISuccEvery} \mid \\
&\qquad \mathit{AlwIPredAny} \mid \mathit{AlwIPredEvery} \mid \mathit{PosIPredAny} \mid \mathit{PosIPredEvery} \\
\mathit{AnyAll} \quad &\triangleq \quad \mathit{Any} \mid \mathit{All}
\end{aligned}$$

A *Predicate* can consist of a simple *Task*, with the intended semantics that all process models containing that *Task* should be retrieved, a *TaskAlw*, with the intended semantics what the basic predicate exists specifies, a *TaskRel*, with the intended semantics that all process models satisfying that particular relation should be retreived, or it can be recursively defined as a binary or unary *Predicate* through the application of logical operators.

$$\begin{aligned}
\mathit{Predicate} \quad &\triangleq \quad \mathit{Task} \mid \mathit{TaskAlw} \mid \mathit{TaskRel} \mid \mathit{Bin\_Predicate} \mid \mathit{Un\_Predicate} \\
\mathit{Bin\_Predicate} \quad &\triangleq \quad o : \mathit{BinLogOp}; p_1, p_2 : \mathit{Predicate} \\
\mathit{Un\_Predicate} \quad &\triangleq \quad o : \mathit{UnLogOp}; p : \mathit{Predicate} \\
\mathit{BinLogOp} \quad &\triangleq \quad \mathit{And} \mid \mathit{Or} \\
\mathit{UnLogOp} \quad &\triangleq \quad \mathit{Not} \\
\mathit{TaskAlw} \quad &\triangleq \quad t : \mathit{Task}, op : \mathit{AlwExists}
\end{aligned}$$

A *TaskRel* can be 1) a relationship between a *Task* and a *TaskSet* checking whether that *Task* occurs in that *TaskSet* (*TaskInTaskSet*), 2) a relationship between a *Task* and a *TaskSet* and involving a *TaskCompOp* and an *AnyAll* indicator determining whether the *Task* has the *TaskCompOp* relationship with any/all *Tasks* in the *TaskSet* (*Task_TaskSet*), 3) a relationship between two *Tasks* involving a *TaskCompOp* predicate determining whether for the two *Tasks* that predicate holds (*Task_Task*), 4) a relationship between two *TaskSets* involving a *TaskCompOp* and an *AnyAll* indicator determining whether the *Tasks* in those *TaskSets* all have that *TaskCompOp* relationship to each other or whether for each *Task* in the first *TaskSet* there is a corresponding *Task* in the second *TaskSet* for which the relationship holds (*Elt_TaskSet_TaskSet*), or 5) a relationship between two *TaskSets* determined by a set comparison operator (*Set_TaskSet_TaskSet*).

$$\begin{array}{rcl}
\mathit{TaskRel} & \triangleq & \mathit{TaskInTaskSet} \mid \mathit{Task\_TaskSet} \mid \\
 & & \mathit{Task\_Task} \mid \mathit{Elt\_TaskSet\_TaskSet} \mid \\
 & & \mathit{Set\_TaskSet\_TaskSet} \\
\mathit{TaskInTaskSet} & \triangleq & t : \mathit{Task}; ts : \mathit{TaskSet} \\
\mathit{Task\_TaskSet} & \triangleq & t : \mathit{Task}; ts : \mathit{TaskSet}; \\
 & & o : \mathit{TaskCompOp}; a : \mathit{AnyAll} \\
\mathit{Task\_Task} & \triangleq & t_1, t_2 : \mathit{Task}; o : \mathit{TaskCompOp} \\
\mathit{Elt\_TaskSet\_TaskSet} & \triangleq & ts_1, ts_2 : \mathit{TaskSet}; o : \mathit{TaskCompOp}; \\
 & & a : \mathit{AnyAll} \\
\mathit{Set\_TaskSet\_TaskSet} & \triangleq & ts_1, ts_2 : \mathit{TaskSet}; o : \mathit{SetCompOp} \\
\mathit{SetCompOp} & \triangleq & \mathit{Identical} \mid \mathit{Subsetof} \mid \mathit{Overlap}
\end{array}$$

2.2 Sample Queries

In this section we will show some sample queries and how they can be captured in APQL in order to further illustrate the language. The sample queries, specified in natural language, are listed below (and numbered Q_1 to Q_{10}). In these queries, by default the value for the *AnyAll* identifier, when applicable, is *all*, and by default the value for the SimDegree is 1. Figure 2 shows the grammar trees for queries Q_1 to Q_6, while Figure 3 shows the grammar trees for queries $Q_7 to Q_{10}$. Note that in the following A to L are task labels (i.e. activity names).

Q_1. Select all process models where task A occurs in some process instance and task B occurs in every process instance;
Q_2. Select all process models where in every process instance task A occurs before task D;
Q_3. Select all process models where in every process instance every execution of task A occurs before task D;
Q_4. Select all process models where in some process instance task A is followed by task B and task B is followed by task K;
Q_5. Select all process models where in some process instances every execution of task A occurs before an execution of task B;
Q_6. Select all process models where task B occurs in parallel to task C;
Q_7. Select all process models where task B occurs in parallel to task C and where task A occurs in parallel to task H;
Q_8. Select all process models where in every process instance either task B occurs or task C;
Q_9. Select all process models where in every process instance the immediate predecessors of task H are among the immediate successors of tasks B;
Q_{10}. Select all process models where in some process instances such that the immediate predecessors of task H occur after some of the immediate successors of both tasks B and C;

2.3 Semantics

In this section, we use denotational semantics to formally describe the semantics of APQL. For each nonterminal T we introduce a semantic function M_T which defines the meaning of the nonterminal in terms of its parts. The notation that we adopt throughout this section is the notation used in [10].

First, we introduce some auxiliary notation in order to facilitate the subsequent definition of the semantics.

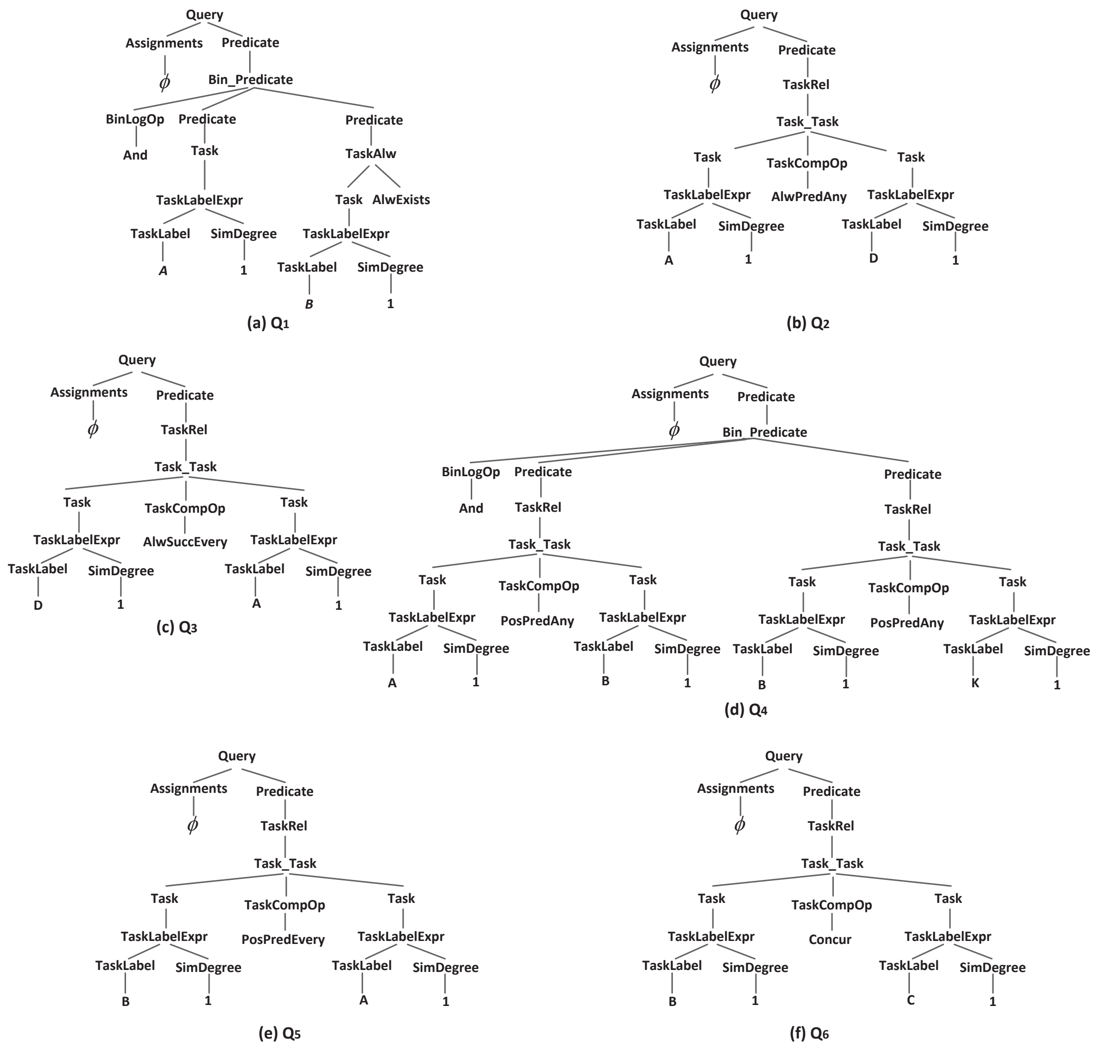
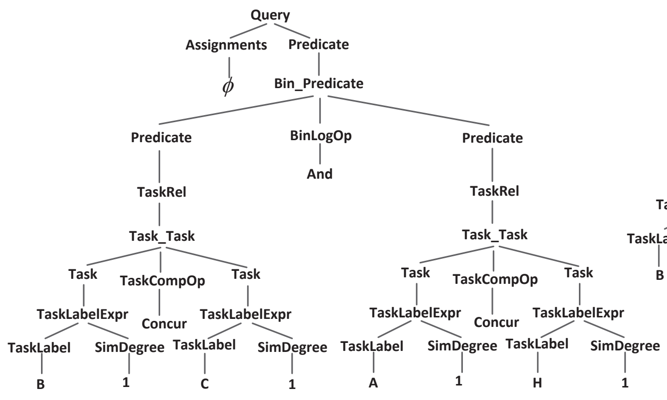
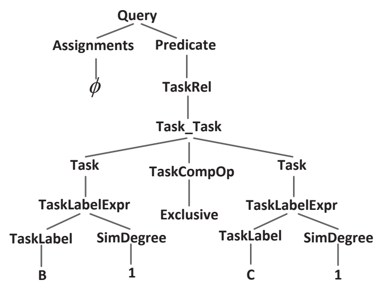


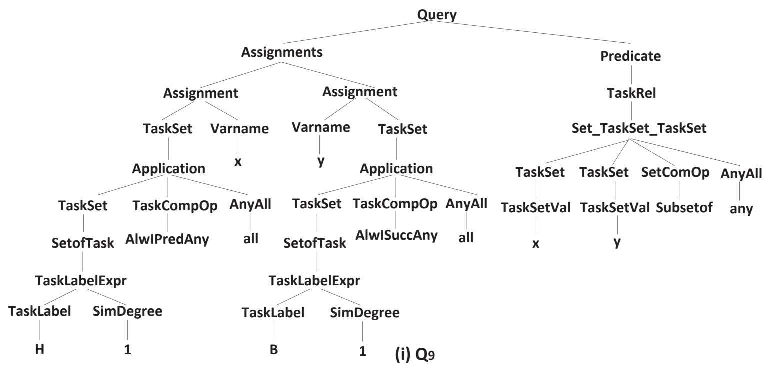
Fig. 2: The APQL grammar trees of sample queries $Q_1 - Q_6$



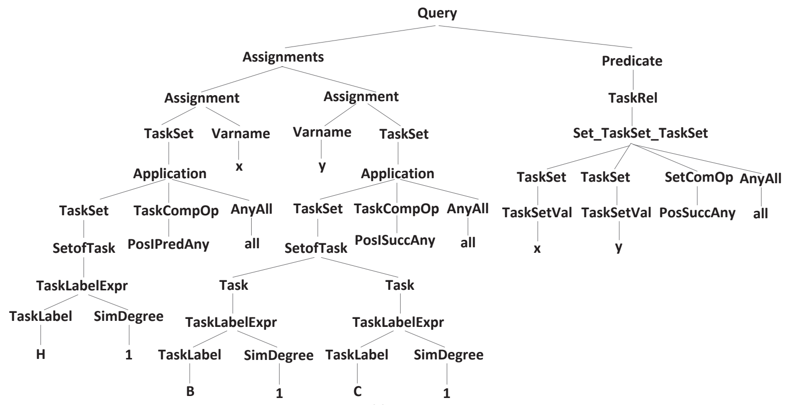
(g) Q7



(h) Q8



(i) Q9



(j) Q10

Fig. 3: The APQL grammar trees of sample queries $Q_7 - Q_{10}$

Definition 1 (overriding union). *The overriding union of $f : X \to Y$ by $g : X \to Y$, denoted as $f \oplus g$, is defined by $g \cup f \backslash \{(x, f(x)) \mid x \in dom(f) \cap dom(g)\}$.*

The following definition introduces a higher order predicate that takes the basic predicates ($\mathbb{BP}$) as input.

Definition 2. *Let N be a net and $t_1, t_2 \in T$ and $\phi \in \mathbb{BP}$*

$$\mathsf{ref}^{\phi}(t_1, t_2, N) = \phi(t_1, t_2, N),$$

i.e. the relation ϕ should hold between t_1 and t_2 in net N.

As queries may use variables, we must know their values during query evaluation. A *Binding* is an assignment of task sets to variables:

$$Binding \triangleq ProcessModel \times Varname \rightarrowtail 2^{Task}$$

Queries are applied to a repository of process models, i.e.

$$Repository \triangleq 2^{ProcessModel}$$

A process model p consists of a collection of tasks T_p. For each task t we can retrieve its label as $L_r(t)$. Label similarity can be determined through the function Sim, where $Sim(l_1, l_2)$ determines the degree of similarity between labels l_1 and l_2 (which yields a value in the range [0,1]).

The query evaluation function M_{Query} takes a query and a repository as input and yields the collection of process models in that repository that satisfy the query:

$$M_{Query} : Query \times Repository \to 2^{ProcessModel}$$

This function is defined as follows:

$$M_{Query}[q : Query, R : Repository] \triangleq M_{Predicate}(q.p, R, M_{Assignments}(q.s, R, \varnothing))$$

The evaluation of the query evaluation function depends on the evaluation of the predicate involved and the assignments involved. When evaluating a sequence of assignments we have to remember the values that have been assigned to the variables involved. Inititally this set of assignments is empty.

$$M_{Assignments} : Assignments \times Repository \times Binding \to Binding$$

The result of a sequence of assignment is a binding where the variables used in the assignments are bound to sets of tasks. If a variable was already assigned a set of tasks in an earlier assignment in the sequence the latest assignment takes precedence over the earlier assignment.

```
$M_{Assignments}[s : Assignments, R : Repository, B : Binding] \triangleq$
  **if** $\neg(s.TAIL).EMPTY$ **then**
    $M_{Assignments}(s.TAIL, R, B \oplus M_{Assignment}(s.FIRST, R, B))$
  **else** $B$
```

The result of an individual assignment is also a binding where the variable is linked to the set of tasks involved.

$$M_{Assignment} : Assignment \times Repository \times Binding \to Binding$$

$$M_{Assignment}[a : Assignment, R : Repository, B : Binding)] \triangleq \{((r, a.v), M_{TaskSet}(a.ts, R, B)(r)) \mid r \in R\}$$

A predicate can be evaluated in the context of a repository and a binding and the result is a set of process models from that repository.

$$M_{Predicate} : Predicate \times Repository \times Binding \to Repository$$

A predicate which is a task yields all process models in the repository that contain a task sufficiently similar to that task (with respect to the task label and similarity degree). A predicate which is a relationship between tasks (a *TaskRel*) yields all the process models that satisfy this relationship. A conjunction yields the union of the process models of the predicates involved, while a disjunction yields the intersection. The negation of a predicate yields the complement of the process models that satisfy the predicate.

```
$M_{Predicate}(p : Predicate, R : Repository, B : Binding) \quad \triangleq$
  case $p$ of
    $Task \Rightarrow \{r \in R \mid \exists t \in T_r[Sim(p.l, L_r(t)) \geq p.d]\}$
    $TaskRel \Rightarrow M_{TaskRel}(p, R, B)$
    $Bin\_Predicate \Rightarrow$
      case $p.o$ of
        $And \Rightarrow M_{Predicate}(p.p_1, R, B) \cap M_{Predicate}(p.p_2, R, B)$
        $Or \Rightarrow M_{Predicate}(p.p_1, R, B) \cup M_{Predicate}(p.p_2, R, B)$
      end
    $Un\_Predicate \Rightarrow R \backslash M_{Predicate}(p, R, B)$
  end
```

A *TaskRel* in the context of a repository and a binding yields a set of process models in that repository.

$$M_{TaskRel} : TaskRel \times Repository \times Binding \rightarrow Repository$$

A *TaskRel* can be used to determine whether a task in a process model occurs in a given task set, whether a given basic predicate holds between a task in a process model and one or all tasks in a given task set, whether a given basic predicate holds between tasks in a process model, whether a given basic predicate holds between two or between all tasks in two given task sets, or whether a given set comparison relation holds between two given task sets.

```
$M_{TaskRel}(tr : TaskRel, R : Repository, B : Binding) \quad \triangleq$
  case $tr$ of
    $TaskInTaskSet \Rightarrow$
      $\{r \in R \mid \exists v \in M_{TaskSet}(tr.ts, R, B)(r)[Sim(tr.t.l, L_r(v)) \geq tr.t.d]\}$
    $Task\_TaskSet \Rightarrow$
      case $tr.a$ of
        $Any \Rightarrow \{r \in R \mid \exists t_1 \in T_r \; \exists t_2 \in M_{TaskSet}(tr.ts, R, B)(r)$
            $[Sim(tr.t.l, L_r(t_1)) \geq t_r.t.d \wedge rel^{tr.o}(t_1, t_2, r)]\}$
        $All \Rightarrow \{r \in R \mid \exists t_1 \in T_r \; \forall t_2 \in M_{TaskSet}(tr.ts, R, B)(r)$
            $[Sim(tr.t.l, L_r(t_1)) \geq t_r.t.d \wedge rel^{tr.o}(t_1, t_2, r)]\}$
      end
    $Task\_Task \Rightarrow \{r \in R \mid \exists v_1, v_2 \in T_r[Sim(tr.t_1.l, L_r(v_1)) \geq tr.t_1.d \wedge$
                $Sim(tr.t_2.l, L_r(v_2)) \geq tr.t_2.d \wedge rel^{tr.o}(v_1, v_2, r)]\}$
    $Elt\_TaskSet\_TaskSet \Rightarrow$
      case $tr.a$ of
        $Any \Rightarrow \{r \in R \mid \exists t_1 \in M_{TaskSet}(tr.ts_1, R, B)(r)$
              $\exists t_2 \in M_{TaskSet}(tr.ts_2, R, B)(r)[rel^{tr.o}(t_1, t_2, r)]\}$
        $All \Rightarrow \{r \in R \mid \forall t_1 \in M_{TaskSet}(tr.ts_1, R, B)(r)$
              $\forall t_2 \in M_{TaskSet}(tr.ts_2, R, B)(r)[rel^{tr.o}(t_1, t_2, r)]\}$
      end
    $Set\_TaskSet\_TaskSet \Rightarrow$
      case $tr.o$ of
        $Identical \Rightarrow$
          $\{r \in R \mid M_{TaskSet}(tr.ts_1, R, B)(r) = M_{TaskSet}(tr.ts_2, R, B)(r)\}$
        $Subsetof \Rightarrow$
          $\{r \in R \mid M_{TaskSet}(tr.ts_1, R, B)(r) \subseteq M_{TaskSet}(tr.ts_2, R, B)(r)\}$
        $Overlap \Rightarrow$
          $\{r \in R \mid M_{TaskSet}(tr.ts_1, R, B)(r) \cap M_{TaskSet}(tr.ts_2, R, B)(r) \neq \varnothing\}$
      end
  end
```

A *TaskSet* within the context of a repository and a binding yields a mapping which assigns to each process model in the repository the collection of tasks within that model that satisfy the restriction imposed by the *TaskSet*.

$$M_{TaskSet} : TaskSet \times Repository \times Binding \rightarrow (ProcessModel \rightarrow 2^{Task})$$

When a *TaskSet* is a set of tasks, then for each process model the result is the set of tasks within that process model that are sufficiently similar to at least one of the tasks in that *TaskSet*. When the *TaskSet* is a variable, then the evaluation is similar except that the task set used is the task set currently bound to that variable. *TaskSets* can also be formed through *Construction* (where the set operators union, difference, and intersection are used) or *Application* (where task sets are formed through set comprehension, i.e. they are defined through properties that they have - these properties relate to the basic predicates).

```
$M_{\mathit{TaskSet}}(tks : \mathit{TaskSet}, R : \mathit{Repository}, B : \mathit{Binding}) \quad \triangleq$
  case $tks$ of
    $\mathit{SetofTasks} \Rightarrow$
      $\{(r, \{t \in T_r \mid \exists_{1 \le i \le tks.Length}[\mathit{Sim}(tks(i).l, L_r(t)) \ge tks(i).d]\}) \mid r \in R\}$
    $\mathit{TaskSetVar} \Rightarrow$
      $\{(r, X) \mid r \in R\}$ where
        $X = \begin{cases} B(r, tks.v) & \textbf{if } (r, tks.v) \in dom(B) \\ \varnothing & \textbf{otherwise} \end{cases}$
    $\mathit{Construction} \Rightarrow$
     case $tks.o$ of
       $\mathit{Union} \Rightarrow$
         $\{(r, M_{\mathit{TaskSet}}(tks.ts_1, R, B)(r) \cup M_{\mathit{TaskSet}}(tks.ts_2, R, B)(r)) | r \in R\}$
       $\mathit{Difference} \Rightarrow$
         $\{(r, M_{\mathit{TaskSet}}(tks.ts_1, R, B)(r) \setminus M_{\mathit{TaskSet}}(tks.ts_2, R, B)(r)) | r \in R\}$
       $\mathit{Intersection} \Rightarrow$
         $\{(r, M_{\mathit{TaskSet}}(tks.ts_1, R, B)(r) \cap M_{\mathit{TaskSet}}(tks.ts_2, R, B)(r)) | r \in R\}$
     end
    $\mathit{Application} \Rightarrow$
     case $tks.a$ of
       $\mathit{Any} \Rightarrow$
         $\{(r, \{t \in T_r \mid \exists v \in M_{\mathit{TaskSet}}(tks.ts, R, B)(r)[rel^{tks.o}(t, v, r)]\}) \mid r \in R\}$
       $\mathit{All} \Rightarrow$
         $\{(r, \{t \in T_r \mid \forall v \in M_{\mathit{TaskSet}}(tks.ts, R, B)(r)[rel^{tks.o}(t, v, r)]\}) \mid r \in R\}$
     end
  end
```

2.4 Semantics of Sample Queries

In order to illustrate the formal semantics of APQL, a number of process models, represented in BPMN, are presented in Figure 4. For each sample query of Section 2.2 and for each model it is indicated whether the model is part of the answer to the query (in that case the box corresponding to the query is ticked otherwise the box is not ticked). Note that in some models tasks with the same label occur (e.g. there are two tasks labeled with activity A in model (f)).

3 Realisation in Petri nets

In this section, we demonstrate how the basic predicates can be derived for Petri nets. In doing so, APQL becomes a concrete query language for repositories of Petri net models.

In order to be able to capture the computation of the basic predicates we introduce some basic Petri net concepts and terminology. In order to make the computation feasible, we also discuss the notion of unfolding. There are many papers discussing these concepts, and we refer the reader to [11, 12, 6, 8, 9, 13, 14] for a more in-depth treatment.

Petri nets are a formal language of which the use for the specification of workflows has been argued by Wil van der Aalst (see e.g. [15, 16]). Petri nets can also be used as a formal foundation for defining the semantics of other process modelling languages or for reasoning about process models specified in these languages, for example BPMN [17], BPEL [18], and EPCs [19].

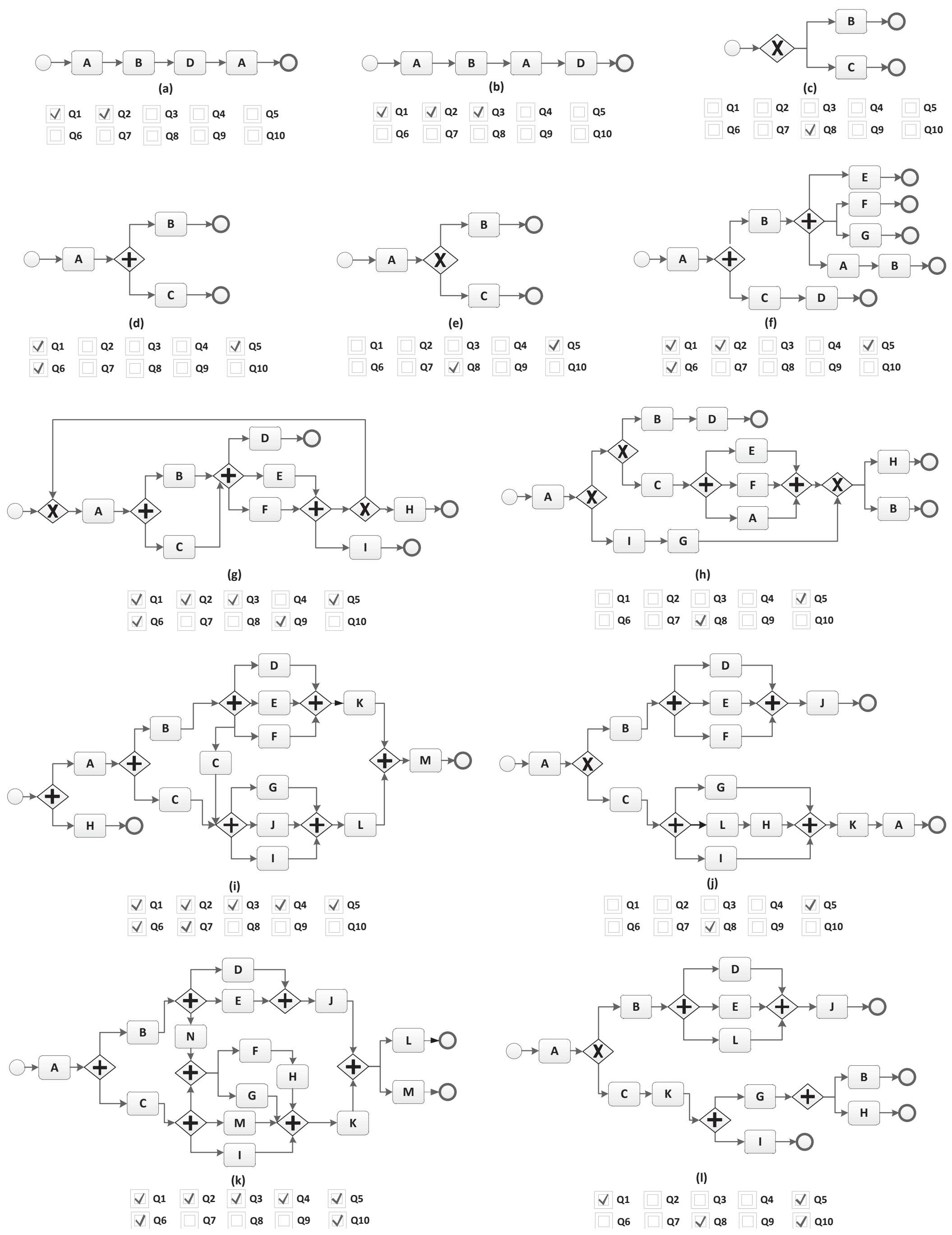


Fig. 4: Some BPMN business process models

3.1 Petri nets

Definition 3 (Petri nets). *A Petri net PN is a tuple (P, T, F), where:*

- *P is a finite set of places;*
- *T is a finite set of transitions, with $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$;*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, such that every transition is the source and the target of an arc.*

The conditions that the sets of places and transitions should be finite and that every transition has at least one input place and at least one output place derive from [9].

For notational convenience we adopt a commonly used notation, where $\bullet n$ represents all the inputs of a node n (which can be a place or a transition) and $n\bullet$ captures all its outputs.

A *marking* of a Petri net is an assignment of tokens to its places. A marking represents a state of the net and a transition, if *enabled*, may change a marking into another marking, thus capturing a state change, by *firing*.

Definition 4 (Marking, Enabling and Firing of a Transition). *Let $PN = (P, T, F)$ be a Petri net.*

- *A marking M of PN is a mapping $M : P \to \mathbb{N}$. A marking may be represented as a collection of pairs, e.g. $\{(p_0, 2), (p_1, 3), (p_2, 0)\}$ or as a vector, e.g. $2p_0 + 3p_1$ (in that case we drop places that do not have any tokens assigned to them). A* Petri net system *is a Petri net with an* initial marking *usually represented as M_0.*
- *Markings can be compared with each other, $M_1 \geq M_2$ iff for all $p \in P$, $M_1(p) \geq M_2(p)$. Similarly, one can define $>$, $<$, $\leq$, $=$.*
- *A transition t is* enabled *in a marking M, denoted as $M \xrightarrow{t}$, iff $M \geq \bullet t$.*
- *A transition t that is enabled in a marking M may* fire *and change marking M into M'. This is denoted as $M \xrightarrow{t} M'$.*

The markings of a Petri net system and the transition relation between these markings constitute a state space.

Definition 5 (Reachability and Safeness). *Let $\Sigma = (P, T, F, M_0)$ be a Petri net system.*

- *A marking M is called* reachable *if a transition sequence $\sigma = t_1 t_2 \ldots t_n$ exists such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_n} M$, which may also be denoted as $M_0 \xrightarrow{\sigma} M$ or, if the choice of σ does not really matter, $M_0 \xrightarrow{*} M$.*
- *Σ is called a* finite *Petri net system if and only if its set of reachable markings is finite.*
- *Σ is called* safe *iff for every reachable marking M and every place $p \in P$: $M(p) \leq 1$.*

In this paper we will only consider safe Petri net systems (noting that such systems are always finite). From a process modelling perspective this is a restriction, but it is not too limiting, as all workflow patterns that are directly supported by Petri nets are also directly supported by safe Petri nets.

3.2 Unfolding

It is well-known that Petri net systems may suffer from the *state space explosion* problem [20]. As such a naive exploration of the state space, especially in the context of a Petri net system which allows highly concurrent behaviour, may not be tractable. In order to deal with this, McMillan [6] proposed the use of the concept of *complete finite prefix*, which is based on the concept of *unfolding*, a well-known partial-order semantics introduced by Nielsen et al. [12] and later elaborated upon by Engelfriet [21] using the term *branching processes*. We will introduce the necessary concepts and notation to make this paper self-contained and to be able to built upon this theory.

Various types of relationship may hold between pairs of nodes in a Petri net.

Definition 6 (Node relations (based on [9])). *Let $PN = (P, T, F)$ be a Petri net.*

- *F^+ is the irreflexive transitive closure of F, while F^* is its reflexive transitive closure. The partial orders defined by these closures are denoted as $<$ and $\leq$ respectively. Hence, for example, $x_1 < x_2$ iff $(x_1, x_2) \in F^+$ and we say that x_1* causally precedes *x_2.*

– *If $x_1 < x_2$ or $x_2 < x_1$ then x_1 and x_2 are* causally related.
– *Nodes x_1 and x_2 are in* conflict, *denoted by $x_1 \# x_2$, iff there exist distinct transitions $t_1, t_2 \in T$ such that $\bullet t_1 \cap \bullet t_2 \neq \varnothing$ and $t_1 \leq x_1$ and $t_2 \leq x_2$. A node x is in self-conflict iff $x \# x$.*
– *Nodes x_1 and x_2 are* concurrent, *denoted as x_1 co x_2, iff x_1 and x_2 are neither causally related nor in conflict.*

The unfolding of a Petri net is an *occurrence net*, usually infinite but with a simple, acyclic structure.

Definition 7 (Occurrence net (based on [9])). *An occurrence net is a net $N' = (B, E, F)$ where:*

– *B is a set of conditions;*
– *E is a set of events, with $B \cap E = \varnothing$;*
– *$F \subseteq B \times E \cup B \times E$ such that 1) for all $b \in B$, $| \bullet b| \leq 1$, 2) F is acyclic, i.e. F^+ is a strict partial order, and 3) for all $x \in B \cup E$ the set of nodes $y \in B \cup E$ for which $y < x$ is finite;*
– *No node is in self-conflict, i.e. for all $x \in B \cup E$, $\neg(x \# x)$.*

We also adopt the notion of $\mathsf{Min}(N')$, as in [9], to denote the set of minimal elements of N' with respect to the strict partial order F^+. As for transitions in Petri nets, we only consider events that have at leaste one input and at least one output conditions. The minimal elements are therefore conditions only, and intuitively $\mathsf{Min}(N')$ can be seen as an initial marking of the net.

Definition 8 (Branching process (based on [21])). *A branching process of a Petri net system $\Sigma = (N, M_0)$, with $N = (P, T, F)$, is a pair (N', h), where*

– *$N' = (B, E, F)$ is an occurrence net;*
– *$h : N' \to N$ is a homomorphism which, following [21], means that:*
  - *$h(B \cup E) \to (P \cup T)$;*
  - *$h \subseteq (B \times P) \cup (E \times T)$, i.e. conditions are mapped to places and events to transitions;*
  - *For every $t \in T$, $h[\bullet t]$ is a bijection between $\bullet t$ and $\bullet h(t)$, and $h[t \bullet]$ is a bijection between $t \bullet$ and $h(t) \bullet$;*
  - *$h[\mathsf{Min}(N')]$ is a bijection between $\mathsf{Min}(N')$ and $\{p \in P \mid M_0(p) > 0\}$.*
– *For all $e, e' \in E$, if $h(e) = h(e')$ and $\bullet e = \bullet e'$, then $e = e'$.*

Note that the definition allows for infinite branching processes. In [21] it is shown that, up to isomorphism, every net system has a unique maximal branching process [9]. For a net system Σ, this unique process is referred to as the *unfolding* of Σ and it is denoted as Unf_Σ.

3.3 Complete Finite Prefix

The unfolding of a labeled Petri net is infinite when the net is cyclic, as for example Unf_{Σ_2} in Figure 5(a). In [7], McMillan proposes an algorithm for the construction of a finite initial part of an unfolding, which contains as much reachability information as the unfolding itself. This part is referred to as a *complete finite prefix (CFP)*, which is often significantly smaller than the unfolding. As illustrated in Figure 5(b) (the dashed arcs should be ignored for the moment), $\mathsf{Fin}^{tp}_{\Sigma_2}$ is the CFP of Σ_2. Also, in Figure 5, events are labeled with integers, and the tuple of conditions positioned next to an event represents the marking of the net upon the occurrence of that event.

The main theoretical notions required to understand the concepts of a CFP are that of *configuration* and *local configuration* of events.

Definition 9 (Configuration [9]). *A configuration C of an occurrence net $N = (B, E, F)$ is a set of events, i.e. $C \subseteq E$, satisfying the following two conditions:*

– *C is causally downward-closed, i.e. $e \in C \wedge e' \leq e \Rightarrow e' \in C$*
– *C is conflict free, i.e. $\forall e, e' \in C : \neg(e \# e')$*

A configuration represents a possible partially ordered run of the net.

Definition 10 (Cut [9]). *Let Σ be a Petri net system and (N', h) be its unfolding. The set of conditions associated with a configuration of N' is called a* cut, *and is defined as $Cut_C = (\mathsf{Min}(N') \cup C\bullet) \setminus \bullet C$. A cut uniquely defines a reachable marking in Σ: $Mark(C) = h(Cut_C)$.*

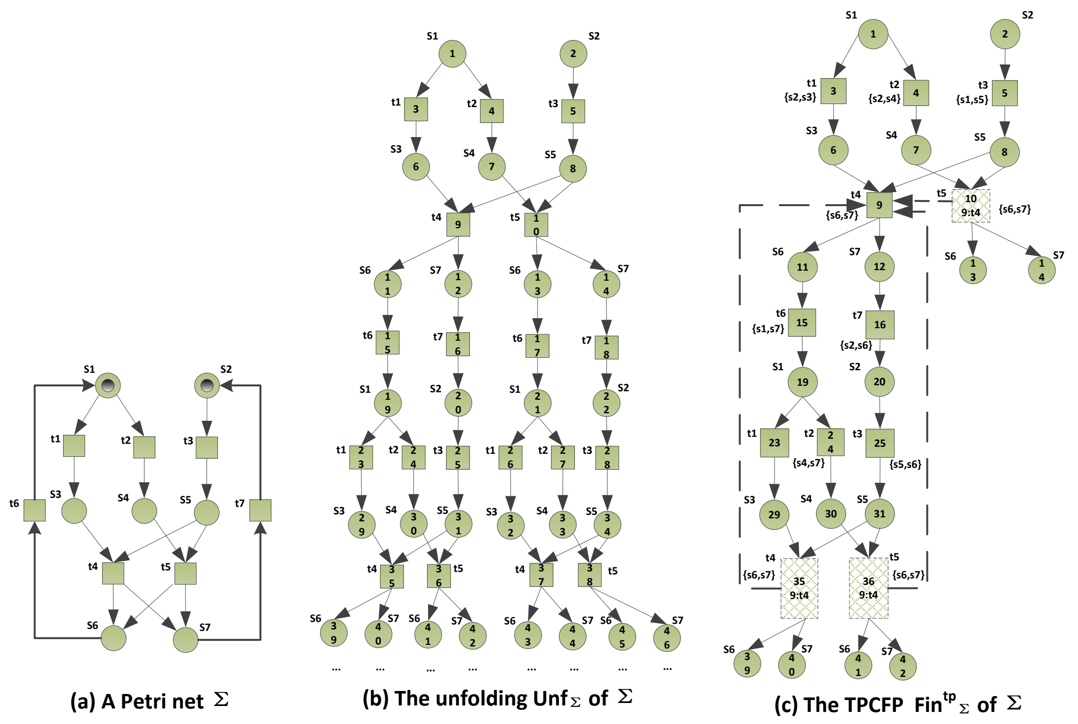

(a) A Petri net Σ
(b) The unfolding Unf_Σ of Σ
(c) The TPCFP Fin^{tp}_Σ of Σ

Fig. 5: Illustration of unfolding and (temporal-order preserving) complete finite prefix using the sample Petri net Σ.

The concepts thus far can be used to introduce the unfolding algorithm. In [9] a branching process (N', h) of a Petri net system Σ is specified as a collection of nodes. These nodes are either conditions or events. A condition is a pair (s, e) where e is the input event of s, while an event is a pair (t, B) where t is a transition and B its input conditions. During the process of unfolding the collection of nodes increases where the function $PE(N', h)$ (which denotes the possible extensions) is applied to determine the nodes to be added. The possible extensions are given in the form of event pairs (t, B), where B is a *co-set* (i.e. a set of nodes that are pairwise in a co relation) of conditions of (N', h) and t is a transition of Σ such that 1) $h(B) = \bullet t$ and 2) no event e exists for which $h(e) = t$ and $\bullet e = B$. In the unfolding algorithm, nodes from the set of possible extensions $PE(N', h)$ are added to the unfolding of the net till this set is empty (i.e. there are no more extensions).

In the complete finite prefix approach, first described in [8] (though we will follow the presentation of [9]), it is observed that a finite prefix of an unfolding may contain all reachability-related information. The key to obtaining a complete finite prefix is to identify those events at which we can cease unfolding (e.g. events 10, 21, and 22 in $\text{Fin}^{tp}_{\Sigma_2}$) without loss of reachability information. Such events are referred to as *cut-off events* and they are defined in terms of an *adequate order* on configurations.

Definition 11 (Adequate order [9]). *Let $\Sigma = (P, T, F, M_0)$ be a Petri net system and let $\prec$ be a partial order on the finite configurations of one of its branching processses, then $\prec$ is an adequate order iff:*

- *$\prec$ is well founded;*
- *For all configurations C_1 and C_2, $C_1 \subset C_2 \Rightarrow C_1 \prec C_2$;*
- *The $\prec$ order is preserved in the context of finite extensions, i.e. if $C_1 \prec C_2$ and $Mark(C_1) = Mark(C_2)$, then if we extend C_1 with E to C'_1 and we extend C_2 to C'_2 by using an extension isomorphic to E then $C'_1 \prec C'_2$.*

The last clause of this definition is not fully formalised here as it requires a certain amount of formalism and we hope that the idea is sufficiently clear from an intuitive point of view. We refer the reader to [9] for a complete formal definition of this notion. Note that, as pointed out in [9], the order $\prec$ is essentially a parameter to the approach.

The concept of *local configuration* captures the idea of all preceding events to an event such that these events form a configuration.

Definition 12 (Local Configuration [9]). *Let $N = (B, E, F)$ be an occurrence net, the local configuration of an event $e \in E$, denoted $[e]$, is the set of events e', where $e' \in E$, such that $e' \leq e$.*

Definition 13 (Cut-off event [9]). *Let Σ be a Petri net system, N' be one of its branching processes and let $\prec$ be an adequate order on the configurations of N', then an event e is a cut-off event iff N' contains a local configuration $[e']$ for which $Mark([e]) = Mark([e'])$ and $[e'] \prec [e]$.*

Without loss of reachability information, we can cease unfolding from an event e, if e takes the net to a marking which can be caused by some earlier other event e'. So in Figure 5(b), we remove the part after event 10 from Unf_{Σ_2} because it is isomorphic to that after event 9, i.e. the continuation after event 10 is essentially the same as the continuation after event 9. For the correctness of this approach we refer to [9].

3.4 Temporal-order Preserving Complete Finite Prefix

So far is our summary of existing theory. We would like to make an observation here and that is that CFPs lose important (for our purposes at least) temporal ordering information.

Consider again Unf_{Σ_2} in Figure 5(a), we can observe that t_3 labeled by event 3 precedes t_7 labeled by event 14, but in the CFP of Σ_2 (without considering the two types of dashed links which will be explained later) in Figure 5(b), there does not exist a path from t_3 labeled by event 3 to t_7 labeled by event 12. This means that not every scenario of t_3 preceding t_7 is preserved in the CFP of Σ_2. In order to deal with this we propose an adaptation of the complete finite prefix algorithm in [9]. We refer to this adaptation as *temporal-order preserving complete finite prefix* or *TPCFP* for short. This type of prefix keeps the simplicity of CFP and adds two relations, which help preserve the temporal ordering of events in the unfolding. First we introduce some notation to capture the adapted algorithm formally.

Definition 14 (Unfolding notations). *Let $\Sigma = (N, M_0)$ be a Petri net system, with $N = (P, T, F)$, and let $\rho = (N', h)$, with $N' = (B', E', F')$, be an unfolding of Σ, then we define:*

- $\mathsf{Eq}(M, \rho) = \{e \in E' \mid Mark([e]) = M\}$ *for any reachable marking M of N. If ρ is clear from the context, we will simply omit it and write $\mathsf{Eq}(M)$ (a similar convention holds for the remainder of this definition and ρ is not introduced explicitly anymore).*
- $\mathsf{continuation}(M)$ *which refers to the continuation node in ρ for a reachable marking M. It is defined as the unique event $e' \in \mathsf{Eq}(M)$ such that for all $e \in \mathsf{Eq}(M)$, if $e \neq e'$ then $[e'] \prec [e]$.*
- $\mathsf{Off}(M) = \mathsf{Eq}(M) \backslash \{\mathsf{continuation}(M)\}$ *which denotes the set of cut-off events for a reachable marking M.*

A temporal-order preserving CFP is then achieved by the introduction of *links* from a cut-off event to a continuation event.

Definition 15 (Temporal-order Preserving Complete Finite Prefix). *A TPCFP for a Petri net system $\Sigma = (N, M_0)$, denoted by $\mathcal{T}_\Sigma$, is a tuple (ρ, L), where:*

- *$\rho = (B, E, G)$ is the* CFP *of Σ;*
- *L is a set of links defined as $L \subseteq E \times E$, and if $(e, e') \in L$, then there is a reachable marking M such that $e' = \mathsf{continuation}(M)$ and $e \in \mathsf{Off}(M)$.*

Example 1 *Consider $\mathcal{T}_{\Sigma_2}$ as shown in Figure 5 (b), $L = \{(22, 7), (21, 8), (10, 9)\}$.*

A *TPCFP* preserves all temporal-order related information (in essence, precedence relations) and does so with little modification to the original *CFP* (among others, it leaves its topology intact). In order to generate a *TPCFP* from a safe Petri net system, we make a small adaptation to the algorithm for generating a *CFP* as proposed in [9]. This adapted algorithm is presented as Algorithm 1. The data structure for the representation of a *TPFCP* comprises that of a CFP in [9] (written $\mathsf{Fin}^{tp}.N$) and a set of links (written $\mathsf{Fin}^{tp}.L$). $PE(\mathsf{Fin}^{tp}.N)$ is the set of events that can be added to a branching process $\mathsf{Fin}^{tp}.N$ (i.e. possible extensions of $\mathsf{Fin}^{tp}.N$), as defined in [9]. Application of $minimal(pe, \prec)$ yields an event e which satisfies the following condition taken from [9]: $e \in pe$ and $[e]$ is minimal with respect to $\prec$. The predicate $expansion\_required(e, cut\_off)$ is an abbreviation of $[e] \cap cut\_off = \varnothing$, the condition used in [9]. Next, $cut\_off\_event(e, \mathsf{Fin}^{tp}.N, c)$ returns the result of whether or not e is a cut-off event of $\mathsf{Fin}^{tp}.N$ (as in [9]), and during its application, the corresonding continuation event for e is returned in the local variable c so that it does not need to be determined again when adding links. Note that we use $X \cup:= Y$ as an abbreviation for $X := X \cup Y$. As we only add

Algorithm 1: Computation of TPCFP via an adaption of the CFP algorithm in [9]

```
**Input**: A safe Petri net system $\Sigma = (P, T, F, M_0)$
**Output**: $\mathsf{Fin}^{tp}$ $(N : Net, L : Links)$ the TPCFP of $\Sigma$
**begin**
    $\mathsf{Fin}^{tp}.N$ := $\{(s, \varnothing) \mid M_0(s) > 0 \wedge s \in P\}$;
    $\mathsf{Fin}^{tp}.L$ := $\varnothing$;
    $pe$ := $PE(\mathsf{Fin}^{tp}.N)$;
    $cut\_off$ := $\varnothing$;
    **while** $pe \neq \varnothing$ **do**
        $e := minimal(pe, \prec)$;
        **if** $expansion\_required(e, cut\_off)$ **then**
            $\mathsf{Fin}^{tp}.N \cup:= \{(e.t, \bullet e)\} \cup \{(s, e) \mid s \in e.t\bullet\}$;
            $pe := PE(\mathsf{Fin}^{tp}.N)$;
            **if** $cut\_off\_event(e, \mathsf{Fin}^{tp}.N, c)$ **then**
                $cut\_off \cup:= \{e\}$;
                $\mathsf{Fin}^{tp}.L \cup:= \{(e, c)\}$;
        **else** $pe := pe \backslash \{e\}$
```

link information in the TPCFP algorithm compared to the CFP algorithm [9], the complexity of the adapted algorithm is comparable to that of the original CFP algorithm. It should be noted that for our purposes the TPCFP algorithm is applied to compute the basic predicates for a repository of process models specified as Petri nets. Hence, the runtime cost of a query is not determined by the complexity of the TPCFP algorithm, as the computation of the basic temporal relations would already have taken place (so in some ways we trade space for time).

3.5 Determining the Basic Predicates

First, we convert a TPCFP to a directed bipartite graph (or bigraph), where links are transformed to connections between corresponding condition nodes via newly added silent events. The following Algorithm 2 to 6 are specified for pre-processing a process TPCFP to a set of directed bigraphs that are free of choices and capture all possible executions of the given process.

Algorithm 2: Algorithm for decomposition of a TPCFP into a set of conflict-free TPCFPs

```
function GetAllExecutions
Input: A TPCFP $U = (\rho, L)$ where $\rho = (B, E, F)$ and $L \subseteq E \times E$
Output: A set of TPCFPs $\mathbb{U}$
begin
    CS := GetAllCoSets($\rho$);
    /* compute CFPs from each of the co-sets */
    $\mathit{\Gamma}$ := ComputeCFPs($\rho$,cs);
    /* generate TPCFPs from the above (conflict-free) CFPs */
    $\mathit{\Gamma}_{tmp}$ := $\mathit{\Gamma}$;
    while $\mathit{\Gamma}_{tmp} \neq \varnothing$ do
        Select $\rho_i \in \mathit{\Gamma}_{tmp}$;
        $E_{cutoff}$ := GetCutoffEvents($\rho_i$);
        $f_{update}$ := FALSE;
        while $E_{cutoff} \neq \varnothing \wedge \neg f_{update}$ do
            Select $e_{cut} \in E_{cutoff}$;
            $L_{tmp}$ := GetLinks_from($L, e_{cut}$);
            /* add link info to each CFP */
            while $L_{tmp} \neq \varnothing \wedge \neg f_{update}$ do
                Select $l_i \in L_{tmp}$;
                $e_{cont}$ := GetContinuationEvent($l_i$);
                if $e_{cont} \in \rho_i.E$ then
                    $L_i \cup := \{l_i\}$;
                    $L_{tmp} \setminus := \{l_i\}$;
                else
                    /* update CFPs */
                    $\mathit{\Gamma}_{add}$ := GetUpdatedCFPs($\rho_i, \mathit{\Gamma}, e_{cut}, e_{cont}$);
                    $\mathit{\Gamma} \setminus := \{\rho_i\}$;
                    $\mathit{\Gamma} \cup := \mathit{\Gamma}_{add}$;
                    $\mathit{\Gamma}_{tmp} \cup := \mathit{\Gamma}_{add}$;
                    /* update links */
                    $L_{add}$ := GetUpdatedLinks($L_{tmp}, e_{cut}, e_{cont}$);
                    $L \setminus := \{l_i\}$;
                    $L \cup := L_{add}$;
                    $f_{updated}$ := TRUE;
            $E_{cutoff} \setminus := \{e_{cut}\}$;
        if $\neg f_{update}$ then
            $\mathbb{U} \cup := \{(\rho_i, L_i)\}$;
            $\mathit{\Gamma}_{tmp} \setminus := \{\rho_i\}$
```

Algorithm 3: Algorithm for update of a CFP with a link leading outside the CFP

```
function GetUpdatedCFPs
Input: A CFP $\rho = (B, E, F)$, a set of CFPs $\Gamma$, a (cut-off) event $e_{cut}$, a (continuation) event $e_{cont}$
Output: A set of (updated) CFPs $\Gamma'$
begin
    /* get $\rho$ ready by removing the successor conditions of $e_{cut}$ (in $\rho$) */
    $B_{tmp}$ := iSuccessors($\rho, e_{cut}$);
    $\rho.B \setminus := B_{tmp}$;
    $\rho.F \setminus := \{e_{cut}\} \times B_{tmp}$;
    /* retrieve and process the CFPs that contain $e_{cont}$ in $\Gamma$ */
    for each $\rho_i \in \Gamma$ where $e_{cont} \in \rho_i.E$ do
        /* remove $e_{cont}$ and all its predecessors in $\rho_i$ */
        $H$ := AllPredecessors($\rho_i, e_{cont}$) $\cup \{e_{cont}\}$;
        $\rho_i.B \setminus := H$;
        $\rho_i.E \setminus := H$;
        $\rho_i.F \cap := (\rho'_i.B \times \rho'_i.H \cup \rho'_i.H \times \rho'_i.B)$;
        /* connect the above (updated) $\rho$ and $\rho_i$ to $\rho'$ */
        $\rho'.B := \rho.B \cup \rho_i.B$;
        $\rho'.E := \rho.E \cup \rho_i.E$;
        $\rho'.F := \rho.F \cup \rho_i.F \cup \{e_{cut}\} \times$ InitialConditions($\rho_i$);
        $\Gamma' \cup := \{\rho'\}$
```

Algorithm 4: Algorithm for update of links resulting from updated CFPs

```
function GetUpdatedLinks
Input: A set of links $L$, a (cut-off) event $e_{cut}$, a (continuation) event $e_{cont}$
Output: A set of (updated) links $L'$
begin
    /* retrieve the links leading to $e_{cont}$ except those from $e_{cut}$ */
    $L_{tmp}$ := GetLinks_to($L, e_{cont}$) $\setminus \{(e_{cut}, e_{cont})\}$;
    /* update the links by replacing $e_{cont}$ with $e_{cut}$ */
    for each $(e, e_{cont}) \in L_{tmp}$ do
        $L' \cup := \{(e, e_{cut})\}$
```

Algorithm 5: Algorithm for converting a TPCFP to a directed bigraph

```
function TPCFP2Bigraph
Input: A TPCFP $U = (\rho, L)$ where $\rho = (B, E, F)$ and $L \subseteq E \times E$
Output: A directed bigraph $G$ = ($V_{cond}$: condition nodes, $V_{event}$: event nodes, $A$: directed edges)
begin
    $V_{cond} := B$;
    $V_{event} := E$;
    $A := F$;
    for each $(e_1, e_2) \in L$ do
        for each $b \in$ iSuccessors($\rho, e_1$) do
            select $b' \in$ iSuccessors($\rho, e_2$) where PlaceId($b$) $=$ PlaceId($b'$);
            $V_{event} \cup := \{\tau_{(b,b')}\}$;
            $A \cup := \{(b, \tau_{(b,b')}), (\tau_{(b,b')}, b')\}$
```

Algorithm 6: Algorithm for preprocessing a TPCFP to a set of directed bigraphs

```
function PreProcess($U$: TPCFP): $\mathbb{G}$ set of bigraphs
begin
    $\mathbb{U}$ := GetAllExecutions($U$);
    for each $U \in \mathbb{U}$ do
        $\mathbb{G} \cup:=$ TPCFP2Bigraph($U$)
```

Algorithm 7: Algorithm for determining the predicate Exists

```
function EXISTS($U$: TPCFP, $t$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess($U$);
    return $\bigwedge_{G\in\mathbb{G}} (\bigvee_{e\in\text{RetrieveAllEvents}(G,t)} e \in G.V_{event})$
```

It is worth mentioning the way to implement the operators $\bigwedge$ and $\bigvee$ in the above algorithm. The way to implement $\bigvee$ is to check for each event $e \in \text{RetrieveAllEvents}(G, t)$, calculate the result of the boolean expression $e \in G.V_{event})$, and return TRUE once the result is true, otherwise return FALSE. The way to implement $\bigwedge$ is to check for each bigraph $G \in \mathbb{G}$, calculate the result of the corresponding boolean expression, and return FALSE once the result is false, otherwise return TRUE.

Algorithm 8: Algorithm for determining the predicate Exclusive

```
function EXCLUSIVE($U$: TPCFP, $t$: taskID, $t'$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess($U$);
    return $\bigwedge_{G\in\mathbb{G}} (\bigwedge_{e\in\text{RetrieveAllEvents}(G,t),e'\in\text{RetrieveAllEvents}(G,t')} \neg(e \in G.V_{event} \wedge e' \in G.V_{event}))$
```

Algorithm 9: Algorithm for determining the predicate Concur

```
function CONCUR($U$: TPCFP, $t$: taskID, $t'$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess($U$);
    return $\bigwedge_{G\in\mathbb{G}} (\bigwedge_{e\in\text{RetrieveAllEvents}(G,t),e'\in\text{RetrieveAllEvents}(G,t')} \text{Co}(G, e, e')^*)$
```

$^*\text{Co}(G, e, e') = (e \in G.V_{event} \wedge e' \in G.V_{event} \wedge e \notin \text{AllPredecessors}(G, e') \wedge e' \notin \text{AllPredecessors}(G, e))$
$\vee(e \notin G.V_{event} \wedge e' \notin G.V_{event})$

Algorithm 10: Algorithm for determining the successor relationship in a conflict-free TPCFP

```
function Succeeds($G$: bigraph, $n$: node, $t$: taskID, $V$: visited nodes): boolean
begin
    $W :=$ RetrieveAllEvents($G, t$);
    if $n \in W$ then
        return TRUE;
    else
        if ($n \in V \vee$ iSuccessors$(G, n) = \varnothing$) then
            return FALSE;
        else
            return $\bigvee_{s\in\text{iSuccessors}(G,n)}$ Succeeds($G, s, t, V \cup \{n\}$)
```

Similarly, we can define the algorithm for determining the *predecessor* relationship in a conflict-free TPCFP in terms of a boolean function *Proceeds*(G: bigraph, n: node, t: taskID, V: visited nodes). The function can be specified by replacing $\text{iSuccessors}(G, n)$ with $\text{iPredecessors}(G, n)$ and replacing recursively called function *Succeeds*$(G, s, t, V \cup \{n\})$ with *Proceeds*$(G, s, t, V \cup \{n\})$ in Algorithm 10.

Algorithm 11: Algorithm for determining the intermediate predicate SuccEvery

```
function SUCCEVERY($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{former})$;
    return $\bigwedge_{e \in W}$ Succeeds$(G, e, t_{latter}, \varnothing)$
```

Algorithm 12: Algorithm for determining the intermediate predicate SuccAny

```
function SUCCANY($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{former})$;
    return $\bigvee_{e \in W}$ Succeeds$(G, e, t_{latter}, \varnothing)$
```

Algorithm 13: Algorithm for determining the intermediate predicate PredEvery

```
function PREDEVERY($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{latter})$;
    return $\bigwedge_{e \in W}$ Proceeds$(G, e, t_{former}, \varnothing)$
```

Algorithm 14: Algorithm for determining the intermediate predicate PredAny

```
function PREDANY($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{latter})$;
    return $\bigvee_{e \in W}$ Proceeds$(G, e, t_{former}, \varnothing)$
```

Algorithm 15: Algorithm for determining the predicate AlwSuccEvery

```
function ALWSUCCEVERY($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigwedge_{G \in \mathbb{G}}$ SuccEvery$(G, t_{former}, t_{latter})$
```

Algorithm 16: Algorithm for determining the predicate AlwSuccAny

```
function ALWSUCCANY($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigwedge_{G \in \mathbb{G}}$ SuccAny$(G, t_{former}, t_{latter})$
```

Algorithm 17: Algorithm for determining the predicate PosSuccEvery

```
function POSSUCCEVERY($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigvee_{G \in \mathbb{G}}$ SuccEvery$(G, t_{former}, t_{latter})$
```

Algorithm 18: Algorithm for determining the predicate PosSuccAny

```
function PosSuccAny($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess($U$);
    return $\bigvee_{G \in \mathbb{G}}$ SuccAny$(G, t_{former}, t_{latter})$
```

The four predicates AlwPredEvery, AlwPredAny, PosPredEvery, and PosPredAny can be defined in a similar way as the above.

Algorithm 19: Algorithm for determining immediate event successors in a conflict-free TPCFP

```
function ISucceeds($G$: bigraph, $e$: event node, $t$: taskID): boolean
begin
    /* gather all the immediate non-silent event successors of $e$ in $G$ */
    $X$ := iSuccessors$(G, e)$;
    while $X \neq \varnothing$ do
        get $e'$ where $(c, e') \in G.A$;
        if $e' \notin G.V_{event}.\tau$ then
            $Y \cup := \{e'\}$;
        else
            $X \cup :=$ iSuccessors$(G, e')$;
        $X \setminus := \{e'\}$
    $W$ :=RetrieveAllEvents$(G, t)$;
    return $Y \cap W \neq \varnothing$
```

Similarly, we can define the algorithm for determining immediate event *predecessors* in a conflict-free TPCFP in terms of a boolean function *IProceeds*(G: bigraph, e: event node, t: taskID). The function can be specified by replacing function iSuccessors with iPredecessors in Algorithm 19.

Algorithm 20: Algorithm for determining the intermediate predicate ISuccEvery

```
function ISuccEvery($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{former})$;
    return $\bigwedge_{e \in W}$ ISucceeds$(G, e, t_{latter})$
```

Algorithm 21: Algorithm for determining the intermediate predicate ISuccAny

```
function ISuccAny($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{former})$;
    return $\bigvee_{e \in W}$ ISucceeds$(G, e, t_{latter})$
```

Algorithm 22: Algorithm for determining the intermediate predicate IPredEvery

```
function IPredEvery($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W :=$ RetrieveAllEvents$(G, t_{latter})$;
    return $\bigwedge_{e \in W}$ IProceeds$(G, e, t_{former})$
```

Algorithm 23: Algorithm for determining the intermediate predicate IPredAny

```
function IPredAny($G$: bigraph, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $W$ := RetrieveAllEvents$(G, t_{latter})$;
    return $\bigvee_{e \in W}$ IProceeds$(G, e, t_{former})$
```

Algorithm 24: Algorithm for determining the predicate AlwISuccEvery

```
function AlwISuccEvery($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigwedge_{G \in \mathbb{G}}$ ISuccEvery$(G, t_{former}, t_{latter})$
```

Algorithm 25: Algorithm for determining the predicate AlwISuccAny

```
function AlwISuccAny($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigwedge_{G \in \mathbb{G}}$ ISuccAny$(G, t_{former}, t_{latter})$
```

Algorithm 26: Algorithm for determining the predicate PosISuccEvery

```
function PosISuccEvery($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigvee_{G \in \mathbb{G}}$ ISuccEvery$(G, t_{former}, t_{latter})$
```

Algorithm 27: Algorithm for determining the predicate PosISuccAny

```
function PosISuccAny($U$: TPCFP, $t_{former}$: taskID, $t_{latter}$: taskID): boolean
begin
    $\mathbb{G}$ := PreProcess$(U)$;
    return $\bigvee_{G \in \mathbb{G}}$ ISuccAny$(G, t_{former}, t_{latter})$
```

The four predicates AlwIPredEvery, AlwIPredAny, PosIPredEvery, and PosIPredAny can be defined in a similar way as the above.

As is clear from the abstract grammar of APQL, when querying a process model repository we need to know the presence of tasks in a process model (which can be supported during querying through the use of indexes, see e.g. [22]) and the basic behavioural relationships: (immediate) precedence/succession, concurrence and exclusivity. Upon deriving these relationships, we store them into a so-called *behaviour relation matrix*.

During the construction of the TPCFP we can keep track of the immediate *succession* relationship between events. After the completion of the computation of the TPCFP we consider all links $(t, u) \in L$ and record every immediate successor of u also as an immediate successor of t. Having recorded all immediate successor relationships, we can now compute the transitive closure (using an algorithm of our choice) and record all successor relations. Having done so, we know all PosPredAny, PosSuccAny, PosIPredAny, PosISuccAny relations between events.

Two events are in a *exclusive* relationship iff their nearest common ancestor is a condition rather than an event (noting that we are dealing with safe nets only). Therefore we have to investigate all pairs that are not in a causal relationship. A good starting point is to consider all conditions that indicate exclusive relationships, i.e. all conditions that have multiple output events. The events in separate branches originating from this condition are mutually exclusive.

Finally, we set all remaining event pairs to *concurrent*.

4 System Implementation

We present in this section the implementation of a query tool, namely APQL Querier, that supports querying business process models with APQL. It is developed as part of the BeehiveZ system v2.0, which is a generic business process models management system, and is implemented using Java. BeehiveZ is available under an open-source license at `http://code.google.com/p/beehivez/downloads/list`, and the code of the APQL Querier as part of the BeehiveZ 2.0 can be downloaded from `http://code.google.com/p/beehivez/downloads/list`.

Figure 6 depicts the structure of the APQL Querier and the Process Models Repository with which the APQL Querier interacts within the framework of BeehiveZ. In BeehiveZ, the Process Models Repository is implemented as MySQL databases. It stores process models that are specified in Petri nets, which does not lose generality since business processes defined by various notations (e.g. BPMN, BPEL, EPC, UML, YAWL) can be transformed into Petri nets [23]. The TPCFP Generator computes TPCFPs for each of the Petri net process models according to Algorithm 1. Then, the behaviour relation matrices between events in TPCFPs can be calculated. Both TPCFPs and the behaviour relation matrices are stored in the repository databases.

The core of the APQL Querier is the Query Engine: in the front-end, it takes as input the queries from users via the Query Editor and produces as output the querying results to users via the Query Results Display; while in the back-end, it interacts with the Process Models Repository and performs searching for process models that satisfy a given query. Below, we provide more detailed descriptions of each of the components in the APQL Querier.

– Query Editor is for users to edit queries using a concrete syntax of APQL. This is defined, based on the abstract syntax of APQL in Section 2.1, using a BNF grammar which can be download from `http://code.google.com/p/beehivez/downloads/list`.
– Query Results Display shows to users the querying results in the form of a list of process models. Users can choose to view a specific model in the Process Model Viewer[8], which is a generic component in BeehiveZ for graphically displaying process models in Petri nets.
– Query Engine comprises mainly two components:
    - Parser converts query statements into grammar trees. It is built using JavaCC5.0[9].

[8] This is developed using existing classes in ProM 5.2 (http://promtools.org/prom5/).
[9] https://javacc.dev.java.net/

- Evaluator searches for the process models that satisfy requirements of a given query, using the grammar tree of the query, the repository of process models, and the behaviour relation matrices computed from the corresponding TPCFPs of these process models. In this component, each construct in APQL has a corresponding function which implements the specific semantics of the construct. The search procedure is carried out by examining from the root to the leaves of the grammar tree to see if a process model matches the grammar tree.

Figure 7 shows a screen shot of querying process models through BeehiveZ.

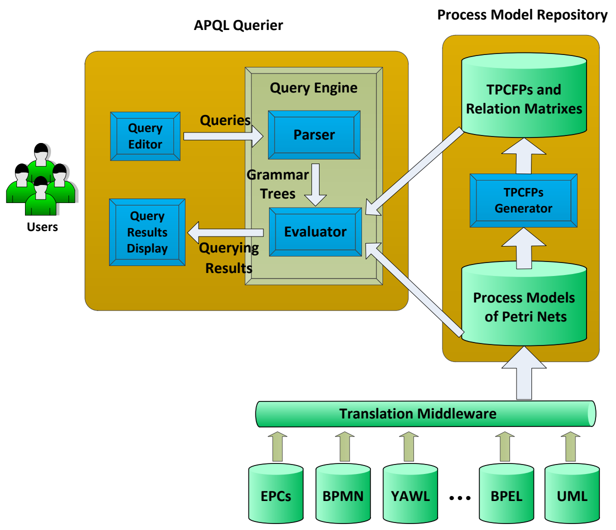


Fig. 6: Structure of the APQL Querier and Process Models Repository within BeehiveZ.

5 Evaluation

In this section we report on the results of two experiments we conducted to evaluate the expressiveness of APQL, and the performance of its implementation in BeehiveZ.

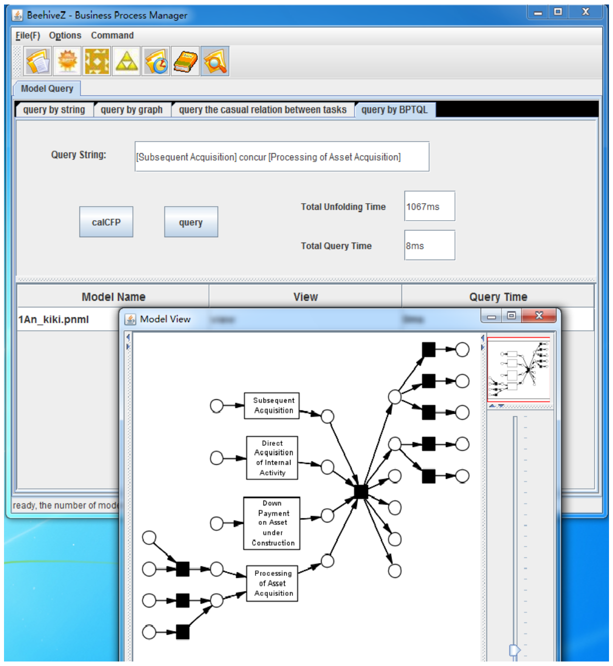
5.1 Performance Measurements

We prepared a set of ten sample queries and we measured the evaluation of each of these queries over two process model collections. The first is a real-life repository, namely the SAP R/3 reference model []. The second is a set of over 100,000 artificially generated [10]. We conducted our tests on an Intel Core i5 CPU 760 @2.80GHz and 4GB RAM, running Windows 7 Ultimate and JDK6. The heap memory for the JVM was set to 1GB.

For these two experiments, the pre-processing includes 3 aspects as follows:

1. Build inverted index for every label appeared in TPCFPs. For each label, we record all processes which have nodes labeled by it. The structure is {label, list of (name of node, name of TPCFP)}. All inverted indexes are managed by Lucene. According to this index, after queries are arisen, system can filter instantly a set of preliminary candidate models in which all labels appeared in queries are involved. The rest of the models can be ignored for not relevant to queries. This step can reduce the scope of searching.
2. The data structure of TPCFP is based on the underlying incidence matrixes of the original nets. Conditions, events and directed flows are represented by nodes of doubly linked lists which support in particular fast insertion of nodes and backward traverse.

[10] The second dataset is available from `http://sourceforge.net/projects/beehivez/models`. The first cannot be disclosed for copyright reasons


BeehiveZ - Business Process Manager
File(F) Options Command
Model Query
query by string
query by graph
query the casual relation between tasks
query by BPTQL
calCFP
query
Model View
Subsequent Acquisition
Direct Acquisition of Internal Activity
Down Payment on Asset under Construction
Processing of Asset Acquisition

Query String: [Subsequent Acquisition] concur [Processing of Asset Acquisition]

Total Unfolding Time 1067ms

Total Query Time 8ms

Model Name	View	Query Time
1An_kiki.pnml		

ready, the number of mod

Fig. 7: Screenshot of the front-end of the APQL Querier as part of BeehiveZ.

3. Each model is stored together with its corresponding TPCFP.

Since the SAP reference model is represented as EPCs, we first transformed it into Petri nets using ProM.[11] This generated 591 Petri nets (13 SAP reference models could not be mapped into Petri nets through ProM). There are at most 1,494 differently labeled transitions out of 4,439 transitions in total, the average number of transactions, places and arcs are 7.5, 12.7 and 19.7 respectively. In the second experiment, we investigate the performance of our technique based on a set of over 100,000 artificially generated process models represented by Petri nets. These models are generated by BeehiveZ based on the rules from [11]. The number of models with different number of transitions follows normal distribution. The number of transitions in one model ranged from 1 to 50, the number of places in one model ranged from 2 to 31, and the number of arcs in one model ranged from 2 to 102[12]. In total, there were 2,643,762 transitions in the repository, and the number of transitions with different labels was 86,961.

The test queries and results are illustrated in Table 1. Specifically, the column Queries shows the sample queries for testing. For readability, we use text statements to describe the query requirements, corresponding queries written by a concrete syntax of APQL can be found in Section Appendix. In these queries, task names in brackets come from the second model collection. The others come from the SAP reference process model set. In these two experiments, each query was executed 12 times, for eliminate the interference of the computer dormancy phenomenon, we remove the highest and lowest value of response time and just evaluate the average of the rest 10 execution time.

Since we build the TPCFP for each model incrementally, when a new model is added into the repository, the corresponding TPCFP can be submitted immediately, and the time for TPCFP updating is very short. The storage size of the TPCFPs (including the behavior feature indexes and the label index) is less than 200% of the storage size of models. With respect to the performance of this technique, it is worth spending the space cost. Now, we can draw a conclusion that our approach works well.

[11] `wwww.processmining.org`

[12] According to 7PMG proposed in [24], models should be decomposed if they have more than 50 elements. That's why we generated models with the maximum number of transitions as 50, the number of places and arcs in a model is not configurable.

Table 1: Time cost and degree of accuracy of Experiment I .

		NR		RT	
	Queries	**SAP**	**AG**	**SAP**	**AG**
q_{11}	List all processes where task [Program Analysis (/t2Q)] occurs in every process instance	3	2	24.6ms	9.4s
q_{12}	List all processes where in every process instance task [Order Release (/Z5j)] occurs before task [Cost Reposting (/WW6)]	1	2	31.7ms	27.8s
q_{13}	List all processes where in every process instance every execution of task [Order Release (/Z5j)] occurs before an execution of task [Cost Reposting (/WW6)]	1	1	44.6ms	23.8s
q_{14}	List all processes where in every process instance every execution of task [Periodic Settlement (/HcG)] is succeeded by an execution of task [Settlement Account Assignment (/Z6)] before [Periodic Settlement (/HcG)] can occur again	1	1	41.9ms	25.2s
q_{15}	List all processes where task [Subsequent Acquisition (/IU)] occurs before [Processing of Asset Acquisition (/Rp)]	1	1	37.1ms	34.3s
q_{16}	List all processes where task [Subsequent Acquisition (/IU)] does not occur before [Processing of Asset Acquisition (/Rp)]	1	0	52.5ms	49.5s
q_{17}	List all processes where task [Measure Processing (/t)] concurs with task [Measure Planning (/AF)]	2	2	49.2ms	56.7s
q_{18}	List all processes where either task [Measure Processing (/t)] or task [Measure Planning (/AF)] occurs	0	1	57.9ms	43.5s
q_{19}	List all processes where in every process instance the immediate successors of an execution of task [Order Release (/Op)] include an execution of [Cost Reposting (/A)] and an execution of task [Reservation (/Gdx)]	1	1	82.3ms	89.2s
q_{20}	List all processes where in every process instance the immediate successors of an execution of [Measure Processing (/t)] are identical to the immediate successors of an execution of [Measure Planning (/AF)].	2	1	91.7ms	124.6

Note:
[*] NR – Number of results.
[**] RT – Response time averaged on 10 executions.
[***] AG – Artificial generated process models.

5.2 Structured interviews

here it goes the description of the experiment with practitioners, in case we decide to include it.

6 Related work

With recognition of the importance of query languages for business processses, the Business Process Management Initiative (BPMI) proposed to define a standard query language for business processes in 2004[13]. While such a standard was never published since then, two major research efforts have been so far dedicated to the development of query languages for business processes. One is known as BP-QL [4], a graphical query language based on an abstract representation of BPEL (Business Process Execution language for Web Services[14]) and supported by a formal model of graph grammars (for processing of queries). BP-QL queries process specifications (written in BPEL) rather than possible executions, and ignores the run-time semantics of certain BPEL constructs such as conditional execution and parallel execution.

The other, namely BPMN-Q [2], is also a visual query language, which extends a subset of modeling notations in BPMN (Business Process Modeling Notations[15]) and supports graph-based query processing. Similarly to BP-QL, BPMN-Q queries business process definitions but not run-time behaviour, and only captures the structural (i.e. syntactical) relationships between tasks. In [25], BPMN-Q is extended with two semantical operators, 'precedes' and 'leads to', between two individual tasks, for compliance checking purposes. In the mean time, in [26], the authors explore the use of an information retrieval technique to derive similarities of activity names, and develop an ontological expansion of BPMN-Q to tackle the problem of querying business processes that are developed with different terminologies. A framework of tool support for querying using BPMN-Q and its extensions is presented in [27].

As compared to the above, the query language APQL proposed in this paper has the following three distinguished features:

- It is independent of a specific process modelling approach (such as BPEL or BPMN). The abstract syntax of APQL allows it to be implemented in a concrete syntax and used for any existing process repository (such as BeehiveZ or APROMORE [28]).
- It is based on execution semantics, rather than the structural information, of process specifications[16]. Such a language can be more intuitive to use by stakeholders who are not necessarily modelling experts. More importantly, querying based on process structure may not capture the correct ordering relationship between tasks, in particular with respect to cyclic process structures (recall the discussions in Section 3).
- It supports rich querying constructs and is more expressive. The querying constructs capture all possible temporal-ordering relations (precedence/succession, concurrence and exclusivity) between individual tasks, and are also extended to capture relations between an individual task and a set of tasks as well as between different sets of tasks.

In addition to the development of process query languages, researchers propose other approaches and techniques that are used or can be useful for querying business processes. In [29, 30] the authors focused on querying the content of business processes based on metadata search. VisTrails system [31] allows users to query scientific workflows by example and to refine workflows by analogies. WISE [32] is a workflow information search engine which supports keyword search on workflow hierarchies. In [33] the authors use graph reduction techniques to find a match to the query graph in the process graph for querying process variants, and the approach however works on acyclic graphs only. In [34–36], a group of similarity-based techniques have been proposed which can be used to support process querying. In [37], the notion of behavioural profile of a process model is defined, which captures dedicated behavioural relations like exclusiveness or potential occurrence of activities. However, since the behavioural relations are derived from the structural information of a process model, they may not cover all necessary relations, in particular with respect to cyclic process models. Also, the approach requires the process model be a sound free-choice Petri net.

[13] http://www.bpmi.org/downloads/BPMI_Phase_2.pdf

[14] http://www.ibm.com/developerworks/library/specification/ws-bpel/

[15] http://www.bpmn.org/

[16] In our earlier work [**?**] is an initial attempt at defining a query language based on execution semantics of process specifications. The language is written in linear temporal logic (LTL) formula and only supports precedence/succession relations between individual tasks.

7 Conclusions

We propose a expressive and powerful process query language in this paper. It can be used to specify temporal-order behavioral relations between tasks. To decide the temporal ordering relations from models efficiently, we adapt the technology of CFP to TPCFP. For efficiency, we use indexes to support the query processing. A system has been implemented and experiments show that our approach works well.

In this paper, we only focus on the control flow perspective of business process models. In the future, we will extend our work to include the data and resource information as well.

Acknowledgements

The work by Song and Wang is supported by the National Basic Research Program of China (2009CB320700), the National High-Tech Development Program of China (2008AA042301), the Project of National Natural Science Foundation of China (90718010), and the Program for New Century Excellent Talents in University of China. The work of ter Hofstede and La Rosa is supported by the ARC Linkage Grant "Facilitating Business Process Standardization and Reuse" (LP110100252).

References

1. China cnr corporation limited. http:// www.chinacnr.com / LISTS / product / _MAINPAGE / englishmain /,2008.
2. A. Awad. Bpmn-q: A language to query business processes. In M. Reichert, S. Strecker, and K. Turowski, editors, *Enterprise Modelling and Information Systems Architectures - Concepts and Applications , Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA'07), St. Goar, Germany, October 8-9, 200*, volume P-119 of *LNI*, pages 115–128. GI, 2007.
3. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 343–354. ACM, 2006.
4. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with bp-ql. *Inf. Syst.*, 33:477–507, September 2008.
5. OMG. *Business Process Model and Notation (BPMN) ver. 2.0*, January 2011. http://www.omg.org/spec/BPMN/2.0.
6. K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G von Bochmann and D. K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177, London, UK, 1993. Springer-Verlag.
7. K. L. McMillan. A technique of state space search based on unfolding. *Form. Methods Syst. Des.*, 6(1):45–65, 1995.
8. J. Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106, London, UK, 1996. Springer-Verlag.
9. J. Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. *Form. Methods Syst. Des.*, 20(3):285–310, 2002.
10. B. Meyer. *Introduction to the theory of programming languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
11. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.
12. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In G. Kahn, editor, *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*, volume 70 of *Lecture Notes in Computer Science*, pages 266–284, London, UK, 1979. Springer-Verlag.
13. A. Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
14. P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.
15. W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
16. W. M. P. van der Aalst. Petri-net-based workflow management software, 1996.
17. S. A. White. *BPMN: modeling and reference guide*. Future Strategies, Incorporated, 2008.

18. Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007.
19. G. Keller, M. Nüttgens, and A. W. Scheer. Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). Technical Report 89, Universität des Saarlandes, Germany, Saarbrücken, Germany, January 1992.
20. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.
21. J. Engelfriet. Branching processes of petri nets. *Acta Inf.*, 28(6):575–591, 1991.
22. T. Jin, J. Wang, and L. Wen. Querying business process models based on semantics. In J. X. Yu, M. Kim, and R. Unland, editors, *Database Systems for Advanced Applications - 16th International Conference, DASFAA 2011, Hong Kong, China, April 22-25, 2011, Proceedings, Part II*, volume 6588 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2011.
23. N. Lohmann, E. Verbeek, and R. Dijkman. Transactions on petri nets and other models of concurrency ii. chapter Petri Net Transformations for Business Processes — A Survey, pages 46–63. Springer-Verlag, Berlin, Heidelberg, 2009.
24. J. Mendling, H. A. Reijers, and W. M. P. van der Aalst. Seven process modeling guidelines (7pmg). *Inf. Softw. Technol.*, 52:127–136, February 2010.
25. A. Awad, G. Decker, and M. Weske. Efficient compliance checking using bpmn-q and temporal logic. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *BPM '08: Proceedings of the 6th International Conference on Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.
26. A. Awad, A. Polyvyanyy, and M. Weske. Semantic querying of business process models. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 85–94. IEEE Computer Society, 2008.
27. S. Sakr and A. Awad. A framework for querying graph-based business process models. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 1297–1300, New York, NY, USA, 2010. ACM.
28. M. L. Rosa, H. A. Reijers, W. M.P. van der Aalst, R. M. Dijkman, J. Mendling, M. Dumas, and L. Garcia-Banuelos. Apromore : an advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040, 2011.
29. J. Vanhatalo, J. Koehler, and F. Leymann. Repository for business processes and arbitrary associated metadata. In *Demo Session of the 4th International Conference on Business Process Management*, pages 25–31, Vienna, Austria, 2006.
30. A. Wasser, M. Lincoln, and R. Karni. ProcessGene Query - a tool for querying the content layer of business process models. In *Demo Session of the 4th International Conference on Business Process Management*, pages 1–8, Vienna, Austria, 2006.
31. C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with vistrails. In J. T. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, SIGMOD '08, pages 1251–1254, New York, NY, USA, 2008. ACM.
32. Q. Shao, P. Sun, and Y. Chen. Wise: A workflow information search engine. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1491 –1494, 2009.
33. R. Lu and S. W. Sadiq. Managing process variants as an information resource. In *Proceedings of the 4th International Conference on Business Process Management*, pages 426–431, Vienna, Austria, 2006.
34. W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters. Process equivalence: Comparing two process models based on observed behavior. 4102:129–144, 2006.
35. M. Ehrig, A. Koschmider, and A. Oberweis. Measuring similarity between semantic business process models. In John F. Roddick and Annika Hinze, editors, *Conceptual Modelling 2007, Proceedings of the Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM2007), Ballarat, Victoria, Australia, January 30 - February 2, 2007, Proceedings*, volume 67 of *CRPIT*, pages 71–80, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.
36. B. Dongen, R. Dijkman, and J. Mendling. Measuring similarity between business process models. In Z. Bellahsene and M. Léonard, editors, *Advanced Information Systems Engineering, 20th International Conference, CAiSE 2008, Montpellier, France, June 16-20, 2008, Proceedings*, volume 5074 of *Lecture Notes in Computer Science*, pages 450–464, Berlin, Heidelberg, 2008. Springer-Verlag.
37. M. Weidlich, J. Mendling, and M. Weske. Efficient consistency measurement based on behavioral profiles of process models. *IEEE Transactions on Software Engineering*, 37:410–429, May/June 2011.