VIEW: An Incremental Approach to Verify Evolving Workflows

Mohammad Javad Amiri and Divyakant Agrawal
University of California Santa Barbara
Santa Barbara, California
{amiri,agrawal}@cs.ucsb.edu

Abstract

Business processes (workflows) are typically the compositions of services (activities and tasks) and play a key role in every enterprise. Business processes need to be changed to react quickly and adequately to internal and external events. Moreover, each business process is required to satisfy certain desirable properties such as soundness, consistency, or some user-defined linear temporal logic (LTL) constraints. This paper focuses on the verification of evolving processes: given a business process, a change operation, and a set of LTL constraints, check whether all execution sequences of the evolved process satisfy all the given constraints. We propose a technique to incrementally check and verify the constraints of evolving business processes. Furthermore, we develop VIEW, a framework to model, change, and verify Evolving Workflows and conduct a study to evaluate the effect of workflow characteristics on the performance of verification approaches. Experiments reveal several interesting factors concerning performance and scalability.

CCS Concepts

- Information systems → Enterprise information systems;
- Applied computing → Business process management;
- Software and its engineering → Formal software verification;

Keywords

Incremental Verification, Business Processes, DecSerFlow, Evolving Business Process

ACM Reference Format:

1 Introduction

A business process consists of a set of activities performed in coordination in an organizational environment to accomplish a business goal. Business process management (BPM) includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes [32]. An important problem in BPM is to determine whether a process model exhibits certain desirable behaviors, known as process verification [9]. Although the problem has been extensively studied, introducing new modeling languages and rapidly changing environment still make verification a challenging problem. On one hand, while earliest verification approaches have mainly focused on control flow correctness, new paradigms such as artifact-centric and decision-aware process modeling deal with data aspect of business processes. On the other hand, business processes need to be changed to react quickly and adequately to internal and external events and adapt the business processes with current and upcoming requirements [7, 30]. Since every change in a business process could affect the correctness, the verification needs to be performed after each change. However, known algorithms especially in the presence of data and objects are highly intractable and thus not practical [19]. A goal is to help BPM applications by (1) developing effective techniques and algorithms to verify evolving processes, and (2) conducting a performance study on verification algorithms based on the characteristics of business processes. This paper provides the first step to realize this goal.

This paper focuses on processes with stateful objects and studies linear temporal logic (LTL) constraints on these processes. Note that while data is not modeled, “finite domain” data can be captured using object states. We study five different classes of LTL constraints which are known as DecSerFlow constraints [28]: cardinality, existence, ordering, alternating, and chain where the first class considers the number of occurrences of one activity within all execution of a process and the four other classes take binary constraints between two activities into account. In this paper, we study the following verification problem: do all possible executions of a process satisfy the aforementioned LTL constraints?

A straightforward approach to LTL constraint checking is to construct automata representing individual constraints and determine if their cross product accepts all the execution sequences of the process schema. This approach has one major problem: as soon as the process schema changes, all the execution sequences of the updated schema need to be checked again. Another possibility is to perform checking incrementally: When a process $P$ is changed into $P'$, constraints of $P'$ is checked based on the auxiliary stored data concerning constraints checking of $P$ and the specific change of $P$. In that way, we only check part of the process schema that is affected by the change.

To support incremental checking, additional information about processes and object state transitions is stored in “auxiliary data stores”, which are associated with nodes in a tree representation of the process in question. At design time, we consider well-structured processes and a set of LTL constraints as the input to the framework. Then a process tree is created from the process model and depending on the given constraints, auxiliary data stores are constructed for the nodes of the tree. Auxiliary data stores can be constructed for nodes of the tree in the bottom-up fashion from the leaf nodes. We first construct auxiliary data for activity nodes and then construct the
auxiliary data store for each internal node using that of its child(ren) depending on the type of the internal node. Finally, the verification can be done by checking the auxiliary data of the root node.

At evolution time, one of the process change operations including updating an activity, expanding a activity to a sequential, conditional, parallel, or loop block, and shrinking a sequential, conditional, parallel, or loop block into an activity is considered as the input. When a process is modified, the leaf level (activity) nodes and their auxiliary data are modified, and the changes are propagated incrementally along the path to the root node. Finally, the constraints can be checked using only the auxiliary data of the root node. Note that, while only the basic change operations are considered, any advanced change operation, e.g., swap two fragments, can easily be presented with a sequence of these basic change operations.

A key contribution of this paper is to show that it is possible to employ incremental methods for object-aware process verification. Technically, this paper makes the following contributions: (1) Techniques of incremental maintenance of auxiliary data store are developed, (2) VIEW, a framework to model, evolve, and (incrementally) verify business processes is developed, and (3) Performance of different verification algorithms, i.e., Spin-based, bottom-up construction, and incremental, for different processes is studied.

The rest of the paper is organized as follows. Section 2 defines the formal process model, DecSerFlow constraints, and schema change operations. Section 3 introduces process trees, formulates auxiliary data stores, and proposes our bottom-up approach to construct the auxiliary data stores. Section 4 presents our incremental approach to construct the auxiliary data stores and the verification step, and Section 5 performs a study on the performance of verification approaches. Section 6 discusses related work, and Section 7 concludes the paper.

2 Business Processes: Model, Constraints, and Evolution

In this section, we first present a model for business processes. Then a set of process schema change operations is provided to evolve processes. Next, different classes of DecSerFlow constraints are introduced, and finally, a SPIN-based algorithm to verify business processes is presented.

2.1 A Model for Business Processes

We assume the existence of a countably infinite set \( I \) of (object) identifiers (or IDs). Let \( S \) be a finite set of states, and subsets \( S_1, S_2 \subseteq S \) the sets of initial and final states respectively. An object is a pair \((o, q)\) where \( o \in I \) is a (unique) ID and \( q \in S \) is a state.

We first introduce elements “activity” and “gateway” in business processes. An activity represents a unit of work. In our model an activity has a name (label) and works on a set of objects where the activity might change the states of its objects and possibly data contents (the data is not modeled in this paper). More formally, an activity is a triple \((\alpha, O, \tau)\) where \( \alpha \) is a unique activity name, \( O \) is a set of object IDs, and \( \tau \subseteq S[O] \times S[O] \) is a set of transitions with \( |O| \) incoming and \( |O| \) outgoing states. We denote an activity \((\alpha, O, \tau)\) simply by \( \alpha \), and denote the set of all activities by \( \mathcal{A} \). Note that a silent activity, i.e., an activity that does nothing, can be presented with the empty sets of objects and transitions. Activities here can also model decision nodes in decision-aware process modeling [4].

A gateway controls the divergence and convergence of sequence (execution) flows. There are four kinds of gateways: (exclusive) choice and merge, and (parallel) split and join. A choice gateway simulates an if-else statement: exactly one of the outgoing flows will be chosen. An object and a set of states is associated with a choice gateway where the decision is based on the current state of the object in this set. In our formalism, we denote a choice gateway as a pair \((o, \chi)\) where \( o \) is an (object) ID, and \( \chi \subseteq S \) is a set of states. A merge gateway continues an incoming flow, a split gateway forwards a flow to every outgoing edge, and finally, a join gateway synchronizes flows from all incoming edges and combines them into one outgoing edge. We use symbols \( M, S, \) and \( J \) to denote merge, split, and join gateways, respectively.

We restrict process schemas to be well-structured, i.e., can be composed from sequential, conditional, parallel, and loop blocks through a finite number of times [21, 29].

Informally, a well-structured process schema is either an activity or a composite block. A composite block consists of two sequential, two conditional, two parallel, or a loop sub-block where each of them is again either an activity or a composite block.

DEFINITION 2.1. A (process) schema is a labeled graph \((N, s, f, L, E, O)\) with a set \( N \) of nodes, a set \( E \) of edges, a single source \( s \), a single sink \( f \), and a set of object IDs \( O \) whose nodes \( N \) are labeled by the function \( L \) with either a gateway or an activity, recursively defined as follows:

1. If \( \alpha \) is an activity name in \( \mathcal{A} \) with a set of object IDs \( O \) and a set of transitions \( \tau \), \((s, f, u), s, f, (u \mapsto \alpha), (s, u), (u, f))\), \( O \) is an atomic schema, where \( s, f, u \) are nodes with start \( s \) and end \( f \).

Assume now \( P_1 = (N_1, s_1, f_1, L_1, E_1, O_1) \) for \( i = 1, 2 \) is a process schema such that \((s_1, u_1), (v_1, f_1)\) are edges in \( E_1 \), and \( N_1 \cap N_2 = \emptyset \).

2. A sequence schema of \( P_1 \) and \( P_2 \), denoted as \( P_1 \circ P_2 \) is \((N_1 \cup N_2 - \{ f_1, s_2 \}, s_1, f_2, L_1 \cup L_2, E_1 \cup E_2 \cup \{(v_1, u_2)\} - \{(v_1, f_1), (s_2, u_2)\}, O_1 \cup O_2) \) (note that a new edge \((v_1, u_2)\) shows that \( P_2 \) begins after \( P_1 \) ends).

3. A conditional schema of \( P_1 \) and \( P_2 \), denoted as \( P_1 \mid P_2 \) is \((N_1 \cup N_2, s_1, f_1, L_1 \cup L_2 \cup \{ s_2 \mapsto (o, \chi) \}, f_2 \mapsto M_2, E_1 \cup E_2 \cup \{(s_1, s_2), (s_2, u_1), (v_1, f_2)\}, (f_2, f_1)\} - \{(s_1, u_1), (v_1, f_1)\}, O_1 \cup O_2 \cup \{o\}) \) where \( (o, \chi) \) is a choice gateway such that if the state of \( o \) is in \( \chi \) at gateway \( s_2 \), the execution proceeds to \( u_1 \).

4. If \( P_1, P_2 \) share no objects \((O_1 \cap O_2 = \emptyset) \), a parallel schema of \( P_1 \) and \( P_2 \), denoted as \( P_1 || P_2 \) is \((N_1 \cup N_2, s_1, f_1, L_1 \cup L_2 \cup \{ s_2 \mapsto S,\)
Lb in each execution of the process, and/or an upper bound (none of them is executed).

coEx if α is executed, then β must be executed (before or after α), and coexistence coEx(α, β) says either both α and β are executed, or none of them is executed.

Each of the final three classes of DecSerFlow constraints can be further divided into three directions: "response" (Res) which specifies that an activity should happen in the future, "precedence" (Pre) which specifies that an activity should have happened in the past, and "succession" (Suc) which combines both response and precedence. Ordering constraints consider the order of activities. Res(α, β) (Pre(α, β)) specifies that each occurrence of α is followed (preceded) by an occurrence of β and Suc(α, β) requires both to be satisfied. Alternating constraints strengthen the ordering constraints, e.g., aRes(α, β) specifies that in addition to Res(α, β), α and β have to alternate. Chain constraints, i.e., cRes(α, β), cPre(α, β), and cSuc(α, β), are even stricter, which require that the executions of the two activities (α and β) are consecutive.

2.4 SPIN-based Verification

We now introduce a SPIN-based verification algorithm to verify DecSerFlow constraints in a given process schema. Spin [18] is the main model checker used in the verification community and supports the verification of LTL properties of models specified in Promela, a C-like modeling language. Here we develop a translation from business process model to Promela. The main elements are objects, activity nodes, and gateways. We define each object as a byte variable and define a typedef variable that contains all objects. Fig. 3 shows the Promela code for a part of the schema in Fig. 1a. As can be seen, objects is a typedef that contains an object o and states is an instance of objects which is used by nodes. For each node in process schema a Promela process (proctype) is defined. Start node s() only runs the next node n5(). We use if statements in choice gateways to choose the next Promela process to run. In choice node n5(), if the state of object o is b, proctype n1() will be executed, otherwise n3() runs. if statements are also used to code state transitions within an activity, e.g., state of object o can be changed from b to a, b to b, or b to c in proctype n1().

```
typedef objects { byte o; } objects states;

proctype s() { //start node sun n3();
}
proctype n5() { //choice gateway 5
if
: states.o==2 -> run n1();
: else -> run n3();
;
}
proctype n1() { //activity node 1
if
: states.o==2 -> {states.o=1};
: states.o==2 -> {states.o=1};
: states.o==2 -> {states.o=1};
:
run n2();
;
}
```

Figure 3: A Promela Code for Process Schema in Fig. 1a

Since decSerFlow constraints are a form of LTL formulas and SPIN can verify LTL, we can easily translate and check the constraints.

3 Process Trees and Auxiliary Data

In this section, we first define the notion of process trees and then formulate auxiliary data that are stored for process tree nodes and used to incrementally check constraints. Finally, a bottom-up algorithm to construct auxiliary data is presented.
3.1 Process Tree

The recursive nature of the definition of a process schema naturally leads to a tree: The root of each (sub)tree corresponds to a schema with children (child) correspond(s) to schema(s) used in constructing the schema.

Definition 3.1. Given a process schema \( P = (N, s, f, L, E, O) \), process tree \( T(P) \) of \( P \) is a tuple \((V, V^+, V^R, L_p)\) with a node set \( V \), two partial functions \( V^+, V^R \) and a node labeling function \( L_p \) assigning each node an activity \( a \) in \( \mathcal{A} \), a choice gateway (\( \alpha, \chi \)) split (parallel) gateway symbol \( \delta \), or \( \zeta \) (sequential composition), recursively defined as follows:

1. If \( P \) is an atomic schema consisting of an activity node \( u \) labeled by activity \( a \), \( T(p) = (u, \emptyset, \emptyset, u \rightarrow a) \).

2. Assume \( P_1 = (N_1, s_1, f_1, L_1, E_1, O_1) \) (for \( i = 1, 2 \)) is a process schema where \( N_1 \cap N_2 = \emptyset \), let \( T(P_i) = (V_i, V^+_i, V^R_i, L_{p_i}) \) be the corresponding process tree of \( P_i \) rooted at node \( u_i \in V_i \), \( V_1 \cap V_2 = \emptyset \), and \( u \notin \{V_1 \cup V_2\} \) is a new node, then \( T(P) = T(P_1 \cup P_2) \) is a conditional composition of \( P_1 \) and \( P_2 \) with a choice gateway node \( v \) then \( T(P) \) consists of two subtrees \( T(P_1) \) and \( T(P_2) \) with node \( v \) as the root where \( u_1 \) and \( u_2 \) are \( v \)'s left and right child nodes, we call \( v \) a sequence node.

3. If \( P = P_1 \parallel P_2 \) is a conditional composition of \( P_1 \) and \( P_2 \) with a choice gateway node \( v \) then \( T(P) \) consists of two subtrees \( T(P_1) \) and \( T(P_2) \) with node \( v \) as the root where \( u_1 \) and \( u_2 \) are \( v \)'s left and right child nodes, we call \( v \) a conditional node.

4. If \( P = P_1 || P_2 \) is a parallel composition of \( P_1 \) and \( P_2 \) with a split gateway node \( v \) then \( T(P) \) consists of two subtrees \( T(P_1) \) and \( T(P_2) \) with node \( v \) as the root where \( u_1 \) and \( u_2 \) are \( v \)'s left and right child nodes, we call \( v \) a parallel node.

5. If \( P = P_1^* \) is a loop composition of \( P_1 \) with a choice gateway node \( v \), then \( T(P) \) consists of a subtree \( T(P_1) \) with node \( v \) as the root where \( u_1 \) is \( v \)'s left child node, we call \( v \) a loop node.

Example 3.2. Fig. 1b shows a process tree corresponding to the schema in Fig. 1a where 1, 2, 3, and 4 are activity nodes, 5 is a conditional node, 8 is a loop node, and 9 and 10 are the sequence nodes.

3.2 Auxiliary Data Stores

The aim of the incremental verification approach is to verify an evolved process without checking all the nodes. To achieve this goal, we need to store some data for each node within a process tree and use it during the verification step. Auxiliary data stores keep track of the execution paths of the corresponding schema. To begin with, let \( P = (N, s, f, L, E, O) \) be a process schema with \( \alpha_1, \alpha_2, \ldots \) \( \alpha_n \) is an enumeration of objects in \( O \) and \( \omega = \alpha_1 \omega_2 \ldots \alpha_n \). We define \( \bar{S} \) as an n-ary relation consisting of all possible state combinations of objects \( \alpha_1 \)'s. We also define an extended transition relation \( t \) for each activity \( (a, O', \tau) \) in the schema \( P \) as the product of transitions in \( \tau \) and the states of objects that are not in \( O' \) (but arranged in the same order as the enumeration).

Example 3.3. Consider a schema \( P \) with a set of objects \( O = \{\alpha_1, \alpha_2\} \) and a set of states \( S = \{s_1, s_2\} \). Relation \( \bar{S} = \{(s_1, s_1), (s_1, s_2), (s_2, s_1), (s_2, s_2)\} \) (all possible state combinations of two objects), and if we have an activity \( (\alpha, \{\alpha_1\}, \tau) \) where \( \tau = \{(s_1), (s_2)\} \), then \( \bar{T} = \{((s_1, s_1), (s_2, s_1)), ((s_1, s_2), (s_2, s_2))\} \).
Auxiliary data stores can be constructed for the nodes of a tree in the bottom-up fashion. We first construct the auxiliary data for the leaf (activity) nodes and then construct an auxiliary data store for each internal node using that of its child(ren). Here we assume the process schemas have the liveness property, i.e., executions never enter an infinite loop. Before we explain the algorithm, we discuss schemas containing parallel compositions. Note that parallel sub-schemas have no common objects, so the order of the execution of different nodes in different branches does not affect the states of objects in the corresponding join node. Considering this fact, since for the cardinality and existence constraints the order of activities is not important, we could conveniently replace the parallel composition by a sequential one of the same sub-schemas. In the last three classes of constraints, we still can replace a parallel composition by a sequential one of the same sub-schemas. In the last three classes of constraints, we still can replace a parallel composition by a sequential one of the same sub-schemas.

Algorithm 1: Bottom-up Construction of Auxiliary Data

**Input:** Process Schema $P$, Process Tree $T(P)$, Constraint class $\varphi$

**Output:** Process Tree $T(P)$ with auxiliary data stores

1. Let $Q$ be an empty queue
2. Add all activity nodes within process $P$ to $Q$
3. while $Q$ is not empty do
   4. $u = Q.dequeue()$
   5. if $u$ is an activity node where $L(u) = (\alpha, O,$ $\tau)$ then
      6. for all $(\bar{x}, \bar{y}) \in \tau$ do
         7. Add $(\bar{x}, \bar{y}, C)$ to $D_u$
      8. end for
   9. else if $u$ is a sequence node with child nodes $v_1$ and $v_2$ then
      10. $D_u(\bar{x}, \bar{y}, f_T(C_1, C_2, \varphi)) \Leftarrow D_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2)$
   11. else if $u$ is a conditional node with child nodes $v_1$ and $v_2$ where $L(u) = (x_1, \bar{\chi})$ then
      12. $D_u(\bar{x}, \bar{y}, C) \Leftarrow D_{v_1}(\bar{x}, \bar{y}, C, \chi_1(x_1))$
      13. $D_u(\bar{x}, \bar{y}, C) \Leftarrow D_{v_2}(\bar{x}, \bar{y}, C, \chi_2(x_1))$
      14. else if $u$ is a parallel node with child nodes $v_1$ and $v_2$ then
      15. $D_u(\bar{x}, \bar{y}, f_P(C_1, C_2, \varphi)) \Leftarrow D_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2)$
   16. else if $u$ is a loop with a child $v_1$ where $L(u) = (x_1, \bar{\chi})$ then
      17. $T_u(\bar{x}, \bar{y}, C) \Leftarrow D_{v_1}(\bar{x}, \bar{y}, C, \chi_1(x_1))$
      18. $T_u(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \Leftarrow T_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2, \chi_1(x_1))$
      19. $T_u(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \Leftarrow T_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2, \chi_1(x_1))$
      20. $T_u(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \Leftarrow T_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2, \chi_1(x_1))$
      21. end if
   22. $Q.enqueue(u.parent)$
23. end while

Figure 4: Execution Paths within Schema of Fig. 1a
Existence Constraint
to with auxiliary data for two different classes of constraints. Recall that


\[ D_D \]

is created by the third rule. Sequence node (function \( f \)) the transitions and count either

\[ \alpha \]

nodes, i.e., nodes \( \phi \)[\( x \)] and

\[ \chi \]

(another constraint) and tuples in

\[ D_U \]

whose input states are not in

\[ \phi \]

(lines 12-13). Unary relations

\[ \chi (x_1) \]

and

\[ 2 \]

\( \phi \)

(lines 12). These paths can enter to the loop schema. Then

\[ u \]

and

\[ v \]

to the element

\[ \phi \]

paths that did not enter the loop schema

\[ \phi \]

to find the set of all possible distinct paths through the loop. The third rule (line 19) captures all the possible execution paths that did not enter the loop schema (\( x_t \not\in \phi \)). Here \( C_0(\phi) \) refers to the element \( C \) with only one path where the path does not visit any nodes, i.e., for the cardinality constraints \( C_0(\phi) = \{0\} \), for the existence constraints \( C_0(\phi) = \{0, 0, 0, 0, 0, 1\} \), for the ordering and alternating constraints \( C_0(\phi) = \{0, 0, 0, 0, 0, 0\} \), and for the chain constraints \( C_0(\phi) = \{0, 0, 0, 0, 0, 0\} \). Finally, in line 20 we select the paths of \( T_u \) that can go out of the loop (\( y_i \not\in \phi \)) and put them into \( D_u \). For simplicity, we use Datalog syntax to explain the rules.

Example 3.6. Fig. 5a shows the process tree of Fig. 1b augmented with auxiliary data for two different classes of constraints. Recall that each record in an auxiliary data store is a tuple \((x, y, c)\) where \( x \) and \( y \) are states (here, \( a, b, c \), or \( c \)) and \( C \) counts the execution paths. In Fig. 5a the auxiliary data stores for the cardinality constraint regarding of activity \( a_t \) (for example \( U(a_2) = 1 \)) are shown. For the activity

\[ a_t \]

nodes, i.e., nodes 1, 2, 3, and 4, the auxiliary data stores contain the transitions and count either [0] (for all activity nodes except 4) or [1] (for node 4). To compute \( C \) for sequence node 9 we use function \( f_0 \) where for the cardinality constraints, \( f_0(C_1, C_2, \phi) = m + n/m \in C_1, n \in C_2 \) which basically adds the counts of two child nodes. Here \( D_0 = (b, c, \{0, 0\}) \) where \( (b, c) \) is the join of the transitions of node 1 and 2. Since \( (b, c) \) is resulted by two different paths and neither of them go through node 4, element \( C \) for \( D_0 \) is \([0, 0]\). For the loop node 8, we have four execution paths: one from \( b \) to \( c \) which is created by the first rule (line 17 of the algorithm), two transitions from \( a \) to \( c \); one comes from the child node 4, and one is the composition of \( (a, b) \) and \( (b, c) \), and one transition \((c, c)\) which is created by the third rule. Sequence node 10 joins the transitions of nodes 3 and 8 resulting in four transitions from \( a \) to \( c \). Element

\[ C = \{0, 1, 1, 2\} \]

shows the number of occurrences of node 4 in these four transitions. The conditional node 5 (root node) simply collects the tuples of its child nodes 9 and 10.

Fig. 5b shows the auxiliary data stores for the existence constraint \( E(x_t, a_t) \) where element \( C \) for the transitions of activity nodes 1 and 2 is \([0, 0, 0, 1]\), for node 3 is \([1, 0, 0, 0]\), and for node 4 is \([0, 1, 0, 0]\). Function \( f_0(C_1, C_2, \phi) \) returns \([1, 0, 3, 0]\) as element \( C \) of the auxiliary data of node 10 which is constructed from auxiliary data of nodes 3 and 8. Here, 1 is the join of \((a, c)\) and \((c, c)\) and 3 is the sum of two joins; first join of \((a, a)\) and \((a, c)\), and second join of \((a, b)\) and \((b, c)\). Element \( C \) for the loop and the conditional nodes are constructed in a straightforward manner. Similarly, we construct the auxiliary data stores for the other three classes of constraints. □

4 Incremental Construction of Auxiliary Data

In this section, we first propose an incremental approach to update auxiliary data stores, and then show how to verify processes by checking only the root node of the corresponding tree. The incremental approach is useful when a schema is updated. When a process is modified, the leaf level (activity) nodes and their auxiliary data are updated, and the changes in auxiliary data are propagated incrementally only along the path to the root node. Therefore, we do not need to check the whole process again. In this approach for each node \( u \) we define two relations \( D_u^+ \) and \( D_u^- \) to contain the elements which are added to or removed from auxiliary data store \( D_u \) respectively. Relations \( D_u^+ \) and \( D_u^- \) of each node are then used to construct: (1) the updated auxiliary data store \( D'_u \) of the node, and (2) the relations \( D_u^+ \) and \( D_u^- \) of the parent node. We first explain the

**Algorithm 2: Incremental Propagation of Auxiliary Data**

**Input:** Process Tree \( T(P) \), Auxiliary data store \( D_v \) for each node \( v \) in \( T(P) \). Updated node \( u \) with auxiliary data stores \( D_u^+ \) and \( D_u^- \), and Constraint class \( \phi \).

**Output:** Process Tree \( T(P) \) with updated auxiliary data stores.

1. Let \( Q \) be an empty queue.
2. Add all activity nodes within process \( P \) to \( Q \).
3. \( curr = u \)
4. while \( curr \neq null \) do
   5. \( curr = curr.parent \)
5. Let \( v_1 \) and \( v_2 \) be the child node(s) of \( curr \) where \( v_1 \) is the updated node
   6. if \( curr \) is a sequence node then
      7. \( D_v^-(x, y, f_0(C_1, C_2, \phi)) \leftarrow D_v^-(x, z, C_1), D_v^-(x, y, C_2) \)
   8. else if \( curr \) is a conditional node and \( L(curr) = (x_1, \chi) \) then
      9. \( D_v^+(x, y, C) \leftarrow D_v^-(x, y, C), \chi(x_1) \)
     10. else if \( curr \) is a parallel node then
       11. \( D_v^+(x, y, f_0(C_1, C_2, \phi)) \leftarrow D_v^+(x, z, C_1), D_v^+(x, y, C_2) \)
     12. else if \( curr \) is a loop node and \( L(curr) = (x_1, \chi) \) then
       13. \( T_{curr}^+(x, y, f_0(C_1, C_2, \phi)) \leftarrow T_{curr}^+(x, z, C_1), D_v^+(x, y, C_2), \chi(x_1) \)
      14. \( T_{curr}^+(x, y, f_0(C_1, C_2, \phi)) \leftarrow T_{curr}^-(x, z, C_1), D_v^+(x, y, C_2), \chi(x_1) \)
     15. \( D_v^+(x, y, C) \leftarrow T_{curr}^+(x, y, C), \chi(y_1) \)
     16. \( D_v^-(x, y, C) \leftarrow T_{curr}^-(x, y, C), \chi(y_1) \)
     17. end
     18. end
   19. end while

20. □
incremental approach to create the auxiliary data store for the leaf node(s) and then propagate the changes along the path to the root in the process tree.

If the change operation is a `RevActivity` or a shrink operation, we construct the auxiliary data for the resulting (new) activity node where for each transition \((x, y, C)\), a tuple \((x, y, C)\) is added to the updated auxiliary data store \(D'\) of the activity. Same as before element \(C\) is constructed depending on the class of constraint. For expand operations where a single activity node is replaced by a (sub-)tree corresponding to a sequential, conditional, parallel, or loop composition of activity node(s), we first construct the auxiliary data for the activity node(s) and then, use that to construct the auxiliary data of the internal node. Since both the activity and the internal nodes are new, we use the bottom-up algorithm to construct auxiliary data of them. We now proceed to formulate the propagation of the auxiliary data updates to the nodes along the path to the root. Since we have four different types of internal nodes, four different cases for propagation are needed. Algorithm 2 presents the incremental propagation of auxiliary data. In the algorithm, we assume that \(D'_{n_1}\) is the updated auxiliary data of node \(u\), and \(D'_{n_2}\) and \(D'_{n_2}\) are the tuples which are added to or removed from \(D_n\) respectively. Now, we want to update auxiliary data of nodes along the path from \(u\) to the root. The algorithm only shows the construction of the relation \(D'_{curr}\) for each node \(curr\). To construct the relation \(D'_{curr}\) we just need to replace + with \(−\) in the relations. In addition, for the sequence, conditional and parallel nodes, we assume that the left child of the current node has been updated. We can use the similar rules for the right node as well. Let us call the current node \(curr\), if \(curr\) node is a sequence node, a join of \(D'_{n_1}\) and \(D_{n_2}\) is sufficient to construct relation \(D'_{curr}\). To construct \(C\) we use the same function \(f_Q(C_1, C_2, \varphi)\) as before. For the conditional node we just collect the tuples in \(D'_{n_2}\) whose input states are in \(\chi\) (unary relation \(q_1(x)\)). The relation \(D'_{curr}\) for the parallel nodes is constructed in the similar way as the sequence node, except that function \(f_P(C_1, C_2, \varphi)\) is used to construct \(C\). For the loop nodes, we collect the tuples in \(D'_{n_2}\) whose input states are in \(\chi\) in a temporary relation \(T_{curr}\). Then, we recursively construct the transitive closure of \(T_{curr}\) to find the set of all possible distinct paths through the loop. Finally, we select the paths that can go out of the loop \((y_i \not\in \chi)\) and put them into \(D'_{curr}\). When both \(D'_{curr}\) and \(D^*_{curr}\) are constructed, the relation \(D'_{curr}\) is defined as \(D_{curr} = D'_{curr} + D^*_{curr}\).

Example 4.1. Continuing with Example 3.6, let a `RepBySeq` change operation evolves the process schema of Fig. 1a, by replacing activity node 1 with a sequence of activity nodes 1 and 11. Let \(L(1) = (\alpha_{11}, \emptyset, \{(a, b), (b, c, (c, c))\})\). Fig. 6a and Fig. 6b represent the updated auxiliary storages of the tree concerning the cardinality constraint \(Ub(\alpha_4) = 1\) and the existence constraints \(Ex(\alpha_4, \alpha_9)\). To construct the auxiliary data storage for activity node 11 and sequence node 12, we use the bottom-up construction algorithm. Then we compute relations \(D_{12}^+\) and \(D_{12}^-\) for the sequence node. Since node 12 is in the earlier position of node 1 (being left child of node 9), \(D_{12}^+\) and \(D_{12}^-\) are constructed by comparing auxiliary data of node 12 and 11 resulted in \(D_{12}^+ = (b, a, C)\) and \(D_{12}^- = (b, c, C)\). Note that since node 12 already had a path from \(b\) to \(c\), we just update the counter \(C\) of that path in \(D'_{12}\). Relations \(D_{12}^+\) and \(D_{12}^-\) are then propagated to the parent node 9 where \(D_{12}^+\) does not affect its auxiliary data, but \(D_{12}^-\) creates a new path from \(b\) to \(c\), so \(D_9^- = 0\) and \(D_9^+ = (b, c, C)\). Similarly, \(D_9^+\) is propagated to the root node which again causes a new path from \(b\) to \(c\) and \(D_2^+\) will be \((b, c, C)\). As can be seen, the algorithm only works on the auxiliary data stores of nodes along the path from node 11 to the root.

4.1 Verification of Constraints

When the auxiliary data stores are constructed, we only need to check the auxiliary data of the root node to verify a given constraint. The checking is performed for the complete execution paths (from an initial to a final snapshot). To check cardinality constraint \(Ub(\alpha)\), all \(c\)’s in bag \(C\) should be greater than or equal to \(Ub(\alpha)\). \(Ub(\alpha)\) checks \(c\)’s to be less than or equal to \(Ub(\alpha)\). \(Ub(\alpha)\) combines the \(Lb(\alpha)\) and \(Ub(\alpha)\), and \(Fix(\alpha)\) requires all \(c\)’s to be the same. \(Ex(\alpha, \beta)\) is satisfied if there is no path consisting \(\alpha\) but not \(\beta\), so \(C[0] = \emptyset\) should be equal to 0. To satisfy \(cEx(\alpha, \beta)\), in addition to the previous condition, \(C[1]\) has to be 0, means there is no path consisting \(\alpha\) but not \(\beta\). \(Res(\alpha, \beta)\) needs all occurrences of \(\alpha\) to be followed by a \(\beta\), i.e., \(C[0] = C[2] = 0\). Similarly, the remainder sets of constraints can be checked.

Example 4.2. Continuing with Example 4.1, since \(D_5 = ((a, c, \{0, 1, 2\}), (b, c, \{0, 0\}))\), there is path with more than on occurrences of \(\alpha_4\), thus \(Ub(\alpha_4) = 1\) is not verified. Existence constraint \(Ex(\alpha_4, \alpha_9)\) is also not verified because \(D_5 = ((a, c, \{0, 1, 3, 0\}), (b, c, \{0, 0, 0, 2\}))\) and there is path that goes through \(\alpha_4\) but not \(\alpha_4\).
We compute the times needed for the approaches for sequential Bottom-up approach has an exponential growth rate in the size of the state space is affected by two parameters: objects and states. Spin-based algorithm is more expensive in terms of the number of transitions and loops. In comparison to the bottom-up, Spin takes less time to verify the results of all three approaches where although for small processes the Spin-based approach than the other approaches report the findings. re: 10 and 11 show the time needed for all three approaches to verify the schemas with 1 ≤ #T ≤ 32 and 1 ≤ #L ≤ 10 respectively (in Fig. 10, #L=1 and in Fig. 11, #T=3) where the incremental algorithm takes less time to verify schemas. The number of transitions directly affects the state space of Spin whereas in our algorithms, although it affects the number of leaf level joins, the resulted paths may not be affected much. The number of loops mostly affects the number of visited states in Spin, but in the other algorithms it causes more complicated computation to obtain transitive closure of paths and to update the resulted paths. However, the results show that even the bottom-up construction has a better performance than Spin. It should be noted that Spin reports results as soon as it finds the first error (unsatisfied constraint), however, in our algorithms, checking is the last step, therefore, Spin could have a better performance for erroneous processes. On the other hand, (1) Spin checking time strongly depends on the developer experience and how the process is translated to the input language Promela, and (2) to check several constraints from the same class, while Spin takes much longer time to verify, the time does not increase in our algorithms (the verification time is negligible in comparison to the auxiliary data construction time in our algorithms).

6 Related Work

Business process modeling frameworks can be divided into three groups of activity-based, object-aware, and artifact-centric, where activity-based [33] and object-aware [4] verification techniques mostly focus on the soundness of business processes. Artifact-centric paradigm treats data as first-class citizens [9]. Since the domain of data is infinite, in general verification is undecidable [9]. However, different techniques are presented to verify temporal logic properties on artifact systems. These techniques are either applied to the restricted classes of artifact systems [16] or reduce the verification problem to standard model checking [5]. Incremental evaluation is a technique that uses online algorithms to evaluate or verify a system. Incremental evaluation of database queries and maintenance of views [11] has been studied for more than two decades. In [15] an algorithm to compute changes to a view in response to relations updates, and a counting algorithm to count the modified tuples for non-recursive views is presented which is adapted in our approach. Incremental verification of software systems is studied especially for model checking and program analysis [26], [34]. A syntactic-semantic approach [6] is also presented for the incremental verification of workflows where the approach focuses on the probabilistic verification of the reliability requirements of processes. In addition, an incremental method to verify BPEL processes is presented in [20]. However that method only checks
whether two parallel activities have access to the same object or not (the non-conflict property).

The problem of verifying whether a process model satisfies the requested compliance rules, e.g., laws, regulations, and service level agreements, is studied [24] [17] [14] form both control flow [2] and data [1] [3] [27] aspects of processes. Furthermore, recent techniques use the event log of processes to check compliance [10][25]. Our work is also related to process evolution, activity-centric [8] or object-aware [23] frameworks. While in this paper a set of primitive update operations are introduced, some researches propose advanced change patterns like swap fragments [31]. However advanced patterns can be replaced by a sequence of our update operations. An approach for propagating changes from an internal, private process to its public process view is also developed in [13] that only considers activity-centric processes.

7 Conclusions
Incremental computation is an effective method and has many applications from online computation to database query evaluation. This paper makes an initial application of this approach to process verification. We present VIEW, an incremental approach to verify workflows. We model a process schema as a process tree and construct auxiliary data stores for nodes of the tree based on the input constraint. When a schema change happens, we construct auxiliary data stores for new nodes, then, we incrementally update auxiliary data stores of nodes along the path to the root, and finally, we verify the given constraint by checking only the root of the tree. Our experiment shows the efficiency of the incremental approach for a large class of processes.

VIEW has two main limitations. First, it supports only well-structured business processes. Although several algorithms are proposed to convert unstructured processes to well-structured ones [22] [12], these methods are not able to fully convert all classes of business processes. Second, the domain of data values has to be finite. Note that “finite domain” data can be captured using object states by assigning a state to each value. Many interesting questions remain. Generalizing the supported constraints makes the approach more practical. We currently support a set of LTL constraints regarding the cardinality, existence, and ordering of the activities. The approach could simply be extended to support the same set of constraints regarding the object states. To support other kinds of constraints we might need to change the structure of our auxiliary data. Another practical problem is to suggest a set of change operations on a process to make the specified constraints satisfiable. In addition, since we count all the paths within a process model, the techniques presented in this paper can easily be applied to the model counting problems.

Acknowledgement
The authors would like to thank Professor Jianwen Su, University of California Santa Barbara, for his appreciative contribution and insightful comments provided in the formative stages of this research.

References