Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2013, Article ID 962765, 24 pages
http://dx.doi.org/10.1155/2013/962765

Research Article

Efficient Semantics-Based Compliance Checking Using LTL Formulae and Unfolding

Liang Song,[1,2,3,4] Jianmin Wang,[1,3,4] Lijie Wen,[1,3,4] and Hui Kong[1,2]

[1] *School of Software, Tsinghua University, Beijing 100084, China*
[2] *Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*
[3] *Key Lab for Information System Security, Ministry of Education, Beijing 100084, China*
[4] *National Laboratory for Information Science and Technology, Beijing 100084, China*

Correspondence should be addressed to Jianmin Wang; jimwang@tsinghua.edu.cn

Received 6 February 2013; Accepted 26 March 2013

Academic Editor: Xiaoyu Song

Business process models are required to be in line with frequently changing regulations, policies, and environments. In the field of intelligent modeling, organisations concern automated business process compliance checking as the manual verification is a time-consuming and inefficient work. There exist two key issues for business process compliance checking. One is the definition of a business process retrieval language that can be employed to capture the compliance rules, the other concerns efficient evaluation of these rules. Traditional syntax-based retrieval approaches cannot deal with various important requirements of compliance checking in practice. Although a retrieval language that is based on semantics can overcome the drawback of syntax-based ones, it suffers from the well-known state space explosion. In this paper, we define a semantics-based process model query language through simplifying a property specification pattern system without affecting its expressiveness. We use this language to capture semantics-based compliance rules and constraints. We also propose a feasible approach in such a way that the compliance checking will not suffer from the state space explosion as much as possible. A tool is implemented to evaluate the efficiency. An experiment conducted on three model collections illustrates that our technology is very efficient.

1. Introduction

Business process models are valuable intellectual assets capturing the ways organisations conduct their business. Current business process management evolves increasingly fast due to changing environments and emerging technologies. As a result, organisations accumulate huge numbers of business process models, and among these may be models with high complexity. For example, Haier is one of the largest Chinese consumer electronics manufacturers. Over the years, Haier has gathered more than 4,000 process models from various domains, including purchase, financing, distribution, and service. In this context, support for business process management, for example, for the purposes of knowledge discovery and process reuse, faces real challenges. In order to stand a competitive advantage, one of these challenges concerns business process compliance checking to make sure that business processes are in line with frequently changing business environments and legal regulations. This problem has also gradually emerged as an important branch of intelligent modeling. There are two key issues must be addressed for automated business process compliance checking. One is a retrieval language that can be employed to capture compliance rules, the other is the efficient evaluation of compliance checking.

In recent years, there are some query languages have been proposed to retrieve process models in repositories, such as BP-QL [1] and BPMN-Q [2]. In [3], BPMN-Q was also used to capture compliance rules. But these languages are based on syntax (structure) of process models, rather than on semantics of them. While in the syntax of a process model, a directed path connecting a task A and a task B does not mean that during execution task A will always occur before task B. Let us consider, for example, the three process models in Figure 2. Among of them, rectangles represent tasks, arcs represent sequential dependencies between tasks, while diamonds represent choices (if each of the diamonds has one

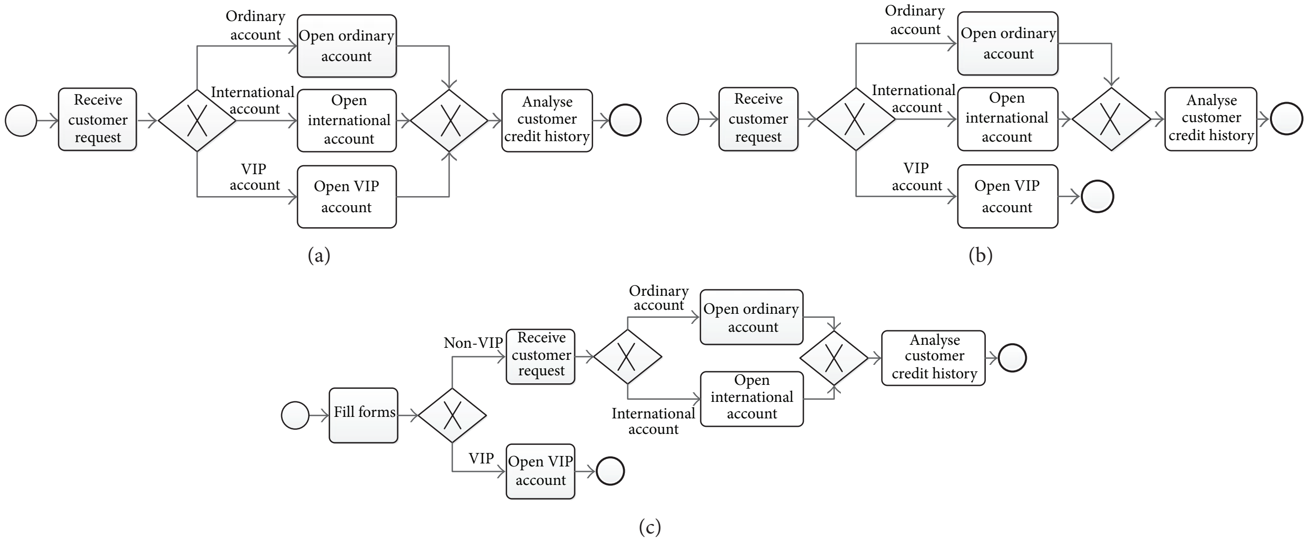


Figure 1: Three variants of a business process for opening bank accounts.

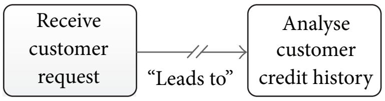


Figure 2: A query written in BPMN-Q.

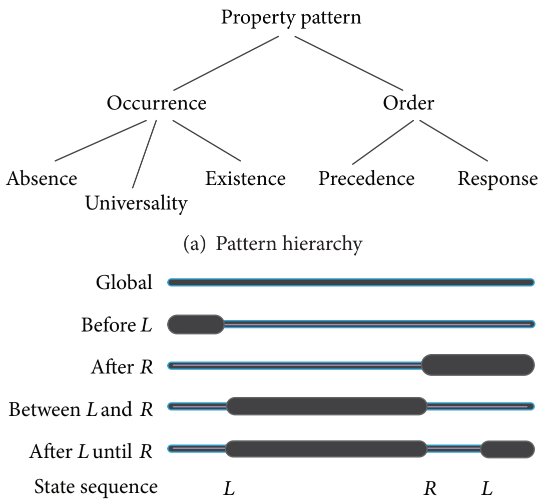
input arc and multiple output arcs) and merges (if each of the diamonds has multiple input arcs and one output arc). These models represent three variants of a business process for opening an account in the BPMN notation [4]. These three variants could specify the way an account is opened in three different states in which the company conducts its business and could be part of a repository of hundreds, even thousands of process models for all states in which the bank operates. Next, let us take BPMN-Q as an example to illustrate the drawback of syntax-based languages. A rule written in BPMN-Q uses a directed edge connecting two activities to represent that these two activities are executed in order (in just some executions of a process). For example, the BPMN-Q query, as shown in Figure 2, can specify the compliance rule that task "receive customer request" must always be followed by task "analyse customer credit history" in some process executions. But if an analyst requires to retrieve processes where in every process execution task "receive customer request" always occurs before task "analyse customer credit history," BPMN-Q cannot capture this kind of requirements. Thus, after executing the query in Figure 2, we would retrieve the first and the third processes, since in both process (a) and process (c), there exists at least one process execution in which if task "receive customer request" occurs, then task "analyst customer credit history" would eventually occur. However, according to the requirement, process (c) does not belong to the result as process (c) has an execution where that task "receive customer request" always precedes task "analyst customer credit history" does not hold (the process execution where task "open VIP account" is run). As a result, the problem of BPMN-Q is that people cannot know whether all of the process executions of a resulting process satisfy the requirement, or just some of them satisfy the requirement. This issue is very important in reuse of business processes, automatic modeling, and verification. For example, in reuse of business process, people often need to know whether there are some process executions that fail to satisfy a requirement, with the goal to check the reason and modify these process executions. Therefore, in order to yield correct result, we have to explore every process execution of every process in a repository, which is indeed based on semantics.

As we can see from the example, syntax-based retrieval languages are not powerful enough. In fact, retrieval technologies based on semantics are indeed in line with process execution and therefore are more intuitive to ordinary users who are not necessarily experts in business process management (BPM). A semantics-based process model query language should capture two types of requirements: (1) it can specify various semantic relationships between tasks; (2) it can explicitly specify that these relationships hold in just some process execution or in every process execution.

In light of the previous, in this paper, we aim to address two questions. One is that how many the semantic relationships between tasks are enough; the other is that the evaluation of semantics-based compliance rules requires to explore every process execution of a process model, which suffers from the well-known state space explosion problem.

In [5], a property specification pattern system (SPS) has been proposed for finite-state verification by Dwyer and so forth. SPS consists of 5 basic patterns and 5 scopes, which results in 5 × 5 = 25 LTL formulae. In this paper, we significantly simplify SPS without affecting its expressiveness through formal logic reasoning. After this simplification, there are only 3 basic LTL formulae from which the rest formulae can be deduced. A retrieval language for expressing semantics-based compliance rules is based on this simplified SPS. With respect to the evaluation of semantics of process models, we proposed a feasible technology which can extract every execution of a business process model. In such a way, the state space explosion can be avoided as much as possible. We achieve this by adopting the theory of *complete finite prefixes* (CFP) [6] and its improvements [7]. Moreover, a tool is implemented to evaluate the performance of our technology over three collections of Petri nets. For the three collections, two are obtained from practice, and the third is a much larger one and obtained by artificially generating.

The remainder of this paper is organized as follows. In Section 2 we simplify SPS to define a process model retrieval language for specifying compliance rules. While in Section 3 the basic concepts of Petri nets, unfolding, and CFP are presented. In Section 4, we detail the mechanism of efficient semantics-based compliance checking. Next, in Section 5 we illustrate the tool implementation and report on the performance evaluation over three process model collections. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.



(a) Pattern hierarchy

(b) Scopes

Figure 3: The pattern hierarchy and the scopes in SPS.

2. Language

As discussed in Section 1, a language is needed for specifying semantics-based compliance rules. This language should be powerful enough while being not too complex. In this section, we will simplify the SPS to obtain a core pattern system from which the rest patterns and scopes of SPS can be derived. Then we present the formal definition of a new query language, namely, "a semantics-based process query language" (ASBPQL), based on this core pattern system.

2.1. LTL Formulae. Linear temporal logic (LTL) is a widely used formalism for specifying properties of concurrent, finite-state systems. In this subsection, we use LTL to reason about the core of SPS.

Definition 1 (linear temporal logic formulae). The formulae of linear temporal logic are built from a finite set of atomic propositions P, the logical operators $\neg$, $\wedge$, and $\vee$, and the temporal modal operators $\bigcirc$ and U. Formally, the set of LTL formulae over P can be inductively defined as follows:

(i) both true and false are LTL formulae;

(ii) for all $p \in P$, p and $\neg p$ are LTL formulae;

(iii) if φ_1 and φ_2 are both LTL formulae, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi_1$, $\varphi_1 U \varphi_2$, and $\varphi_1 R \varphi_2$ are LTL formulae.

The operator $\bigcirc$ is read as "*next*" and denotes in the next state. The operator U is read as "*until*" and means that its first argument has to hold until its second argument is true, where it is required that the second argument holds eventually (some literatures also define the weak *until* operator (W) which related to the strong *until* operator (U) through the following equivalences: $Wq \equiv (pUq) \vee p \equiv pU(q \vee \Box p) \equiv (\Box p) \vee (p\ U\ q) \equiv p\ U\ (q \vee \Box p)$, $pUq \equiv \Diamond q \wedge (p\ W\ q)$). The operator R is read as "*releases*" and is the dual of U. In addition, two derived operators are in common use. They are as follows:

(i) $\Diamond$ is read as "*eventually,*" $\Diamond\varphi = true\, U\, \varphi$, which requires that its argument be true eventually, that is, at some states in the future;

(ii) $\Box$ is read as "*always,*" $\Box\varphi = false\, R\, \varphi$, which requires that its argument be true at all future states.

2.2. Simplification. SPS consists of 5 basic patterns (the other three patterns are defined based on them) and 5 scopes, as shown in Figure 3. The intents of the 5 basic patterns are as follows:

(i) *Absence*, a given task never occurs within a scope;

(ii) *Universality*, a given task occurs throughout a scope;

(iii) *Existence*, a given task occurs at least once within a scope;

(iv) *Precedence*, a task P occurs before a task T within a scope;

(v) *Response*, a task P must be followed by a task T within a scope.

The meanings of the five scopes are presented as follows:

(i) *Global* means the entire extent of a process execution;

(ii) *Before L* means the extent up to an occurrence of the given task L within a process execution;

(iii) *After R* means the extent after an occurrence of the given task L within a process execution;

(iv) *Between L and R* means the part of a process execution from an occurrence of the task L and that of the task R;

(v) *After L until R* is similar to the scope *Between L and R* except that the designated part of a process execution continues if the task R does not occur.

Table 1: LTL formulae in SPS.

Pattern	Scope	LTL Formula	Pattern	Scope	LTL Formula
	Global	$\Box(\neg P)$		Global	$\Box P$
	Before R	$\Diamond R \rightarrow ((\neg P)\ U\ R)$		Before R	$\Diamond R \rightarrow (P\ U\ R)$
Absence	After L	$\Box(L \rightarrow \Box(\neg P))$	Universality	After L	$\Box(L \rightarrow \Box P)$
	Between L and R	$\Box((L \wedge (\neg R) \wedge \Diamond R) \rightarrow ((\neg P)\ U\ R))$		Between L and R	$\Box((L \wedge (\neg R) \wedge \Diamond R) \rightarrow (P\ U\ R)))$
	After L until R	$\Box(L \wedge (\neg R) \rightarrow ((\neg P)\ W\ R))$		After L Until R	$\Box((L \wedge (\neg R)) \rightarrow (P\ W\ R)))$
	Global	$\Diamond P$		Global	$(\neg P)\ W\ T$
	Before R	$(\neg R)\ W\ (P \wedge (\neg R))$		Before R	$\Diamond R \rightarrow ((\neg P)\ U\ (T \vee R))$
Existence	After L	$\Box(\neg L) \vee \Diamond(L \wedge \Diamond P)$	Precedence	After L	$\Box(\neg L) \vee \Diamond(L \wedge ((\neg P)\ W\ T))$
	Between L and R	$\Box((L \wedge (\neg R)) \rightarrow ((\neg R)\ W\ (P \wedge (\neg R))))$	(T precedes P)	Between L and R	$\Box((L \wedge (\neg R) \wedge \Diamond R) \rightarrow ((\neg P)\ U\ (T \vee R)))$
	After L until R	$\Box((L \wedge (\neg R)) \rightarrow ((\neg R)\ U\ (P{\wedge}(\neg R))))$		After L until R	$\Box(L \wedge (\neg R) \rightarrow ((\neg P)\ W\ (T \vee R)))$
	Global	$\Box(P \rightarrow \Diamond T)$			
	Before R	$\Diamond R \rightarrow (P \rightarrow ((\neg R)\ U\ (T \wedge (\neg R))))\ U\ R$			
Response	After L	$\Box(L \rightarrow \Box(P \rightarrow \Diamond T))$			
(P responds T)	Between L and R	$\Box((L \rightarrow (\neg R) \wedge \Diamond R) \rightarrow (P \rightarrow ((\neg R)\ U\ (T \wedge (\neg R))))\ U\ R)$			
	After L until R	$\Box((L \wedge (\neg R)) \rightarrow (P \rightarrow ((\neg R)\ U\ (T \wedge (\neg R))))\ W\ R)$			

As shown in Table 1, for each scope there is an LTL formula corresponding to a pattern, which results in 25 formulae.

Next, we provide proofs that the SPS can be simplified from 5 patterns and 5 scopes to only 3 patterns (*Absence*, *Existence*, and *Precedence*) and 1 scope (*After L until R*). This can significantly reduce the number of formulae from 25 to 3.

First, we take pattern *Absence* as an example to prove that scope *Before R* can be derived from scope *After L until R*. According to the semantics of LTL, if L is always true, that is, $\Box L$, scope *Before R* can be derived from scope *After L until R*, that is, $\Box L \wedge \Box(L \wedge (\neg R)) \rightarrow ((\neg P)\ WR) \Rightarrow \Diamond R \rightarrow ((\neg P)\ U\ R)$. Now we prove that this proposition holds.

Proposition 2. *Consider* $\Box L {\wedge} \Box(L {\wedge} (\neg R)) \rightarrow ((\neg P)\ W\ R) \Rightarrow \Diamond R \rightarrow ((\neg P)\ U\ R)$.

Proof. By contradiction, assume $(\Box L \wedge \Box(L \wedge (\neg R)) \rightarrow ((\neg P)\ W\ R)) \wedge \neg(\Diamond R \rightarrow ((\neg P)\ U\ R))$:

(1) $\neg((\neg \Diamond R) \vee ((\neg P)\ U\ R))$ (by assumption),

(2) $\Box L$ (given),

(3) $\Box((L \wedge (\neg R)) \rightarrow ((\neg P)\, W\ R))$ (given),

(4) $\Diamond R \wedge \neg(\Diamond R \wedge ((\neg P) W\ R))$ (by (1)),

(5) $\Diamond R \wedge (\neg(\Diamond R) \vee \neg((\neg P)\, W\ R))$ (by (4)),

(6) $(\Diamond R \wedge \neg(\Diamond R)) \vee (\Diamond R \wedge \neg((\neg P)\ W\ R))$ (by (5)),

(7) $\Diamond R \wedge \neg((\neg P)\ W\ R)$ (by (6)),

(8) $\Diamond \mathrm{R}$ (by (7)),

(9) $\neg((\neg P)\ W\ R)$ (by (7)),

(10) $L \wedge (\neg R) \rightarrow ((\neg P)\ W\ R)$ (by (3)),

(11) L (by (2)),

(12) $(\neg R) \rightarrow ((\neg P)\ W\ R)$ (by (10), (11)),

(13) R (by (9), (12)),

(14) $\neg P\ U\ R$ (by (13)),

(15) $\neg(((\neg P)\ W\ R) \vee \Box(\neg P))$ (by (9)),

(16) $\neg((\neg P)\ U\ R)$ (by (15)).

By (14), (16), we get a contradiction. So, we conclude that proposition $\Box L \wedge \Box(L \wedge (\neg R)) \rightarrow ((\neg P)\ W\ R) \Rightarrow \Diamond R \rightarrow ((\neg P)\ U\ R)$ holds. $\square$

Next, if R is always false, that is, $\Box(\neg R)$, we can prove that for pattern *Absence* the formula corresponding to scope *After L* can be derived from the formula corresponding to scope *After L until R*.

Proposition 3. *Consider* $\Box(\neg R) \wedge \Box((L \wedge (\neg R)) \rightarrow ((\neg P)\, W\, R)) \Rightarrow \Box(L \rightarrow \Box(\neg P))$.

Proof. One has

(1) $\Box(\neg R)$ (given),

(2) $\Box((L \wedge (\neg R)) \rightarrow ((\neg P)\ W\ R))$ (given),

(3) $\neg R$ (by (1)),

(4) $L \wedge (\neg R) \rightarrow ((\neg P)\ W\ R)$ (by (2)),

(5) $L \rightarrow ((\neg P)\ W\ R)$ (by (3), (4)),

(6) $L \rightarrow (((\neg P)\ U\ R) \vee \Box(\neg P))$ (by (5)),

(7) $\neg(\Diamond R)$ (by (1)),

(8) $\neg(\Diamond R) \vee (\neg((\neg P)\ W\ R))$ (by (7)),

(9) $\neg(\Diamond R \wedge ((\neg P)\ W\ R))$ (by (8)),

(10) $\neg((\neg P)\ U\ R)$ (by (9)),

(11) $L \rightarrow \Box(\neg P)$ (by (6), (10)),

(12) $\Box(L \rightarrow \Box(\neg P))$ (by (11)).

□

Next, we prove that if R holds eventually, that is, $\Diamond R$, we can derive the formula corresponding to scope *Between L and R* from the formula corresponding to scope *After L until R*.

Proposition 4. *Consider* $\Diamond R \wedge \Box((L \wedge (\neg R)) \rightarrow ((\neg P)\ W\ R)) \Rightarrow \Box((L \wedge (\neg R) \wedge \Diamond R) \rightarrow ((\neg P)\ U\ R))$.

Proof. One has

(1) $\Box((L \wedge (\neg R)) \rightarrow ((\neg P)\ W\ R))$ (given),

(2) $L \wedge (\neg R) \rightarrow ((\neg P)\ W\ R))$ by (1),

(3) $\Diamond R$ (given),

(4) $L \wedge (\neg R) \wedge \Diamond R \rightarrow ((\neg P)\ W\ R) \wedge \Diamond R$ (by (2), (3)),

(5) $L \wedge (\neg R) \wedge \Diamond R \rightarrow ((\neg P)\ U\ R)$ (by (4)),

(6) $\Box(L \wedge (\neg R) \wedge \Diamond R \rightarrow ((\neg P)\ U\ R))$ (by (1), (5)).

□

Now we have proved that the formulae corresponding to three scopes (*After L*, *Before R*, and Between L and R) can be derived from the formulae corresponding to scope *After L until R*. If L always holds and R always does not hold, that is, $\Box L \wedge \Box(\neg R)$, the formula corresponding to scope *Global* can be derived from that of scope *After L until R*. This proof is straightforward and is easy to be reasoned about. For page limit, we do not present it in this paper.

Next, we prove that only pattern *Absence*, *Existence*, and *Precedence* are core patterns, the rest patterns in SPS can be derived from these three patterns. Firstly, when we replace $\neg P$ in the formulae corresponding to pattern *Absence* with P, and the formulae corresponding to pattern *Universality* can be derived. Pattern *Absence* and *Universality* are dual of each other. Next, we present as follows the explicit proofs of the derivation of pattern *Response* from pattern *Absence* and *Existence*, in scope *After L until R*. Lemmas 5 and 6 will be used in this reasoning.

Lemma 5. *Consider* $(\neg R)\ U\ (P \wedge (\neg R)) \Leftrightarrow ((\neg R)\ U\ (P \wedge (\neg R)))\ W\ R$.

Proof. One has

$$
\begin{aligned}
&(\Longrightarrow)\\
&(\neg R)\, U\, (P \wedge (\neg R))\\
&\Longrightarrow ((\neg R)\, U\, (\neg R)) \wedge ((\neg R)\, U\, P)\\
&\Longrightarrow \Box\, (\neg R) \wedge ((\neg R)\, U\, P)\\
&\Longrightarrow ((\neg R)\, U\, (P \wedge (\neg R)))\, W\, R\\
&(\Longleftarrow)\\
&((\neg R)\, U\, (P \wedge (\neg R)))\, W\, R\\
&\Longrightarrow (((\neg R)\, U\, (P \wedge (\neg R)))\, U\, R)\\
&\quad \vee \Box\, ((\neg R)\, U\, (P \wedge (\neg R)))\\
&\Longrightarrow (\Diamond R \wedge ((\neg R)\, W\, (P \wedge (\neg R))))\\
&\quad \vee ((\neg R)\, U\, (P \wedge (\neg R)))\\
&\Longrightarrow \Diamond R \wedge ((\neg R)\, U\, (P \wedge (\neg R)))\\
&\Longrightarrow (\neg R)\, U\, (P \wedge (\neg R))\,.
\end{aligned}
\tag{1}
$$

□

Lemma 6. *Consider* $((\neg P)\ W\ R) \vee ((\neg R)\ U\ (T \wedge (\neg R))) \Rightarrow (P \rightarrow ((\neg R)\ U\ (T \wedge (\neg R))))\ W\ R$.

Proof. By Lemma 5,

$$
\begin{aligned}
&((\neg P)\ \ W\ \ R) \vee ((\neg R)\ \ U\ \ (T \wedge \neg R))\\
\Longleftrightarrow\ &((\neg P)\ \ W\ \ R) \vee ((\neg R)\ \ U\ \ (T \wedge \neg R))\, W\, R\\
\Longleftrightarrow\ &((\neg P) \vee ((\neg R)\ \ U\ \ (T \wedge (\neg R))))\, W\, R\\
\Longleftrightarrow\ &(P \longrightarrow ((\neg R)\ \ U\ \ (T \wedge (\neg R))))\, W\, R.
\end{aligned}
\tag{2}
$$

□

Proposition 7. *Consider* $\Box((L \wedge (\neg R) \rightarrow ((\neg P)\ W\ R) \wedge \Box((L \wedge (\neg R) \rightarrow ((\neg R)\ U\ (T \wedge (\neg R)))) \Rightarrow \Box((L \wedge (\neg R)) \rightarrow (P \rightarrow ((\neg R)\ U\ (T \wedge (\neg R))))\ W\ R$.

Proof. One has

(1) $\Box(L \wedge (\neg R) \rightarrow (\neg P)\ W\ R)$ (given),

(2) $L \wedge (\neg R) \rightarrow (\neg P)\ W\ R$ (by (1)),

(3) $\Box((L \wedge (\neg R)) \rightarrow ((\neg R)\, U\, (T \wedge (\neg R))))$ (given),

(4) $L \wedge (\neg R) \rightarrow ((\neg R)\, U\, (T \wedge (\neg R)))$ (by (3)),

(5) $L \wedge (\neg R) \rightarrow ((\neg P)\, W\, R) \vee ((\neg R)\, U\, (T \wedge (\neg R)))$ (by (2), (4)),

(6) $L \wedge (\neg R) \rightarrow (P \rightarrow ((\neg R)\, U\, (T \wedge (\neg R))))\ W\, R$ (by Lemma 6),

(7) $\Box((L \wedge (\neg R)) \rightarrow (P \rightarrow ((\neg R)\, U\, (T \wedge (\neg R))))\, W\, R)$ (by (6)).

□

Finally, we obtain a simplified pattern system that consists of only 3 patterns (*Absence*, *Existence*, and *Precedence*) and one scope (*After* $\cdots$ *until*), as shown in Figure 4. As we can see, this simplified pattern system is far more concise than SPS.

2.3. Syntax. Based on the simplified SPS, we can define the basic relationships between tasks in ASBPQL. One is *Existence*, and the other is *Precedence*. And two other relationships are in very common use in business process management. One is *Exclusive*, and the other is *Concurrence*. As

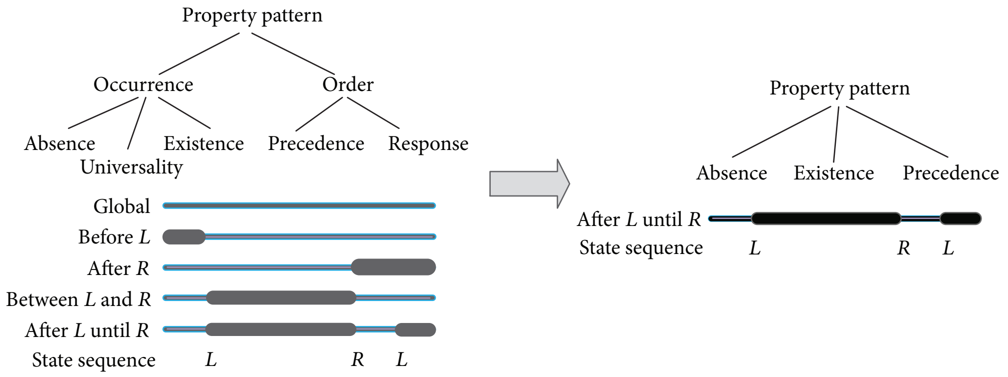


Figure 4: The simplification of SPS.

discussed in Section 1, after defining the basic semantic relationships between tasks, we have to determine whether these relationships hold in just some process executions or in every process execution of a business process. Combining with all these considerations, we can define 6 basic predicates to capture the occurrence of tasks and the relationships between tasks in some or every process execution. In the following, the first two basic predicates, *posoccur* and *alwoccur*, capture the occurrence of a given task in some or every process execution of a process model. These two basic predicates are based on pattern *Existence*:

(1) $posoccur(t_1, r)$: there exists some executions of process model r where at least one instance of t_1 occurs,

(2) $alwoccur(t_1, r)$: in every execution of process model r, at least one instance of t_1 occurs.

The next two basic predicates, *concur* and *exclusive*, capture the concurrent and exclusive relationships between tasks, respectively. Note that these two basic predicates do not assume that an instance of t_1 and t_2 should eventually occur:

(3) $exclusive(t_1, t_2, r)$: t_1 and t_2 are both executable tasks (i.e., not dead tasks) of process model r; in every process execution of r, it is never possible that an instance of t_1 and an instance of t_2 both occur;

(4) $concur(t_1, t_2, r)$: t_1 and t_2 are both executable tasks of process model r; t_1 and t_2 are not causally related; and in every execution of r, if an instance of t_1 occurs, then an instance of t_2 occurs and vice versa.

The last two basic predicates, *pospred* and *alwpred*, capture the basic relationship *Precedence* between tasks in some or every process execution of a given process model, respectively:

(5) $alwpred(t_1, t_2, r)$: in every process execution of process model r, it holds that an instance of t_1 occurs before an instance of t_2;

(6) $pospred(t_1, t_2, r)$: there exists some process executions of process model r where an instance of t_1 occurs before an instance of t_2.

Finally, we define ASBPQL by BNF grammar. A *Query* in ASBPQL is a *Condition*. The result of the *Query* is those process models that satisfy the *Condition*. A *Condition* can consist of ⟨*Task*⟩“*posoccur*,” with the intended semantics what the basic predicate $posoccur(t_1, r)$ specifies, a ⟨*Task*⟩“*alwoccur*,” with the intended semantics what the basic predicate $alwoccur(t_1, r)$ specifies, and a ⟨*TaskRel*⟩, with the intended semantics that all process models satisfying that particular relation between tasks must be retrieved, or it can be recursively defined as a binary or unary *Condition* through the application of logical operators, that is, ⟨*BinCondition*⟩ or ⟨*UnCondition*⟩. Specifically, a disjunction retrieves the union of the process models of the conditions involved, while a conjunction retrieves the intersection. The negation of a condition retrieves the process models that do not satisfy the condition. A *task* can be defined as its label which is a string as follows:

$$\langle \mathit{Query} \rangle ::= \langle \mathit{Condition} \rangle$$

$$\langle \mathit{Condition} \rangle ::= \langle \mathit{Task} \rangle \text{ “}\mathit{posoccur}\text{” } | \langle \mathit{Task} \rangle \text{ “}\mathit{alwoccur}\text{”} |$$

$$\langle \mathit{TaskRel} \rangle \, | \langle \mathit{BinCondition} \rangle | \, \langle \mathit{UnCondition} \rangle ,$$

$$\langle \mathit{Task} \rangle ::= \text{“““} \langle \mathit{TaskLabel} \rangle \text{”””},$$

$$\langle \mathit{TaskLabel} \rangle ::= \text{“} (\sim [\text{”}]) + \text{”},$$

$$\langle \mathit{TaskRel} \rangle ::= \langle \mathit{Task} \rangle \, \langle \mathit{TaskCompOp} \rangle \, \langle \mathit{Task} \rangle ,$$

$$\begin{aligned}\langle \mathit{TaskCompOp}\rangle &::= \text{“}\mathit{concur}\text{”} \mid \text{“}\mathit{exclusive}\text{”} \mid \text{“}\mathit{pospred}\text{”} \mid \text{“}\mathit{alwpred}\text{”},\\ \langle \mathit{BinCondition}\rangle &::= \langle \mathit{Condition}\rangle\, \langle \mathit{BinLogic}\rangle\, \langle \mathit{Condition}\rangle\,,\\ \langle \mathit{BinLogic}\rangle &::= \text{“}\mathit{and}\text{”} \mid \text{“}\mathit{or}\text{”},\\ \langle \mathit{UnCondition}\rangle &::= \text{“}\mathit{not}\text{”}\, \langle \mathit{Condition}\rangle\,.\end{aligned} \tag{3}$$

Using ASBPQL, we can capture the semantics-based compliance rules in which we are interested, including the relationship between tasks and the occurrence of tasks in some or every process execution. For example, rule *“A” pospred “B”* and *“B” alwpred “C”* mean that we want to search for all process models where in some process execution task A occurs before task B and in every execution task B occurs before task C.

3. Petri Nets and Unfoldings

In this section, we discuss the basic concepts of Petri nets and unfolding on which we base our work. For more details, readers can refer to [8] for an in-depth introduction to Petri nets and to [6, 7, 9, 10] for unfolding and its related definitions.

3.1. Petri Nets. Petri nets are a formal notation system which can be employed to specify workflow systems (see, e.g., [11, 12]). Petri nets are also used as a formal foundation for defining the semantics of other process modeling languages or for reasoning about process models specified in these languages, for example, BPMN [13], BPEL [14, 15], and EPCs [16]. A formal definition of Petri nets is presented as follows.

Definition 8 (Petri nets). A Petri net is a tuple (P, T, F), where

(i) P is a finite set of places;

(ii) T is a finite set of transitions, with $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$;

(iii) $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs representing the flow relation, connecting transitions and places together.

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 [7]. 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.

Next, a labeled Petri net is basically a Petri net with annotated transitions and the annotation does not affect the semantics of the net.

Definition 9 (Labeled Petri nets). A labeled Petri net is a tuple (P, T, F, A, L), where

(i) (P, T, F) is a Petri net;

(ii) A is a finite set of task names;

(iii) $L : T \rightarrow A \cup \{\tau\}$ is a label mapping function for T, where $\tau \notin A$ is a silent action (i.e., an action not visible to the outside world).

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 10 (marking, enabling, and firing of a transition). Let $PN = (P, T, F)$ be a Petri net.

(i) A marking M of PN is a mapping $M : P \rightarrow \mathbb{N}$. A marking may be represented as a collection of pairs, for example, $\{(p_0, 2), (p_1, 3), (p_2, 0)\}$ or as a vector, for example, $2p_0 + 3p_1$ (in that case we drop places that do not have any tokens assigned to them). A *labeled Petri net system* is a labeled Petri net with an *initial marking* usually represented as M_0.

(ii) Markings can be compared with each other, $M_1 \geq M_2$ if and only if for all $p \in P$, $M_1(p) \geq M_2(p)$. Similarly, one can define $>, <, \leq, =$.

(iii) A transition t is *enabled* in a marking M, denoted as $M \xrightarrow{t}$, if and only if the following holds: $\forall p \in \bullet t, M(p) > 0$.

(iv) 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. In this paper we consider *n-bounded* Petri net systems (noting that such systems are always finite) which are necessary for the application of unfoldings.

Definition 11 (reachability and boundedness). Let $\Sigma = (P, T, F, M_0)$ be a Petri net system.

(i) A marking M is called *reachable* if a transition sequence $\sigma = t_1 t_2 \cdots t_n$ exists such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \cdots \xrightarrow{t_n} M$, which may also be denoted as $M_0 \xrightarrow{\sigma} M$ or, if the choice of σ does not really matter, $M_0 \xrightarrow{*} M$.

(ii) Σ is called a finite Petri net system if and only if its set of reachable markings is finite.

(iii) Σ is called *n-bounded* if and only if for every reachable marking M and every place $p \in P$: $M(p) \leq n$.

3.2. Unfolding. It is well known that Petri nets may suffer from the *state space explosion* problem [17]. As such a naive exploration of the state space, especially in the context of a Petri net which allows highly concurrent behaviour, may not be tractable. In order to deal with this, McMillan [6] proposed a state space search technique based on the use of *unfolding* (this technique was later on improved by Esparza et al. [7] and is discussed in the next subsection). Unfoldings are applied to *n-bounded* (or called *n-safe* in [7]) Petri net systems and provide a method of searching the state space of concurrent systems without considering all possible interleavings of concurrent events. The concept of unfolding was firstly introduced by Nielsen et al. [9] and later elaborated upon by Engelfriet [10] using the term *branching processes*. In the following we introduce the necessary concepts and notations to make this paper self-contained and to be able to build upon this theory. Most of these defintions are adopted from [7].

Firstly, various types of relationship may hold between pairs of nodes in a Petri net.

Definition 12 (node relations (based on [7])). Let $PN = (P, T, F)$ be a Petri net.

(i) 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, $x_1 < x_2$ if and only if $(x_1, x_2) \in F^+$, and we say that x_1 causally precedes x_2.

(ii) If $x_1 < x_2$ or $x_2 < x_1$, then x_1 and x_2 are causally related.

(iii) Nodes x_1 and x_2 are in conflict, denoted by $x_1 \# x_2$, if and only if there exist distinct transitions $t_1, t_2 \in T$ such that $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, $t_1 \leq x_1$, and $t_2 \leq x_2$. A node x is in self-conflict if and only if $x \# x$.

(iv) Nodes x_1 and x_2 are concurrent, denoted as $x_1 \,\mathtt{co}\, x_2$, if and only if 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 13 (occurrence net (based on [7])). An occurrence net is a net $N' = (B, E, F)$, where

(i) B is a set of conditions;

(ii) E is a set of events, with $B \cap E = \emptyset$;

(iii) $F \subseteq B \times E \cup B \times E$ such that (1) for all $b \in B$, $|\bullet b| \leq 1$, (2) F is acyclic; that is, 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;

(iv) No node is in self-conflict; that is, for all $x \in B \cup E$, $\neg(x \# x)$.

We also adopt the notion of $\mathrm{Min}(N')$, as in [7], 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 least one input and at least one output condition. The minimal elements are therefore conditions only, and intuitively $\mathrm{Min}(N')$ can be seen as an initial marking of the net.

Definition 14 (branching process (based on [10])). A branching process of a Petri net system $\Sigma = (N, M_0)$, with $N = (P, T, F)$, is a pair (N', h), where

(i) $N' = (B, E, F)$ is an occurrence net;

(ii) $h : N' \to N$ is a homomorphism which, following [10], means that

(a) $h(B \cup E) \to (P \cup T)$;

(b) $h \subseteq (B \times P) \cup (E \times T)$; that is, conditions are mapped to places and events to transitions;

(c) 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 $h(t)\bullet$;

(d) $h[\mathrm{Min}(N')]$ is a bijection between $\mathrm{Min}(N')$ and $\{p \in P | M_0(p) > 0\}$;

(iii) 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 [10] it is shown that, up to isomorphism, every net system has a unique maximal branching process. For a net system Σ, this unique process is referred to as the *unfolding* of Σ and it is denoted as $\mathtt{Unf}_\Sigma$. For example, in Figure 5 the Petri nets in (a) can be unfolded into the occurrence net in (b). Note that in Figure 5(b) all (condition/event) nodes are identified by integers and annotated by the corresponding place or transition identifiers in Figure 5(a).

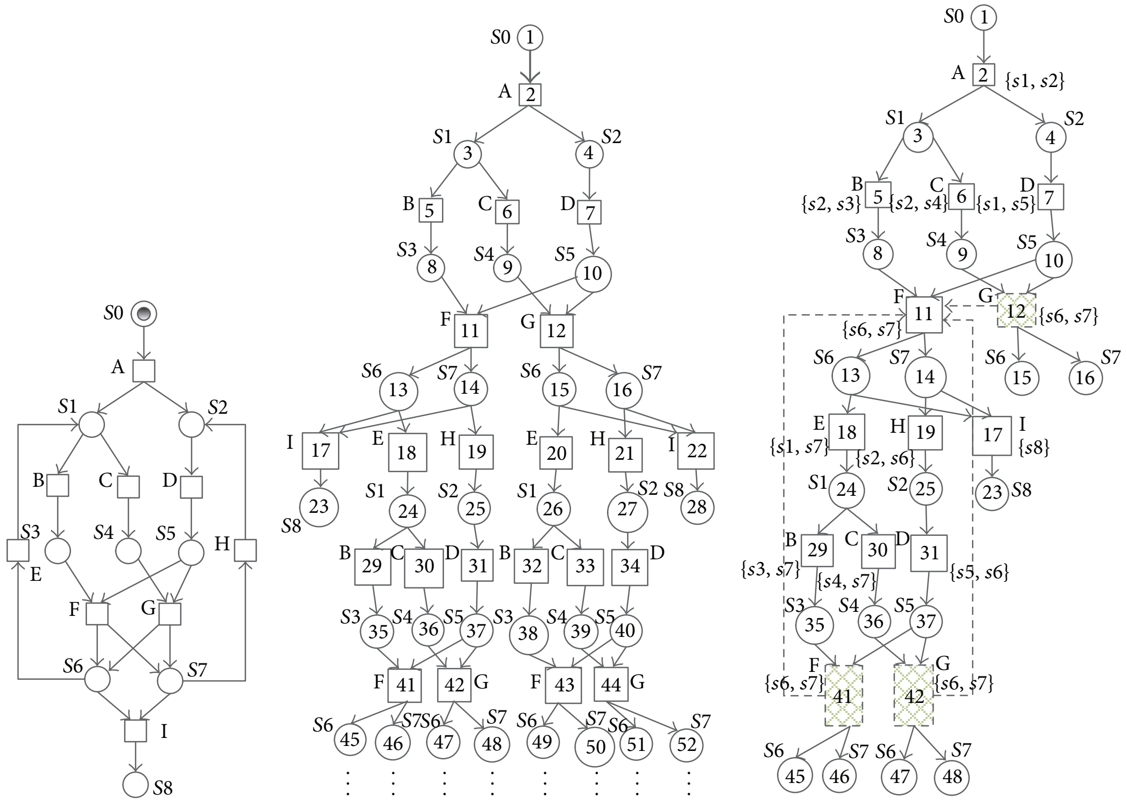
3.3. Complete Finite Prefix. The unfolding of a Petri net is infinite when the net is cyclic, as, for example, $\mathtt{Unf}_\Sigma$ in Figure 5(b). In [6], McMillan proposed an algorithm for the construction of a so-called truncated unfolding, which is a finite initial part of an unfolding and contains as much reachability information as the unfolding itself but may be much larger than necessary. In [7], Ezparza et al. referred to this truncated unfolding as *complete finite prefix* (CFP) and proposed an improved algorithm for computing a minimal CFP. For example, as illustrated in Figure 5(c) (the dashed arcs should be ignored for the moment), $\mathtt{Fin}_\Sigma$ is a minimal CFP of Σ. Note that in Figure 5(c) the tuple of conditions positioned next to an event node 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. Firstly, a configuration represents a possible partially ordered run of the net.

Definition 15 (configuration [7]). A configuration C of an occurrence net $N = (B, E, F)$ is a set of events, that is, $C \subseteq E$, satisfying the following two conditions:

(i) C is causally downward closed, that is, $(e \in C \land e' \leq e) \Rightarrow e' \in C$;

(ii) C is conflict free, that is, for all $e, e' \in C : \neg(e \# e')$.



(a) A Petri net Σ

(b) The unfolding $\mathtt{Unf}_\Sigma$

(c) The CFP $\mathtt{Fin}_\Sigma$ annotated with explicit links from cut-off events to continuation events

Figure 5: Illustration of unfolding and complete finite prefix using the Petri net Σ adapted from [7] (the net in (a) without s_0, s_8, A, and I is the same as the example net in Figure 1 in [7]).

Given a configuration C the set of places Cut_C represents a reachable marking, which is denoted by $Mark(C)$. In other word, $Mark(C)$ is the marking to reach by firing the configuration C. For example, in the unfolding $\mathtt{Unf}_\Sigma$ in Figure 5(b) we have $Mark(\{2, 5, 7, 11, 17\}) = \{s_8\}$.

Definition 16 (cut [7]). Let Σ be a Petri net system, and let (N', h) be its unfolding. The set of conditions associated with a configuration of N' is called a cut and is defined as $Cut_C = (\text{Min}(N') \cup C\bullet) \setminus \bullet C$. A cut uniquely defines a reachable marking in Σ: $Mark(C) = h(Cut_C)$.

The concepts thus far can be used to introduce the unfolding algorithm. In [7] 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 is its input conditions. A set of conditions of a branching process is a *coset* if its elements are pairwise in corelation. For example, in Figure 5(b) each of the node sets $\{13, 14\}$, $\{15, 16\}$, $\{45, 46\}$, $\{47, 48\}$, $\{49, 50\}$, and $\{51, 52\}$ is a coset.

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 coset 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, it is observed that a finite prefix of an unfolding may contain all reachability-related information. The key to obtain a CFP is to identify those events at which we can cease unfolding (e.g., events 12, 41, and 42 in $\mathtt{Fin}_\Sigma$ in Figure 5(c)) 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 17 (adequate order [7]). 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 processes, then $\prec$ is an adequate order if and only if

(i) $\prec$ is well founded;

(ii) for all configurations C_1 and C_2, $C_1 \subset C_2 \Rightarrow C_1 \prec C_2$;

(iii) the $\prec$ order is preserved in the context of finite extensions; that is, 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 [7] for a complete formal definition of this notion. Note that, as pointed out in [7], 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 18 (local configuration [7]). Let $N = (B, E, F)$ be an occurrence net, and 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 19 (cut-off event [7]). Let Σ be a Petri net system, let 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 if and only if 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(c), we remove the part after event 12 from Unf_Σ because it is isomorphic to that after event 11; that is, the continuation after event 12 is essentially the same as the continuation after event 11. For a proof of this approach we refer to [7].

4. Evaluation

In this section, we demonstrate how the basic predicates introduced in Section 2 can be derived for Petri nets based on the process executions extracted from CFPs.

4.1. Annotating Complete Finite Prefix. In this work, the repository of process models are captured in terms of CFPs. All predicates between tasks are determined by examining the possible firing sequences in the CFP of each process model. To facilitate our algorithms for determining these predicates (presented in the next subsection), we would like to represent the continuation from cut-off events slightly more explicit in a CFP. The idea is that for each of the cut-off events e in a CFP we mark out some earlier other event e' that can lead to the same marking as e (i.e., $Mark([e]) = Mark([e'])$ and $[e'] \prec [e]$). We referred to e' as the *continuation event* of e in the CFP. We then annotate the CFP with links that connect from each cut-off event to its continuation event.

Definition 20 (notations of continuation events and cut-off events). 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 the following:

(i) $\texttt{Eq}(M, \rho) = \{e \in \texttt{E}' \mid Mark([e]) = M\}$ for any reachable marking M of N. If ρ is clear from the context, we will simply omit it and write $\texttt{Eq}(M)$ (a similar convention holds for the remainder of this definition, and ρ is not introduced explicitly anymore);

(ii) $\texttt{continuation}(M)$ which refers to the continuation node in ρ for a reachable marking M. It is defined as the unique event $e' \in \texttt{Eq}(M)$ such that for all $e \in \texttt{Eq}(M)$, if $e \neq e'$ then $[e'] \prec [e]$;

(iii) $\texttt{cutoff}(M) = \texttt{Eq}(M) \setminus \{\texttt{continuation}(M)\}$ which denotes the set of cut-off events for a reachable marking M.

Definition 21 (annotated complete finite prefix). Let $\Sigma = (N, M_0)$ be a Petri net system, and $\texttt{Fin}^a_\Sigma$ denotes a CFP of Σ that is annotated with links from cut-off events to their continuation events, shortly referred to as an annotated CFP: $\texttt{Fin}^a_\Sigma = (\texttt{Fin}_\Sigma, L)$, where

(i) $\texttt{Fin}_\Sigma = (B, E, G)$ is the CFP of Σ;

(ii) L is a set of links defined as $L \subseteq E \times E$, and if and only if $(e, e') \in L$, then there is a reachable marking M such that $e' = \texttt{continuation}(M)$ and $e \in \texttt{cutoff}(M)$.

Example 22. Consider $\texttt{Fin}^a_\Sigma$ as shown in Figure 5(c). For this annotated CFP, $L = \{(41, 11), (42, 11), (12, 11)\}$.

To generate an annotated CFP, we propose a slight adaptation of the algorithm for computing a CFP for a n-safe net system in [7]. This adapted algorithm is presented as Algorithm 1. Based on Definition 21, the data structure for the representation of an annotated CFP comprises that of a CFP in [7] (written $\texttt{Fin}^a.N$) and a set of links (written $\texttt{Fin}^a.L$). *PE* ($\texttt{Fin}^a.N$) is the set of events that can be added to a branching process $\texttt{Fin}^a.N$ (i.e., possible extensions of $\texttt{Fin}^a.N$), as defined in [7]. Application of $minimal(pe, \prec)$ yields an event e which satisfies the following condition taken from [7]: $e \in pe$ and $[e]$ is minimal with respect to $\prec$. The predicate $expansion\_required(e, cut\_off)$ is an abbreviation of $[e] \cap cut\_off = \emptyset$, the condition used in [7]. Next, $cut\_off\_event(e, \texttt{Fin}^a.N, c)$ returns the result of whether or not e is a cut-off event of $\texttt{Fin}^a.N$ (as in [7]), and during its application, the corresponding 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 $X := X \cup Y$ and $X \setminus := Y$ for $X := X \setminus Y$.

4.2. Determining the Basic Predicates. In Section 2, we defined a set of 6 basic predicates based on process execution semantics and to check if such a predicate holds requires in principle exploration of *all* process executions. Since different process executions result from choices in a process model, we propose to preprocess the annotated CFP of each process model (Algorithm 2) as follows: first we transform such a CFP to a set of conflict-free CFPs (specified by function *GetAllExecutions* in Algorithms 3) and then convert each resulting CFP to a directed bipartite graph (or bigraph) (specified by *AnnotatedCFP2Bigraph* in Algorithm 5).

In Algorithm 3, `GetLeafCondCoSets` yields all cosets of leaf conditions in the input CFP. By traversing backwards

```
Input: An $n$-safe Petri net system $\Sigma = (P, T, F, M_0)$
Output: $\mathsf{Fin}^a$ $(N : Net,\ L : Links)$ an annotated CFP of $\Sigma$
begin
    $\mathsf{Fin}^a.N := \{(s,\emptyset) \mid M_0(s) > 0 \wedge s \in P\}$;
    $\mathsf{Fin}^a \cdot L := \emptyset$;
    $pe := PE(\mathsf{Fin}^a \cdot N)$;
    $cut\_off := \emptyset$;
    while $pe \neq \emptyset$ do
        $e := minimal(pe, \prec)$;
        If $expansion\_required(e, cut\_off)$ then
            $\mathsf{Fin}^a \cdot N \cup := \{(e.t, \bullet e)\} \cup \{(s, e) \mid s \in e.t\bullet\}$;
            $pe := PE(\mathsf{Fin}^a \cdot N)$;
            if $cut\_off\_event(e, \mathsf{Fin}^a \cdot N, c)$ then
                $cut\_off \cup := \{e\}$;
                $\mathsf{Fin}^a \cdot L \cup := \{(e, c)\}$;
        else $pe \setminus := \{e\}$
```

Algorithm 1: Computation of an annotated CFP via an adaption of Algorithm 4.7 in [7].

```
function $PreProcess$
Input: An annotated CFP $U = (\rho, L)$ where $\rho = (B, E, F)$ and $L \subseteq E \times E$
Output: A set of bigraphs $\mathbb{G}$
begin
    $\mathbb{G} := \emptyset$;
    $\mathbb{U} := GetAllExecutions(U)$;
    for $U \in \mathbb{U}$ do
        $\mathbb{G} \cup := AnnotatedCFP2Bigraph(U)$
```

Algorithm 2: Preprocessing an annotated CFP to a set of directed bigraphs.

the input CFP (without considering the set of links) from each of these co-sets, `ComputeCFPs` produces the set of CFPs as a decomposition of the input CFP. This set of CFPs are free of conflicts due to the corelation between the leaf conditions in each co-set. For illustration, Figure 6 depicts the set of conflict-free CFPs as decomposition of $\mathtt{Fin}_{\Sigma}$ in Figure 5(c) via computation of `GetLeafCondCoSets` and `ComputeCFPs`.

Next, we convert the link annotations of the input CFP to the link annotations for each of the conflict-free CFPs (that result from the above decomposition of the input CFP). If such a CFP does not contain a cut-off event ($E_{cutoff} = \emptyset$), there is no link annotation, and the CFP will remain as it is. Otherwise, for a CFP with cut-off events, there are two cases to consider depending on whether a cut-off event (e_{cut}) in the CFP links to a continuation event (e_{cont}) within or outside this CFP. If the CFP contains both events, the link (e_{cut}, e_{cont}) is directly added into the link annotations of the CFP. Otherwise, if the CFP contains e_{cut} but not e_{cont}, we propose to update the CFP (specified by function `GetUpdatedCFPs` in Algorithm 4) and the link annotations till there exists no link across two different CFPs.

Algorithm 4 specifies how to update a CFP with a cut-off event linking to a continuation event outside the CFP. The basic idea is to identify among the set of conflict-free CFPs (Γ) those (ρ_i) that contain e_{cont} and to replace the part before and including e_{cont} in such a CFP (ρ_i) with the part before and including e_{cut} in the original CFP (ρ). This results in the same number of updated CFPs (ρ') as that of the CFPs containing e_{cont}. Since e_{cont} is replaced by e_{cut} in the updated CFPs and (e_{cut}, e_{cont}) is not used any more, the link annotations need update as well.

Back to Algorithm 3, we retrieve the links (L_{add}) that lead to e_{cont} except for (e_{cut}, e_{cont}) and replace e_{cont} with e_{cut} in these links. Accordingly, the flag f_{update} is set to TRUE signaling the fact that CFP updates have been applied, and the updated CFPs are added to the set of remaining CFPs (Γ_{tmp}) for processing of link annotations. For a given CFP (ρ'), if all the cut-off events in the CFP are processed without CFP updates ($\neg f_{update}$), the set of links (L') that are computed from such processing is added as the CFP's link annotations. The previous procedure for converting link annotations is repeated till there are no more remaining CFPs ($\Gamma_{tmp} = \emptyset$). For illustration, Figure 7 depicts the set of conflict-free annotated CFPs as decomposition of $\mathtt{Fin}^{a}_{\Sigma}$ in Figure 5(c) via computation of Algorithm 3. Note that Figures 7(d)–7(f) show the three updated CFPs as result of combining the part before and including cut-off event 12 in the CFP in Figure 6(d) with the part after continuation event 11 in each of the CFPs in Figures 6(a)-6(c), respectively, and then replacing continuation event 11 with event 12 in the corresponding CFPs.

```
function GetAllExecutions
Input: An annotated CFP $U = (\rho, L)$ where $\rho = (B, E, F)$ and $L \subseteq E \times E$
Output: A set of annotated CFPs $\mathbb{U}$
begin
  $\mathbb{U} := \emptyset$;
  $\Gamma := \emptyset$;
  /* compute CFPs from each of the co-sets of leaf conditions */
  CS:= GetLeafCondCoSets($\rho$);
  for $cs \in CS$ do
    $\Gamma \cup := \{$ComputeCFP$(\rho, cs)\}$;
  /* generate annotated CFPs from the above (conflict-free) CFPs */
  $\Gamma_{tmp} := \Gamma$;
  repeat
    Select $\rho' \in \Gamma_{tmp}$;
    $L' := \emptyset$;
    $E_{cutoff} :=$ GetCutoffEvents$(\rho')$;
    $f_{update} :=$ FALSE; /* the flag changes to TRUE if there are CFP updates */
    while $E_{cutoff} \neq \emptyset \land \neg f_{update}$ do
      Select $e_{cut} \in E_{cutoff}$;
      $e_{cont} :=$ GetContinuationEvent$(L, e_{cut})$;
      if $e_{cont} \in \rho' \cdot E$ then
        $L' \cup := \{(e_{cut}, e_{cont})\}$;
      else
        $\Gamma_{add} := GetUpdatedCFPs(\rho', \Gamma, e_{cut}, e_{cont})$; /* see Algorithm 4 */
        $\Gamma \setminus := \{\rho'\}$;
        $\Gamma \cup := \Gamma_{add}$;
        $L_{add} :=$ GetLinks_to$(L, e_{cont}) \setminus \{(e_{cut}, e_{cont})\}$;
        for $e$ where $(e, e_{cont}) \in L'$ do
          $L_{add} \cup := \{(e, e_{cut})\}$
        $L \setminus := \{(e_{cut}, e_{cont})\}$;
        $L \cup := L_{add}$;
        $f_{update} :=$ TRUE; /* set the flag to TRUE upon CFP updates */
        $\Gamma_{tmp} \cup := \Gamma_{add}$; /* add to the remaining CFPs for link annotations */
      $E_{cutoff} \setminus := \{e_{cut}\}$;
    if$\neg f_{update}$ then
      $\mathbb{U} \cup := \{(\rho', L')\}$;
    $\Gamma_{tmp} \setminus := \{\rho'\}$
  until $\Gamma_{tmp} = \emptyset$;
```

Algorithm 3: Transforming an annotated CFP into a set of conflict-free annotated CFPs.

Finally, Algorithm 5 specifies how to convert an annotated CFP into a directed bigraph. The transformation is straight-forward where the events in the CFP become event nodes in the bigraph, conditions become condition nodes, the arcs become the directed edges, and the links are converted to the edges leading from a cut-off event to each of the immediate successors (conditions) of the corresponding continuation event. For illustration, Figure 8 depicts an example of converting an annotated CFP to a directed bigraph.

During preprocessing, we first generate a CFP from a Petri net, and then from the CFP we extract one of more bigraphs. As we only add link information in an annotated CFP, the complexity of the adapted CFP generation algorithm (cf. Algorithm 1) is the same as that of the original CFP algorithm, which is exponential on the number of arcs of the Petri net [7]. The complexity of generating a bigraph from a CFP (cf. Algorithm 2) is linear on the size of the CFP, since the latter is traversed depth-first in reverse order (i.e., starting from a leaf condition).

Now we define the algorithms for determining the 6 basic predicates. First, we introduce two common functions: `RetrieveBigraphs` which returns the set of bigraphs for a process model (r) from the above preprocessing, and `RetrieveAllEvents` which returns the set of event nodes for (i.e., labeled with) a task (t) in a bigraph (G). Each such bigraph represents a possible execution of the corresponding process, and each event node labeled with a task identifier in the bigraph captures an occurrence of the corresponding task in that process execution. For a short notation, an event node labeled with task t is hereafter referred to as an t-event node.

Algorithms 6 and 7 specify how to evaluate the two unary predicates. Predicates *posoccur* or *alwoccur* of task t in process model r can be determined by checking the presence of a t-event node in *any* or *all* bigraphs of r. Based on the fact that

```
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
    $\Gamma' := \emptyset$;
    /* get $\rho$ ready by removing the successor conditions of $e_{cut}$(in $\rho$) */
    $B_{tmp}$:= iSuccessors($\rho, e_{cut}$);
    $\rho \cdot B \setminus := B_{tmp}$;
    $\rho \cdot F \setminus := \{e_{cut}\} \times B_{tmp}$;
    /* retrieve and process the CFPs that contain $e_{cont}$ in $\Gamma$ */
    for $\rho_i \in \Gamma$ where $e_{cont} \in \rho_i \cdot E$ do
        /* remove from $\rho$ the part before $e_{cont}$, $e_{cont}$ itself, and the outgoing edges of $e_{cont}$ */
        $H$ := GetSubCFP_to($\rho_i, e_{cont}$);
        $\rho_i \cdot B \setminus := H \cdot B$;
        $\rho_i \cdot E \setminus := H \cdot E$;
        $\rho_i \cdot F \setminus := H \cdot F \cup (\{e_{cont}\} \times$ iSuccessors($\rho_i, e_{cont}$));
        /* connect the above (updated) $\rho$ and $\rho_i$ to $\rho'$ */
        $\rho' \cdot B := \rho \cdot B \cup \rho_i \cdot B$;
        $\rho' \cdot E := \rho \cdot E \cup \rho_i \cdot E$;
        $\rho' \cdot F := \rho \cdot F \cup \rho_i \cdot F \cup (\{e_{cut}\} \times$ InitialConditions($\rho_i$));
        $\Gamma' \cup := \{\rho'\}$
```

Algorithm 4: Updating a CFP with a cut-off event that links to a continuation event outside the CFP.

```
Function AnnotatedCFP2Bigraph
Input: An annotated CFP $U = (\rho, L)$ 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 $(e_1, e_2) \in L$ do
        $B_1$ := iSuccessors($e_1$);
        $B_2$ := iSuccessors($e_2$);
        $V_{cond} \setminus := B_1$;
        $A \setminus := \{e_1\} \times B_1$;
        $A \cup := \{e_1\} \times B_2$;
```

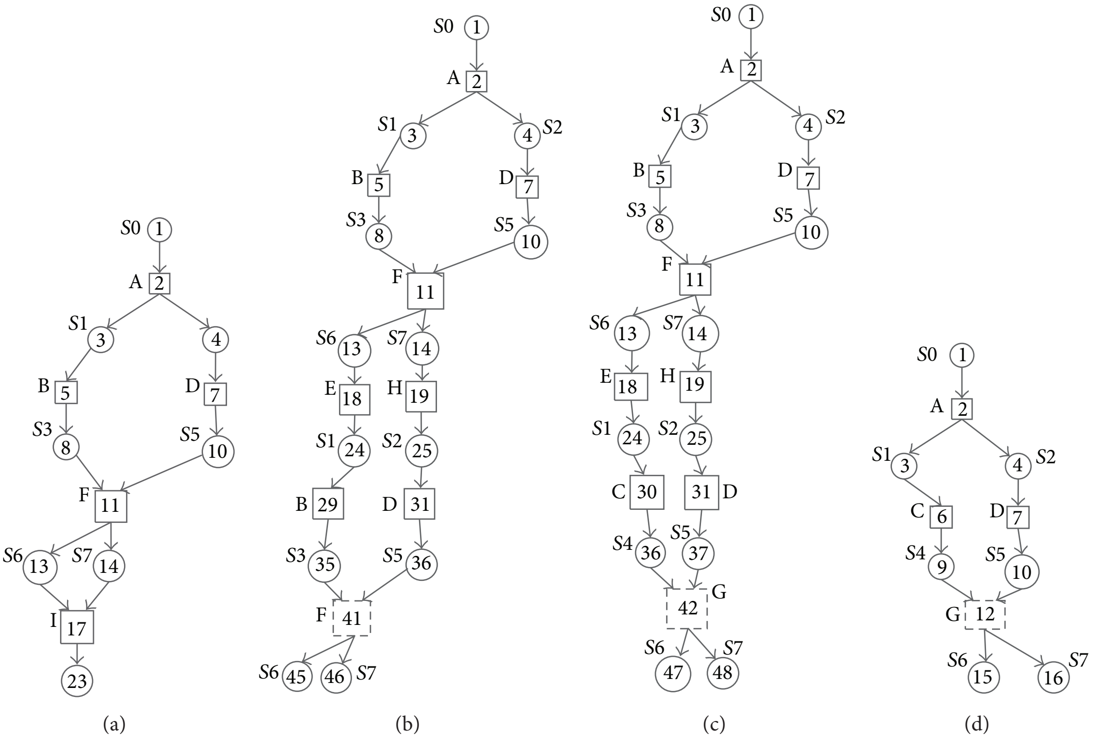
Algorithm 5: Converting an annotated CFP to a directed bigraph.

the set of bigraphs of process model r is each free of choices, the *exclusive* relation between two tasks t and t' is determined by checking in *every* bigraph of r if there are both a t-event node and a t'-event node, as specified in Algorithm 8. In Algorithm 9, the *concur* relation between t and t' in r holds if and only if in each bigraph of r either (1) there are no t- and t'-event nodes at all, or (2) there are both an t-event node and an t'-event node, and no directed path exists between the two nodes.

Next, the remaining algorithms are defined for basic predicates capturing causal relationships between tasks. Evaluation of each such predicate is based on the result of evaluating the corresponding intermediate predicate in individual process executions. Given a process model r, predicate *alwpred* holds only when its intermediate predicate (i.e., Pred) holds in *all* process executions of r, while predicate *pospred* holds as long as its intermediate predicate (i.e., Pred) holds in *one* process execution of r. To capture such semantics, we apply logical operator $\wedge$ (for predicate *alwpred*) or $\vee$ (for predicate *pospred*) between the intermediate predicate over the set of bigraphs ($\mathbb{G}$) of r in the algorithms. Algorithm 10 specifies the evaluation of predicate *alwpred*, and Algorithm 11 specifies the evaluation of *pospred*.

Let us move on to the algorithms for evaluation of intermediate predicates *Pred*. Consider an execution i of process model r and two tasks t_1 and t_2 in r. Algorithm 12 specifies the evaluation of *Pred*. In this algorithm, t_{former} refers to t_1 and t_{latter} to t_2 in the previous discussion, and function *Precedes* (which we will shortly describe in more detail) is used to evaluate causal relationship between two specific task occurrences.

Finally, we introduce the definition of function *Precedes*. In Algorithm 13, function *Precedes* determines if a given e-event node *eventually* precedes a given e'-event node in



(a) (b) (c) (d)

Figure 6: The set of conflict-free CFPs as decomposition of $\texttt{Fin}_{\Sigma}$ in Figure 5(c).

bigraph G (representing a process execution). Following a typical graph search algorithm, it traverses bigraph G from the e-event node (via recursively calling itself) until reaching the e'-event node ($n = m$), the end of the graph ($\texttt{iSuccessors}\ (G, n) = \emptyset$ where $\texttt{iSuccessors}\ (G, n)$ denotes the immediate successors of node n in graph G), or a node that was visited before ($n \in V$ where V stores the set of visited nodes). Also, we consider that the *Precedes* relationship is irreflexive; that is, a task occurrence cannot have a *Precedes* relationship with itself. Hence, when e and e' refer to the same task occurrence ($n = m \wedge V = \emptyset$), *Precedes* returns a negative result.

A basic predicate is evaluated by traversing breadth first each bigraph of each process model in the repository; thus this operation is linear on the size s of a bigraph. Let b be the total number of bigraphs in the repository, and let p be the number of basic predicates in a compliance rule. Hence, the complexity of evaluating a single rule (cf. Algorithms 6, 7, 8, 9, 10, 11, and 12) is linear on p times b times $\max_s$, where $\max_s$ is the size of the largest bigraph in the repository.

It should be noted that for our purposes the adapted CFP generation algorithm and bigraph extraction algorithm are applied to computing the basic predicates over a repository of process models specified as Petri nets. Hence, these operations are performed when inserting a Petri net in the repository. This means that the cost of evaluating a rule is not determined by the complexity of these two algorithms, as the computation of the basic behavioural relations would already have taken place (so essentially we trade space for time).

5. Experiments

In this section, we first describe the implementation of ASBPQL in a software tool, and then we report on the performance of ASBPQL which we measured using this tool.

5.1. Implementation. In order to evaluate the performance of ASBPQL we implemented a tool, namely, ASBPQL Querier, that supports compliance checking for business process models with ASBPQL. A screen shot of ASBPQL Querier is shown in Figure 9. The tool is part of the BeehiveZ toolset v3.0. BeehiveZ is an open-source BPM analysis system based on Java (BeehiveZ can be downloaded from http://code.google.com/p/beehivez/downloads/list).

The architecture of the ASBPQL Querier and of the process model repository with which the ASBPQL Querier interacts inside BeehiveZ is illustrated in Figure 10. The core of the ASBPQL Querier is the query engine: it takes as input the compliance rules produced by users via the query editor and generates as output the results of compliance checking via the query results display. The query editor uses the syntax of ASBPQL. Using this syntax, users can easily specify the semantic relationships in which they are interested. For example, *"A" alwpred "B"* and *"C" concur "F"* mean that the users want to retrieve all process models where in every execution task A precedes task B as well as task C occurs parallel with task F.

Under the hoods, the query engine exploits an internal parser which converts each query statement into a grammar tree. This parser is built by JavaCC (http://javacc.java.net/)

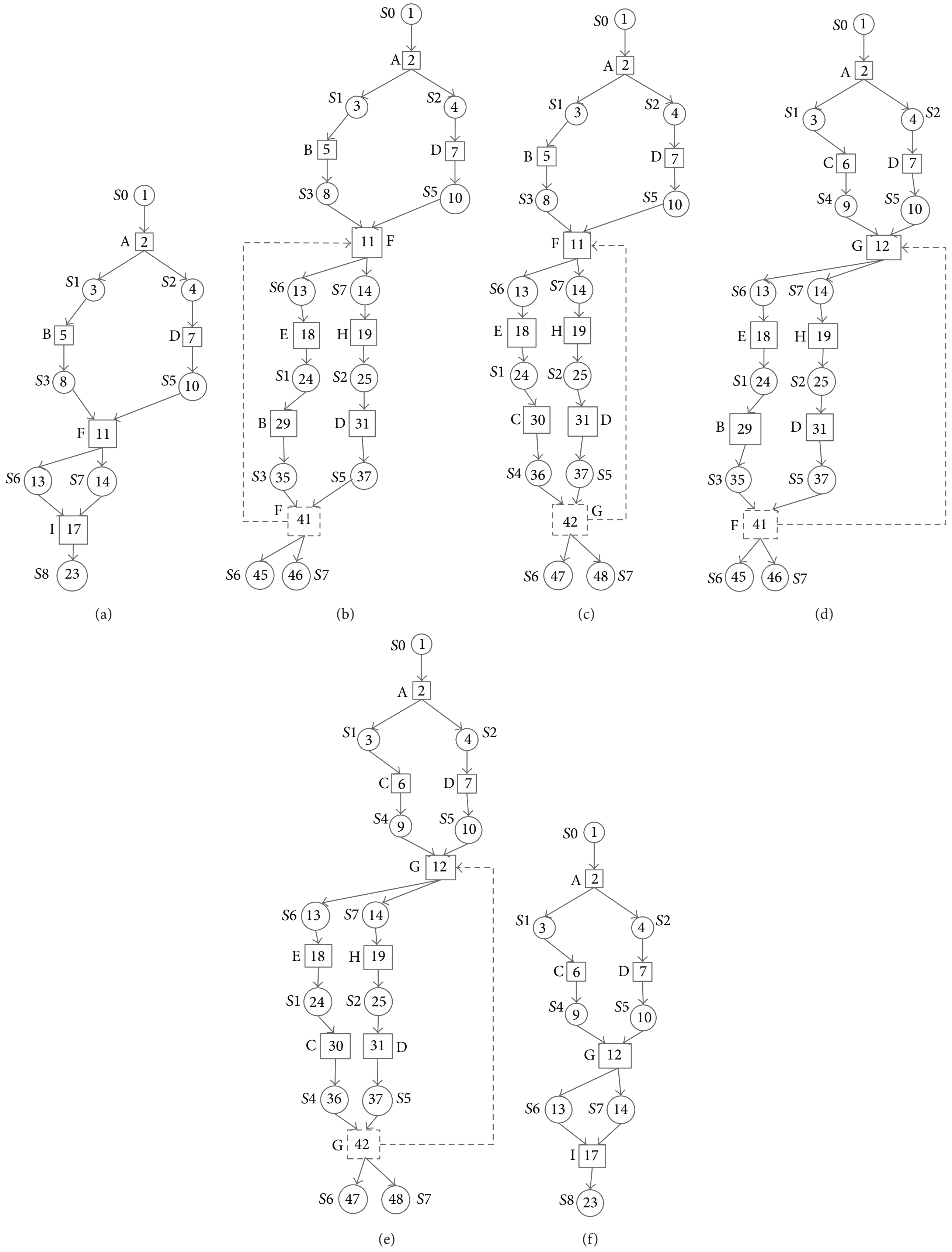


Figure 7: The set of conflict-free annotated CFPs transformed from $\mathtt{Fin}^a_\Sigma$ in Figure 5(c) according to Algorithm 3.

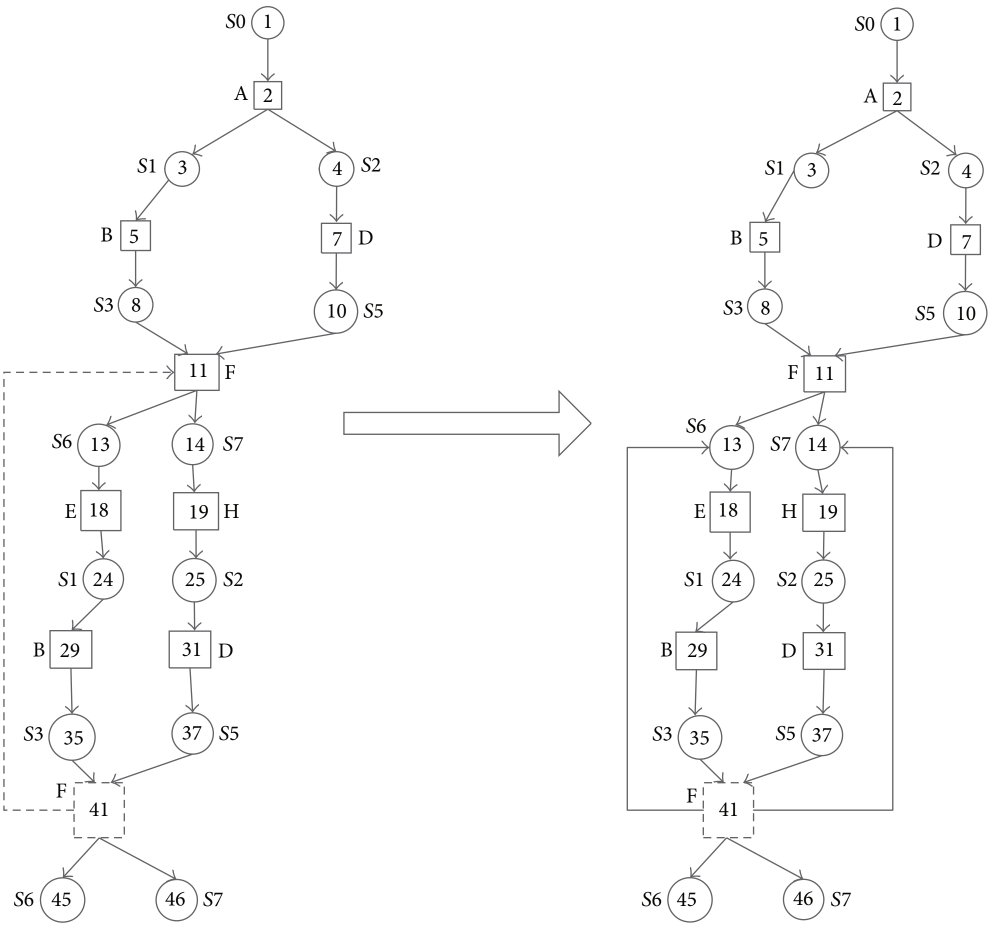


Figure 8: Converting a conflict-free annotated CFP to a directed bigraph.

```
function POSOCCUR
Input: A taskID $t$, a process model $r$
Output: A boolean value
begin
   $\mathbb{G}$ := RetrieveBigraphs($r$);
   return $\bigvee_{G\in\mathbb{G}}$ (RetrieveAllEvents($G, t$) $\neq \emptyset$)
```

Algorithm 6: Determining the (unary) basic predicate *posoccur*.

```
function ALWOCCUR
Input: A taskID $t$, a process model $r$
Output: A boolean value
begin
   $\mathbb{G}$ := RetrieveBigraphs($r$);
   return $\bigwedge_{G\in\mathbb{G}}$ (RetrieveAllEvents($G, t$) $\neq \emptyset$)
```

Algorithm 7: Determining the (unary) basic predicate *alwoccur*.

```
function EXCLUSIVE
Input: Two taskID $t$ and $t'$, a process model $r$
Output: A boolean value
begin
    $\mathbb{G} := \mathsf{RetrieveBigraphs}(r)$;
    return $\bigwedge_{G \in \mathbb{G}} \neg(\mathsf{RetrieveAllEvents}(G, t) \neq \emptyset \wedge \mathsf{RetrieveAllEvents}(G, t') \neq \emptyset)$
```

Algorithm 8: Determining the basic predicate *exclusive*.

```
function CONCUR
Input: Two taskID $t$ and $t'$, a process model $r$
Output: A boolean value
begin
    $\mathbb{G} := \mathsf{RetrieveBigraphs}(r)$;
    return $\bigwedge_{G \in \mathbb{G}} ((\mathsf{RetrieveAllEvents}(G, t) = \emptyset \wedge \mathsf{RetrieveAllEvents}(G, t') = \emptyset) \vee$
        $(\mathsf{RetrieveAllEvents}(G, t) \neq \emptyset \wedge \mathsf{RetrieveAllEvents}(G, t') \neq \emptyset \wedge$
        $\forall_{e \in \mathsf{RetrieveAllEvents}(G,t)} \forall_{e' \in \mathsf{RetrieveAllEvents}(G,t')} [\mathsf{NoDirectedPath}(e, e', G) \wedge \mathsf{NoDirectedPath}(e', e, G)]))$
```

Algorithm 9: Determining the basic predicate *concur*.

```
function ALWPRED
Input: Two taskID $t_{former}$ and $t_{latter}$, a process model $r$
Output: A boolean value
begin
    $\mathbb{G} := \mathsf{RetrieveBigraphs}(r)$;
    return $\bigwedge_{G \in \mathbb{G}} Pred(t_{former}, t_{latter}, G)$
```

Algorithm 10: Determining the basic predicate *alwpred*.

```
function POSPRED
Input: Two taskID $t_{former}$ and $t_{latter}$, a process model $r$
Output: A boolean value
begin
    $\mathbb{G} := \mathsf{RetrieveBigraphs}(r)$;
    return $\bigvee_{G \in \mathbb{G}} Pred(t_{former}, t_{latter}, G)$
```

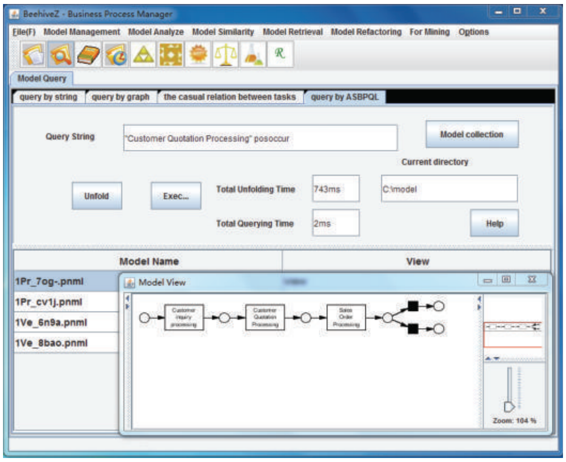
Algorithm 11: Determining the basic predicate *pospred*.

```
function PRED
Input: Two taskID $t_{former}$ and $t_{latter}$, a bigraph $G$
Output: A boolean value
begin
    $W := \mathsf{RetrieveAllEvents}(G, t_{former})$;
    $X := \mathsf{RetrieveAllEvents}(G, t_{latter})$;
    return $\exists_{e \in W} \exists_{e' \in X} Precedes(G, e, e', \emptyset)$
```

Algorithm 12: Determining the intermediate predicate Pred.

```
Function Precedes
Input: A bigraph $G$, a node $m$, a event node $n$, a set of nodes $V$ (the set of visited nodes)
Output: A boolean value
Begin
  if $n = m \wedge V = \emptyset$ then
    return FALSE;
  else
    if $n = m \wedge V \neq \emptyset$ then
      return TRUE;
    else
      if $n \in V \vee \mathrm{iSuccessors}(G, n) = \emptyset$ then
        return FALSE;
      else
        return $\bigvee_{s \in \mathrm{iSuccessors}(G,n)} Precedes(G, s, m, V \cup \{n\})$
```

Algorithm 13: Determining the Precedes relationship in the bigraph of a conflict-free annotated CFP.



Query String: "Customer Quotation Processing" posoccur

Current directory: C:\model

Total Unfolding Time: 743ms

Total Querying Time: 2ms

1Pr_7og-.pnml

1Pr_cv1j.pnml

1Ve_6n9a.pnml

1Ve_8bao.pnml

Figure 9: A screenshot of ASBPQL Querier.

which is a widely used open source parser generator and lexical analyzer generator for Java. Grammar trees are then used by the evaluator to identify all process models in the repository that satisfy the requirements of a given query. To do so, the evaluator needs to get access to the collection of process models stored in the process model repository in Petri net format, as well as the directed bigraphs which have been constructed from the annotated CFPs of each Petri net by the annotated CFP decomposer using Algorithm 2. The generation of annotated CFPs is performed by the annotated CFP generator using Algorithm 1. For an annotated CFP, the data structure of conditions, events, and directed arcs are represented by nodes of doubly linked lists which support in particular fast insertion of nodes and backward traversing.

Moreover, for efficiency reasons, we keep an inverted index for every node label that appears in the set of annotated CFPs. We use Apache Lucene to manage these indexes (http://lucene.apache.org/). Specifically, for each label we record all processes which contain that label in some nodes. Based on this index, after a compliance rule is issued the tool can instantly filter out a set of candidate models containing the labels used in the compliance rule. The rest of the models are thus ignored since they are not relevant to the current rule. This step typically reduces the scope of searching and increases the tool's performances. Furthermore, an advantage of using inverted indexes is that they can be easily updated as a result of changing a node label in the repository. For more details on this index, we refer to previous work [18].

5.2. Performance Measurements. We prepared a set of eight sample rules using various ASBPQL basic predicates and measured the evaluation of each of these rules over three process model collections. The first two collections are real-life repositories: the SAP R/3 reference model, consisting of 604 EPC models, and the IBM BIT library, consisting of 1,128 Petri nets. The SAP dataset is used by SAP consultants to deploy the SAP enterprise resource planning system within organizations [19]. The IBM BIT library includes five collections (A, B1, B2, C1, and C2) of process models from various domains, including insurance and banking [20]. The third dataset contains 10,000 artificially-generated models. (This dataset is available at http://code.google.com/p/beehivez/downloads/list.)

Since the SAP dataset is represented in the EPC notation, we first transformed these models into Petri nets using ProM (http://www.processmining.org/). This resulted in 591 Petri nets for the SAP dataset (13 SAP reference models could not be mapped into Petri nets through ProM). In the resulting dataset there are 4,439 transitions out of which 1,494 are uniquely labeled (33% of the total), while in the IBM dataset there are 9,083 transitions with 946 uniquely labeled one (10% of the total). The structural characteristics of the three datasets used in the experiments are reported in Table 2. In particular, we can see that the SAP and IBM collections have models of comparable sizes based on the average number of their elements (transitions, places, and arcs).

We generated the third dataset using BeehiveZ based on the reduction rules from [8]. The number of nodes per model follows a normal distribution. Specifically, the number of transitions per model ranges from 1 to 50 (average 24.85), the number of places from 1 to 47 (average 16.81), and the number of arcs from 2 to 162 (average 63.22). The labels of transitions

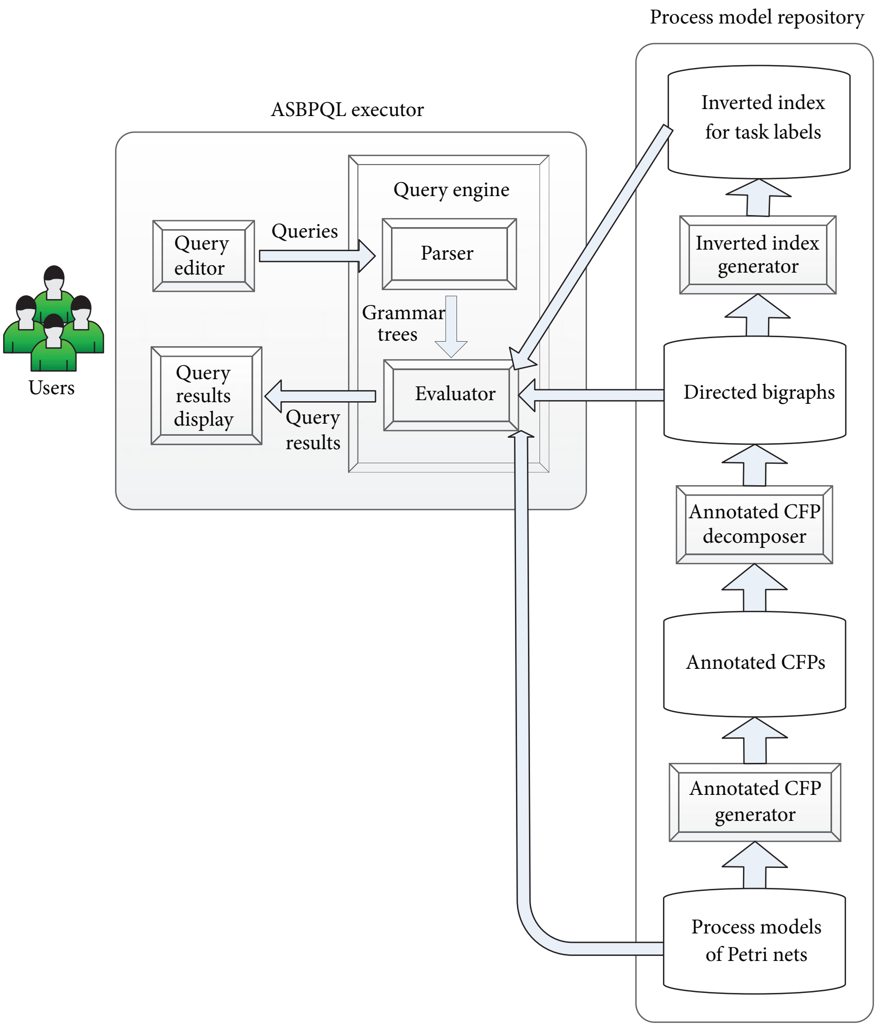


Figure 10: BeehiveZ: architecture of ASBPQL Querier and Process Models Repository.

Table 2: Structural characteristics of the three datasets.

Dataset	Models	Transitions	Unique transitions	Avg. transitions	Avg. places	Avg. arcs
SAP	591	4,439	1,494 (33%)	7.5	12.7	19.7
IBM	1,128	9,083	946 (10%)	8.06	10.97	21.47
AG	10,000	248,493	62 (0.026%)	24.85	16.81	63.22

AG: artificially generated dataset.

were randomly chosen from a fixed label set comprising the characters "A–Z" and "a–z" and the numbers "0–9", each label being made by a single character or number. In total, this led to 248,493 transitions in this dataset, with 62 unique labels (corresponding to 0.026% of the total number of transitions). As we mentioned earlier that we deployed inverted labels for each task label, we chose such a very low set of unique labels compared to the total number of transitions in order to increase the number of models that can potentially satisfy a rule; thus we can get precise measurement result about the efficiency of executing a compliance rule. All models used in the experiments are bounded Petri net, which is a requirement for unfolding according to [21].

We conducted our tests on an Intel Core i7-2600 @3.4 GHz and 8 GB RAM, running Windows 7 ultimate and JDK6. The heap memory for the JVM was set to 1 GB. We executed each compliance rule twelve times and measured each response time. We then discarded the highest and lowest response times for each rule and computed the average response time over the remaining ten values. The test rules and the response times for the three datasets are reported in Table 3.

Table 3: Response times to execute eight sample compliance rules over the three datasets.

	Queries	Candidate models			Returned models			Response time [ms]		
		SAP	IBM	AG	SAP	IBM	AG	SAP	IBM	AG
r_1	$[x_1]$ posoccur	5	2	3674	5	2	3674	2.9	53	85
r_2	$[x_1]$ alwoccur	4	2	3307	1	1	286	7.6	131	374
r_3	$[x_1]$ posoccur and $[x_2]$ alwoccur	6	1	1646	2	1	143	11.4	193	411
r_4	$[x_1]$ concur $[x_2]$	1	1	603	1	0	22	17.5	318	1392
r_5	$[x_1]$ exclusive $[x_2]$	1	2	549	1	0	18	18.6	360	1451
r_6	$[x_1]$ pospred $[x_2]$	1	1	552	1	1	72	12.8	294	633
r_7	$[x_1]$ alwpred $[x_2]$	3	1	540	1	1	46	14.7	253	804
r_8	$[x_1]$ pospred $[x_2]$ or $[x_2]$ alwpred $[x_3]$	4	1	593	2	0	33	12.5	187	638

In particular, r_1 to r_3 are used to test the unary basic predicates *posoccur* and *alwoccur*, and r_4 and r_5 are for the *concur* and *exclusive* predicates, while r_6 to r_8 are for *causal relation* predicates. For readability, in the table we use‘fictitious labels for transitions (e.g., x_1). The real labels from the three datasets, can be found in the Appendix.

The second and third columns of Table 3 show for each rule the number of models being filtered by BeehiveZ's inverted index ("candidate models") and the number of models that actually satisfy the rule ("returned models"). These numbers are very low for the SAP and IBM datasets (e.g., r_3 yields six models in the SAP dataset, out of which only two satisfies the rule), due to the high number of unique labels within these collections (see Table 2). However, as expected, these numbers grow significantly in the artificially generated collection (as an example, r_6 yields 552 models of which 72 satisfy the rule).

The last column of Table 3 shows the response times to execute the sample queries. These times are in the order of milliseconds for the SAP and IBM datasets (average 15 ms and 254.7 ms) and less than one second for the artificial dataset (average 850.4 ms). This shows that the technique is highly scalable to very large datasets. Having said that our technique shifts computation time from compliance checking to model insertion. In other words, most of the time is employed in generating the CFPs rather than in executing the compliance checking. Specifically, the overall time for building the set of CFPs and the corresponding bigraphs for the three datasets is 12.6 mins (SAP dataset), 28.5 mins (IBM), and 8.1 hours (artificial dataset). However, since we build annotated CFPs incrementally as we insert each Petri net into the repository, in practice the time for creating a single CFP is very short: only 1.28 s on average for a model from the SAP dataset, 1.52 s for a model from the IBM dataset, and 2.92 s for a model from the artificial dataset. These times are reasonable since repository users typically insert or remove single process models, or small groups thereof, at once, rather than inserting or removing entire model collections at once.

As expected, the storage size of the CFPs (including the label indexes) and corresponding bigraphs can be large. While it is only 26.8 MB for the SAP dataset and 18.1 MB for the IBM dataset, this value gets to 3.38 GB for the artificial dataset. However, this space is still acceptable considering that in organizational settings dedicated servers are typically employed to host process model repositories, rather than single desktop machines.

6. Related Work

Based on the importance of query languages for business process models, in 2004, the Business Process Management Initiative (BPMI) planned to define a standard process model query language. While such a standard has never been published, two major research efforts have been dedicated to the development of query languages for process models. One is known as BP-QL [1], a graphical query language based on an abstract representation of BPEL and supported by a formal model of graph grammars for processing of queries. BP-QL can be used to query 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 effort, namely, BPMN-Q [2, 3], is also a visual query language which extends a subset of the BPMN modelling notation and supports graph-based query processing. Similar to BP-QL, BPMN-Q only captures the structural (i.e., syntactical) relationships between tasks. BPMN-Q uses a directed path (enhanced by operators like ≪*leads to*≫ and ≪*precedes*≫) connecting two activities to capture the requirement that they occur in order. The processing of BPMN-Q queries includes several steps. In short, BPMN-Q query engine searches for the process models that contain subgraphs that structurally match a query, reduces these subgraphs (remove elements that are not relevant to the query), translates the reduced subgraphs into Petri nets, and then calculates the corresponding reachability graph for each Petri net. Next, the query is translated into temporal logic formula which is fed into a model checker together with the reachability graphs generated from Petri nets. Finally, the model checker would output the process models that satisfy the query. Although part of the evaluation of BPMN-Q queries is based on LTL formulae, one of the most important step is subgraph matching which is totally structure based. For example, for the BPMN-Q query in Figure 2, the subgraphs obtained from the process model (c) in Figure 1 is shown in Figure 11. If only consider this subgraph, this

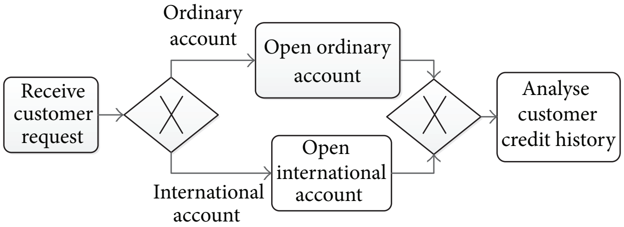


Figure 11: The resulting subgraph of the process model in Figure 1(c) after executing the query in Figure 2.

BPMN-Q query holds, but this is not the case for the process execution where task "open VIP account" occurs. Accordingly, as discussed in Section 1, the main problem of BPMN-Q is that it cannot answer the question whether for the resulting processes the requirements of a query hold in every process execution or in just some process executions. BPMN-Q only returns process models where requirements hold in some process executions, rather than in every process execution. A comparison between ASBPQL and BPMN-Q is shown in Figure 12 where empty cells mean that the corresponding requirements cannot be captured by BPMN-Q. In [22], 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 process model repositories using BPMN-Q and its extensions is presented in [23]. In [24], the authors proposed an indexing mechanism to improve the efficiency of evaluating BPMN-Q queries.

ASBPQL provides three distinguishing features compared to the previous languages. First, its abstract syntax and semantics have been purposefully defined to be independent of a specific process modelling language (such as BPEL or BPMN). This allows ASBPQL and its query evaluation technique to be implemented for a variety of process modelling languages. Second, ASBPQL can express various temporal-ordering relations (precedence/succession, concurrence, and exclusivity) between individual tasks, between an individual task and a set of tasks, and between different sets of tasks (in some or every process execution). Third, these rich querying constructs are evaluated over the execution semantics of process models, rather than their structural relationships. In fact, structural characteristics alone are not able to capture all possible order relations among tasks which can occur during execution, in particular with respect to cycles and task occurrences (recall the discussions in Section 1).

In earlier work [25], we provided an initial attempt at defining a query language based on execution semantics of process models. The language was written in linear temporal logic (LTL) and only supported precedence/succession relations among individual tasks (not sets of tasks). Queries in this language are evaluated based directly on annotated CFPs (i.e., TPCFPs in [25]), rather than on the directed bigraphs which are built from decomposing the annotated CFPs (a directed bigraphs represents an execution of a process). As a result, this language only returns the process models which satisfy the requirements just in some process executions, rather than in every execution. In addition, using LTL formulae as queries is not very user friendly for ordinary users. In [26], the authors proposed an business query language (BQL) to capture 4 types of relations (*Exist*, *ParallelWith*, *Exclude*, and *Precede*). A query in BQL returns processes of which some executions satisfy these four types of relations. Furthermore, BQL suffers from a drawback that the formal semantics of it has not been defined.

In addition to the development of a specific process model query language, other techniques are available in the literature which can be useful for querying process model repositories. In [27, 28] the authors focus on querying the content of business process models based on metadata search. In [29], an XML-based process query language, IPM-PQM, was designated to express search requirements. IPM-PQM can express four types of search conditions: Process-Has-Attribute, Process-Has-Activity, Process-Has-Subprocess, and Process-Has-Transition. IPM-PQM is a typical structure-based process querying technology. VisTrails system [30] allows users to query scientific workflows by example and to refine workflows by analogies. WISE [31] is a workflow information search engine which supports keyword search on workflow hierarchies. In [32] 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 [33–36], a group of similarity-based techniques has been proposed which can be used to support process querying. In previous work, we designed a technique to query process model repositories based on an input Petri net [18]. In [37], the authors introduced an execution-log-based query language which enables users to find elements and their relationships in process logs. In [38, 39], an approach that supports "static" and "dynamic" querying of process has been presented. As for the static querying, this approach searches for matching processes which contains specified context elements, such as business function, roles, and resources. This is based on keyword matching. As for the dynamic querying, similar to BPMN-Q, it tries to find process models where the requirements hold in just some process instances. In [40], the authors proposed an approach to searching business process models. This approach induces relationships between activities from their labels; it provides an approximate process model search mechanism. Finally, in [41], the notion of behavioural profile of a process model is defined, which captures dedicated behavioural relations like exclusiveness or potential occurrence of activities. These behavioural relations are derived from the structure of the unfolding of a process model. However, the main foundation of beavioural profile is the weak order (two transitions t_1, t_2 are in weak order, if there *exists* an process execution in which t_1 occurs before t_2). Thus, for the reasons mentioned above, behavioural profile only provides an approximation of a process model's behavior which just holds in some process executions, whereas we can precisely determine whether a process model satisfies or not a given query in every process execution. Moreover, the efficient computation of this approach requires process models to be sound free-choice Petri nets, whereas our query

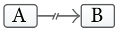
Requirements	ASBPQL	BPMN-Q
Select all process where in some process executions task A precedes task B	A pospred B	
Select all process where in every process executions task A precedes task B	A alwpred B	
Select all process where in every process execution task A occurs in parallel with task B	A concur B	
Select all process where in every process execution task A occurs exclusively with task B	A exclusive B	
Select all process where in some process executions task A occurs	A posoccur	
Select all process where in every process execution task A occurs	A alwoccur	

Figure 12: A comparison between ASBPQL and BPMN-Q.

Table 4: Mapping of the task labels used in Table 3 with those in the SAP, IBM and AG datasets.

Queries	Labels	SAP	IBM	AG
r_1	x_1	Customer quotation processing	process.s0000009##s000001827.outputCriterion.s00000859	Z
r_2	x_1	Goods receipt	process.s00000275##s00002184.outputCriterion.s00000838	K
r_3	x_1,	Customer quotation processing,	process.s00000285##s00002171.outputCriterion.s00000743,	R,
	x_2	Sales order processing	process.r00000266##n00002468.outputCriterion.s00000773	S
r_4	x_1,	Subsequent acquisition	process.s00000247##s00002258.outputCriterion.s00000772,	A,
	x_2	Processing of asset acquisition	process.s00000266##s00002468.outputCriterion.s00000773	H
r_5	x_1,	Product structure management for variant products,	process.s00000265##s00002061.outputCriterion.s00000774,	M,
	x_2	Product structure management via CAD	process.r00000266##s00009387.outputCriterion.s00000721	R
r_6	x_1,	Subsequent acquisition,	process.s00000247##s00002258.outputCriterion.s00000772,	M,
	x_2	Processing of asset acquisition	process.s00000266##s00002468.outputCriterion.s00000773	N
r_7	x_1,	Customer quotation processing,	process.s00000285##s00002171.outputCriterion.s00000743,	X,
	x_2	Sales order processing	process.r00000266##n00002468.outputCriterion.s00000773	Y
r_8	x_1,	Settlement account assignment,	process.s00000243##s00002261.outputCriterion.s00005267,	J,
	x_2,	Periodic settlement,	process.r00000106##n00002617.outputCriterion.s00000704,	W,
	x_3	Contact release order	process.r00000231##n00006324.outputCriterion.s00001894	t

evaluation technique only requires Petri nets to be bounded, in order to unfold them.

7. Conclusions

In this paper, we simplify SPS by logic reasoning to define a concise and expressive retrieval language to specify semantics-based compliance rules. And we contribute an efficient technology based on unfolding to explore the semantics of process models. In such a technology, we can extract every independent execution from business process models without suffering from well-known state explosion. The language and its evaluation have been implemented as a component of the process analysis tool BeehiveZ. We also conduct experiments over three large datasets to evaluate the efficiency of our technology. Indeed, the performance measurements show that the technique can efficiently cope with very large datasets (the artificial collection counted 10,000 process models).

In the future, we will introduce graphical interface for querying in order to make BeehiveZ more intuitionistic.

Appendix

In Table 4, we provide the mapping between the fictitious labels used in Table 3 and the real labels used in the SAP, IBM, and artificial datasets.

Acknowledgments

Song, Wang, and Wen are supported by the National Basic Research Program of China (2009CB320700), the National High-Tech Development Program of China (2012AA040904), the Project of National Natural Science Foundation of China (90718010,61003099), the Program for New Century Excellent Talents in University of China, and the Ministry of Education and China Mobile Research Foundation (MCM20123011).

References

[1] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo, “Querying business processes with BP-QL,” *Information Systems*, vol. 33, no. 6, pp. 477–507, 2008.

[2] A. Awad, “BPMN-Q: a language to query business processes,” in *Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA ’07)*, M. Reichert, S. Strecker, and K. Turowski, Eds., pp. 115–128, St.Goar, Germany, October 2007.

[3] A. Awad, G. Decker, and M. Weske, “Efficient compliance checking using BPMN-Q and temporal logic,” in *Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2–4, 2008. Proceedings*, M. Dumas, M. Reichert, and M. Shan, Eds., Lecture Notes in Computer Science, pp. 326–341, Springer, Berlin, Germany.

[4] OMG, *Business Process Model and Notation (BPMN) ver. 2.0*, January 2011, http://www.omg.org/spec/BPMN/2.0.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Property specification patterns for finite-state verification,” in *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP ’98)*, pp. 7–15, ACM, March 1998.

[6] K. L. McMillan, “A technique of state space search based on unfolding,” *Formal Methods in System Design*, vol. 6, no. 1, pp. 45–65, 1995.

[7] J. Esparza, S. Römer, and W. Vogler, “An improvement of McMillan’s unfolding algorithm,” *Formal Methods in System Design*, vol. 20, no. 3, pp. 285–310, 2002.

[8] T. Murata, “Petri nets: properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[9] M. Nielsen, G. D. Plotkin, and G. Winskel, “Petri nets, event structures and domains,” in *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2–4, 1979*, G. Kahn, Ed., Lecture Notes in Computer Science, pp. 266–284, Springer, London, UK, 1979.

[10] J. Engelfriet, “Branching processes of Petri nets,” *Acta Informatica*, vol. 28, no. 6, pp. 575–591, 1991.

[11] W. M. P. van der Aalst, “The application of Petri nets to workflow management,” *Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[12] W. M. P. van der Aalst, “Petri-net-based workflow management software,” in *Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems*, A. Sheth, Ed., pp. 114–118, Citeseer, 1996.

[13] R. M. Dijkman, M. Dumas, and C. Ouyang, “Semantics and analysis of business process models in BPMN,” *Information and Software Technology*, vol. 50, no. 12, pp. 1281–1294, 2008.

[14] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede, “Formal semantics and analysis of control flow in WS-BPEL,” *Science of Computer Programming. Methods of Software Design: Techniques and Applications*, vol. 67, no. 2-3, pp. 162–198, 2007.

[15] N. Lohmann, “A feature-complete petri net semantics for ws-bpel 2.0,” in *Proceedings of the 4th International Workshop on Web Services and Formal Methods (WS-FM ’07)*, Brisbane, Australia, September 2007.

[16] W. M. P. van der Aalst, “Formalization and verification of event-driven process chains,” *Information and Software Technology*, vol. 41, no. 10, pp. 639–650, 1999.

[17] 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, Held in Dagstuhl, Germany, September 1996*, W. Reisig and G. Rozenberg, Eds., pp. 429–528, Springer, 1996.

[18] T. Jin, J. Wang, N. Wu, M. La Rosa, and A. H. M. ter Hofstede, “Efficient and accurate retrieval of business process models through indexing (short paper),” in *On the Move to Meaningful Internet Systems: OTM 2010-Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25–29, Proceedings, Part I*, R. Meersman, T. S. Dillon, and P. Herrero, Eds., vol. 6426 of *Lecture Notes in Computer Science*, pp. 402–409, Springer, 2010.

[19] G. Keller and T. Teufel, *SAP R/3 Process Oriented Implementation*, Addison-Wesley Longman, Boston, Mass, USA, 1998.

[20] D. Fahland, C. Favre, B. Jobstmann et al., “Instantaneous soundness checking of industrial business process models,” in *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8–10, 2009. Proceedings*, vol. 5701 of *Lecture Notes in Computer Science*, Springer, 2009.

[21] J. Esparza, S. Römer, and W. Vogler, “An improvement of McMillan’s unfolding algorithm,” in *Tools and Algorithms For Construction and Analysis of Systems, Second International Workshop, TACAS ’96, Passau, Germany, March 27–29, 1996, Proceedings*, T. Margaria and B. Steffen, Eds., vol. 1055 of *Lecture Notes in Computer Science*, pp. 87–106, Springer, London, UK, 1996.

[22] A. Awad, A. Polyvyanyy, and M. Weske, “Semantic querying of business process models,” in *Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference (EDOC ’08)*, pp. 85–94, IEEE Computer Society, Munich, Germany, September 2008.

[23] S. Sakr and A. Awad, “A framework for querying graph-based business process models,” in *Proceedings of the 19th International World Wide Web Conference (WWW ’10)*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds., pp. 1297–1300, ACM, Raleigh, NC, USA, April 2010.

[24] A. Awad and S. Sakr, “On efficient processing of BPMN-Q queries,” *Computers in Industry*, vol. 63, no. 9, pp. 867–881, 2012.

[25] L. Song, J. Wang, L. Wen, W. Wang, S. Tan, and H. Kong, “Querying process models based on the temporal relations between tasks,” in *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference (EDOCW ’11)*, pp. 213–222, IEEE Computer Society, Helsinki, Finland, September 2011.

[26] T. Jin, J. Wang, and L. Wen, “Querying business process models based on semantics,” in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications (DASFAA ’11), Part II*, Hong Kong, April 2011.

[27] J. Vanhatalo, J. Koehler, and F. Leymann, “Repository for business processes and arbitrary associated metadata,” in *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5–7, 2006, Proceedings*, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102 of *Lecture Notes in Computer Science*, pp. 426–431, Springer, Vienna, Austria, 2006.

[28] A. Wasser, M. Lincoln, and R. Karni, “ProcessGene Query—a tool for querying the content layer of business process models,” in *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5–7, 2006, Proceedings*, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102 of *Lecture*

Notes in Computer Science, pp. 1–8, Springer, Vienna, Austria, 2006.

[29] I. Choi, H. Jeong, M. Song, and Y. U. Ryu, "IPM-EPDL: an XML-based executable process definition language," *Computers in Industry*, vol. 56, no. 1, pp. 85–104, 2005.

[30] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva, "Querying and Re-using workflows with VIsTralls," in *Proceedings of the International Conference on Management of Data (SIGMOD '08)*, J. T. Wang, Ed., pp. 1251–1254, Vancouver, BC, Canada, June 2008.

[31] Q. Shao, P. Sun, and Y. Chen, "WISE: a workflow information search engine," in *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE '09)*, pp. 1491–1494, IEEE, Shanghai, China, April 2009.

[32] R. Lu and S. W. Sadiq, "Managing process variants as an information resource," in *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, Eptember 5–7, 2006, Proceedings*, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102 of *Lecture Notes in Computer Science*, pp. 426–431, Springer, Vienna, Austria, 2006.

[33] M. Momotko and K. Subieta, "Process query language: a way to make workflow processes more flexible," in *Proceedings of the 8th East European Conference on Advances in Databases and Information Systems (ADBIS '04)*, pp. 306–321, Budapest, Hungary, September 2004.

[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," in *Proceedings of the 4th International Conference on Business Process Management (BPM '06)*, Vienna, Austria, September 2006.

[35] M. Ehrig, A. Koschmider, and A. Oberweis, "Measuring similarity between semantic business process models," in *Conceptual Modelling 2007, Proceedings of the Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM 2007), Ballarat, Victoria, Australia, January 30-February 2, 2007, Proceedings*, J. F. Roddick and A. Hinze, Eds., CRPIT, pp. 71–80, Australian Computer Society, Darlinghurst, Australia, 2007.

[36] B. Dongen, R. Dijkman, and J. Mendling, "Measuring similarity between business process models," in *Advanced Information Systems Engineering, 20th International Conference, CAiSE, Montpellier, France, June 16-20, 2008, Proceedings*, Z. Bellahsene and M. Léonard, Eds., vol. 5074 of *Lecture Notes in Computer Science*, pp. 450–464, Springer, Berlin, Germany, 2008.

[37] S. Beheshti, B. Benatallah, H. R. M. Nezhad, and S. Sakr, "A query language for analyzing business processes execution," in *Proceedings of the 9th International Conference Business Process Management (BPM '11)*, Clermont-Ferrand, France, September 2011.

[38] I. Markovic, A. C. Pereira, D. de Francisco Marcos, and H. Muñoz, "Querying in business process modeling," in *Proceedings of the International Conference on Service Oriented Computing (ICSOC '07)*, Vienna, Austria, September 2007.

[39] I. Markovic, A. C. Pereira, and N. Stojanovic, "A framework for querying in business process modelling," in *Multikonferenz Wirtschaftsinformatik (MKWI '08)*, München, Germany, February 2008.

[40] M. Lincoln and A. Gal, "Searching business process repositories using operational similarity," in *On the Move to Meaningful Internet Systems: OTM 2011—Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Part I*, Hersonissos, Crete, Greece, October 2011.

[41] M. Weidlich, J. Mendling, and M. Weske, "Efficient consistency measurement based on behavioral profiles of process models," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 410–429, 2011.