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:
Mohammad Javad Amiri and Divyakant Agrawal. 2019. VIEW: An Incremental Approach to Verify Evolving Workflows. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297280.3297291

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

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3297291

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 an 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.

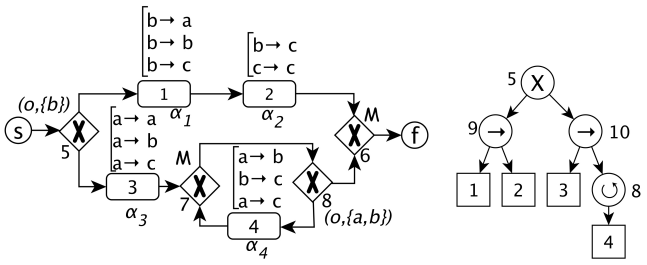
2 Business Processes: Model, Constraints, and Evolution

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 $\mathcal{I}$ of (object) *identifiers* (or IDs). Let S be a finite set of states, and subsets $S_I, S_F \subseteq S$ the sets of *initial* and *final* states respectively. An *object* is a pair (o, q) where $o \in \mathcal{I}$ 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 (α, O, τ) where α 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 (α, O, τ) simply by α, 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) A conditional Process schema P **(b) process Tree $T(P)$**

Figure 1: A Process Schema and its Tree

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

Assume now $P_i = (N_i, s_i, f_i, L_i, E_i, O_i)$ (for $i = 1, 2$) is a process schema such that (s_i, u_i), (v_i, f_i) are edges in E_i, 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 $P_1 \cup P_2$ is $(N_1 \cup N_2, s_1, f_1, L_1 \cup L_2 \cup \{s_2 \mapsto (o, \chi), f_2 \mapsto \mathtt{M}\}, E_1 \cup E_2 \cup \{(s_1, s_2), (s_2, u_1), (v_1, f_2), (f_2, f_1)\} - \{(s_1, u_1), (v_1, f_1)\}, O_1 \cup O_2 \cup \{o\})$ where (o, χ) is a choice gateway such that if the state of o is in χ 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 $(N_1 \cup N_2, s_1, f_1, L_1 \cup L_2 \cup \{s_2 \mapsto \mathtt{S},$

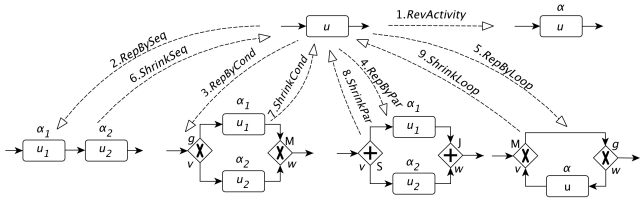


Figure 2: Process schema change operations

$f_2 \mapsto \mathtt{J}\}, E_1 \cup E_2 \cup \{(s_1, s_2), (s_2, u_1), (v_1, f_2), (f_2, f_1)\} - \{(s_1, u_1), (v_1, f_1)\}, O_1 \cup O_2)$,

(5) a *loop* schema of P_1 denoted as P^* is $(N_1 \cup \{s, f\}, s, f, L \cup \{s_1 \mapsto (o, \chi), f_1 \mapsto \mathtt{M}\}, E_1 \cup \{(s, f_1), (s_1, f), (f_1, s_1)\}, O_1 \cup \{o\})$, where (o, χ) is a choice gateway such that if the state of o is in χ at gateway node s_1 the execution proceeds to u_1 (enters to the loop).

EXAMPLE 2.2. Fig. 1a shows a conditional process schema. Let $\{o\}$ be the singleton set of objects and $S = \{a, b, c\}$ ($S_{\mathrm{I}} = \{a\}, S_{\mathrm{F}} = \{c\}$) be the set of states. s and f are the start and final nodes, 1 to 4 are activity nodes which are labeled by α_i ($1 \leqslant i \leqslant 4$), e.g., $L(1) = (\alpha_1, \{o\}, \{(b, a), (b, b).(b, c)\}$. Nodes 5 and 8 are two choice gateways where $L(5) = (o, \{b\})$, and $L(8) = (o, \{a, b\})$. Finally, 6 and 7 are merge gateways ($L(6) = L(7) = \mathtt{M}$). □

2.2 Business Process Change Operations

We now proceed to introduce 9 operations to update a process schema. The *RevActivity* operation replaces an activity associated with a node by another activity. The other 8 operations are divided into two groups to *expand* (replacing an activity node by a sequence/-conditional/parallel/loop block) and *shrink* schemas (replacing a sequence/conditional/parallel/loop block by an activity node).

Fig. 2 shows different update operations. *RevActivity* simply modifies the label of node u to an activity α (set of objects and transitions may change), The expand operation *RepBySeq* replaces an activity node u with a sequence of two activity nodes u_1, u_2 that are labeled by α_1 and α_2 resp. *RepByCond* replaces an activity node u with a conditional schema containing two activities α_1 and α_2, a choice gateway v, and a merge gateway w. The remaining two operations *RepByPar* and *RepByLoop* are similar. The four shrink operations reverse the expand operations in a straightforward manner.

2.3 DecSerFlow Constraints

We now present DecSerFlow constraints [28]. DecSerFlow constraints can be categorized into five classes: cardinality, existence, ordering, alternating, and chain.

Cardinality constraints define the required number of executions of an activity node (name) in a process in terms of a lower bound $Lb(\alpha)$ that specifies the minimum number of occurrences of activity α in each execution of the process, and/or an upper bound $Ub(\alpha)$ that defines the maximum number of occurrences of activity α. The combination of $Lb(\alpha)$ and $Ub(\alpha)$ can either specify the exact number $Fix(\alpha)$ of occurrences of activity α, or a range $Rng(\alpha)$ of occurrences of activity α in each execution.

The *existence* class has no restrictions on temporal orders and consists of two constraints. Responded existence $Ex(\alpha, \beta)$ specifies if α is executed, then β must be executed (before or after α), and coexistence $coEx(\alpha, \beta)$ 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. $Res(\alpha, \beta)$ ($Pre(\alpha, \beta)$) specifies that each occurrence of α is followed (preceded) by an occurrence of β and $Suc(\alpha, \beta)$ requires both to be satisfied. *Alternating* constraints strengthen the ordering constraints, e.g., $aRes(\alpha, \beta)$ specifies that in addition to $Res(\alpha, \beta)$, α and β have to alternate. *Chain* constraints, i.e., $cRes(\alpha, \beta)$, $cPre(\alpha, \beta)$, and $cSuc(\alpha, \beta)$, 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 consists 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 chose 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
   run n5();
}
proctype n5(){ //choice gateway 5
   if
   :: (states.o==2) -> run n1();
   :: else -> run n3();
   fi;
 }
proctype n1(){ //activity node 1
   if
   ::(states.o==2)->(states.o=1);
   ::(states.o==2)->(states.o=1);
   ::(states.o==2)->(states.o=1);
   fi;
   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 $(V, V^{\mathrm{L}}, V^{\mathrm{R}}, L_p)$ with a node set V, two partial functions $V^{\mathrm{L}}, V^{\mathrm{R}}{:}V \to V$ to specify left and right child nodes (resp.), and a node labeling function L_p assigning each node an activity α in $\mathcal{A}$, a choice gateway (o, χ), split (parallel) gateway symbol S, or Q (sequential composition), recursively defined as follows:

(1) If P is an atomic schema consisting of an activity node u labeled by activity α, $T(p) = (u, \emptyset, \emptyset, u \mapsto \alpha)$,

Assume $P_i = (N_i, s_i, f_i, L_i, E_i, O_i)$ (for $i = 1, 2$) is a process schema where $N_1 \cap N_2 = \emptyset$, let $T(P_i) = (V_i, V_i^{\mathrm{L}}, V_i^{\mathrm{R}}, L_{p_i})$ be the corresponding process tree of P_i rooted at node $u_i \in V_i$, $V_1 \cap V_2 = \emptyset$, and $v \notin (V_1 \cup V_2)$ is a new node,

(2) If $P = P_1 \circ P_2$ is a sequential composition of schemas P_1 and P_2, then $T(P)$ consists of two subtrees $T(P_1)$ and $T(P_2)$ with a new sequence 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 $P = P_1 \cup P_2$ 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 *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*, and

(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 where o_1, o_2, ..., o_n is an enumeration of objects in O, and $\bar{o} = o_1 o_2 ... o_n$. We define $\bar{S}$ as an n-ary relation consisting of all possible state combinations of objects o_i's. We also define an extended transition relation $\bar{\tau}$ for each activity (α, O', τ) in the schema P as the product of transitions in τ 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 $O = \{o_1, o_2\}$ 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 $(\alpha, \{o_1\}, \tau)$ where $\tau = \{(s_1), (s_2)\}$ then $\bar{\tau} = \{((s_1, s_1), (s_2, s_1)), ((s_1, s_2), (s_2, s_2))\}$. □

Recall that each node in a process tree is corresponding to a subschema (block) in its process schema, e.g., node 9 in Fig. 1b is corresponding to the sequence subschema of Fig. 1a consisting of nodes 1 and 2. To construct the auxiliary data for each node of a tree, one idea is to store: (1) all possible execution paths within the corresponding sub-schema and (2) information on whether each execution path satisfies a given constraint or not. Although the idea works, it suffers from two main issues. First, storing all the state transitions within each execution path consumes significant storage, and second, updating the execution paths after each change is costly. A more efficient way is to store only the first and the last state relations of each execution path instead of all the state transitions. Since we might have several execution paths with the same first and last state relations, a counter is used to store the number of each path. Note that without the counter, if a path is removed at evolving time, we cannot decide to either keep or remove that path from the auxiliary store, because there might be other paths with the same first and last states which still exist in the schema.

To define auxiliary data store, we first define a notion of "snapshot" for processes. A *snapshot* of P is a pair $\Sigma = (D, I)$ where $D : O \mapsto S$ assigns each object in P a state, and I is a set of edges in P.

DEFINITION 3.4. Let $P = (N, s, f, L, E, O)$ be a process schema where u has an incoming edge from s and v has an outgoing edge to f, and r be the root node of the corresponding tree. An auxiliary data store of node r is a relation D_r that consists of a set of tuples $(\bar{x}, \bar{y}, C)$ where $\bar{x}, \bar{y} \in \bar{S}$ are two state relations, and C counts execution paths within the schema P from a snapshot $(D_1, \{(s, u)\})$ to $(D_n, \{(v, f)\})$ such that for each $i \in [1 : n]$, $D_1(o_i) = x_i$ and $D_n(o_i) = y_i$.

Element C is defined based on the class of constraint we want to check. In the following, we briefly explain C for different classes of constraints.

For the cardinality constraints, e.g., $Lb(\alpha)$, C is a bag of natural numbers where each c in C shows the number of α's in a distinct path.

In the existence constraints, e.g., $Ex(\alpha, \beta)$, C is an array of size 4 where $C[0]$, $C[1]$, $C[2]$ and $C[3]$ are the number of distinct paths consisting α but not β, β but not α, both α and β, and neither α nor β respectively. We keep information about paths that satisfy a part of the constraint, e.g., paths consisting α but not β, because these paths might compose with other paths and construct satisfiable paths.

In the ordering constraints, e.g., $Res(\alpha, \beta)$, C is an array of size 6 where in $C[0]$ paths there is an α without a following β, and each β is preceded by an α, in $C[1]$ paths there is a β without a preceding α, and each α is followed by a β, in $C[2]$ paths there is an α without a following β, and there is a β without a preceding α, in $C[3]$ paths each α is followed by a β, and each β is preceded by an α, $C[4]$ paths have neither α nor β, and $C[5]$ counts the paths that never satisfy the constraint.

In the alternating constraints, e.g., $aRes(\alpha, \beta)$, also C is an array of size 6 which is defined similar to ordering constraints, except that the following and preceding α and β are distinct. Here, $C[5]$ also counts paths where there are two subsequent α's without any β in between or there are two subsequent β's without any α in between (paths that never satisfy the constraint).

Element C for the chain constraints is an array of size 7 where $C[0]$ to $C[5]$ are defined similar to ones in the alternating constraints,

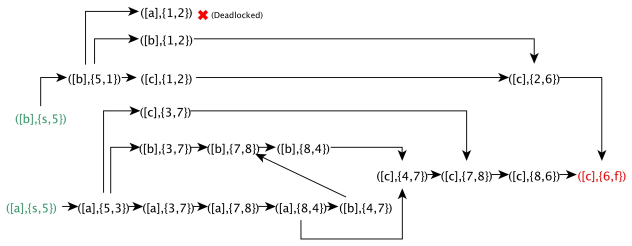


Figure 4: Execution Paths within Schema of Fig. 1a

except that the following and preceding α and β are immediate and $C[6]$ counts paths with no activity nodes. Note that in other classes of constraints we ignore activity nodes other than α or β, but in the chain constraints since the immediate occurrences of activities is important, e.g., $\alpha\beta$ is different from $\alpha\gamma\beta$, we distinguish between paths with activity nodes other than α or β ($C[4]$) and paths without any activity nodes ($C[6]$).

EXAMPLE 3.5. Fig. 4 shows all the execution paths of process schema P in Fig. 1a. Since the process has only one object, we represent each snapshot using a tuple $([s], \{u, v\})$ where s is the state of object o in that snapshot and $\{u, v\}$ is an edge. As can be seen, the process has totally six complete executions where two of them are from $([b], \{s, 5\})$ to $([c], \{6, f\})$ and four of them are from $([a], \{s, 5\})$ to $([c], \{6, f\})$. Here since the process has two initial states, there are two initial snapshots. Note that there is also another path starting from $([b], \{s, 5\})$ which is stuck in activity node 2 (since node 2 has no transition with incoming state a) and not counted as a complete execution (the algorithm ignores such paths because they are not successful executions). We now show the auxiliary stores of the root node 5 for two classes of constraints. If the constraint is a cardinality constraint "$Ub(\alpha_4) = 1$", (activity α_4 is executed at most once) then $D_5 = \{(a, c, \{0, 1, 1, 2\}), (b, c, \{0, 0\})\}$. Tuple $(b, c, \{0, 0\})$ is in D_5, because there are two execution paths from $([b], \{s, 5\})$ to $([c], \{6, f\})$ where neither of them go through activity α_4. On the other hand, there are four execution paths from $([a], \{s, 5\})$ to $([c], \{6, f\})$ where one of them does not visit activity α_4, two of them visit activity α_4 once, and one path visits activity α_4 twice, so $(a, c, \{0, 1, 1, 2\})$ is in D_5. Note that since there is an execution path that goes through activity α_4 twice, the constraint $Ub(\alpha_4)$ is not verified. For the existence constraint $Ex(\alpha_3, \alpha_4)$, relation D_5 is $\{(a, c, [1, 0, 3, 0]), (b, c, [0, 0, 0, 2])\}$. Here tuple $(b, c, [0, 0, 0, 2])$ shows that there are two execution paths from $([b], \{s, 5\})$ to $([c], \{6, f\})$ consisting neither activity α_3 nor α_4, and tuple $(a, c, [1, 0, 3, 0])$ represents that there are four paths from $([a], \{s, 5\})$ to $([c], \{6, f\})$: one path contains activity α_3 but not α_4, and three paths contain both activities α_3 and α_4. Similarly, D_5 can be constructed for other types of constraints. □

3.3 Bottom-up Construction of Auxiliary Data

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

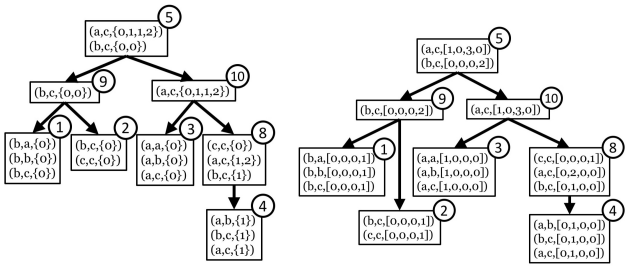
Algorithm 1 Bottom-up Construction of Auxiliary Data

Input: Process Schema P, Process Tree $T(P)$, Constraint class φ
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: $u = Q.deque()$
5: **if** u is an activity node where $L(u) = (\alpha, O, \tau)$ **then**
6: **for all** $(\bar{x}, \bar{y}) \in \bar{\tau}$ **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: $D_u(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \leftarrow D_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2)$
11: **else if** u is a *conditional* node with child nodes v_1 and v_2 where $L(u) = (x_i, \chi)$ **then**
12: $D_u(\bar{x}, \bar{y}, C) \leftarrow D_{v_1}(\bar{x}, \bar{y}, C), \chi_1(x_i)$
13: $D_u(\bar{x}, \bar{y}, C) \leftarrow D_{v_2}(\bar{x}, \bar{y}, C), \chi_2(x_i)$
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 $L(u) = (x_i, \chi)$ **then**
17: $T_u(\bar{x}, \bar{y}, C) \leftarrow D_{v_1}(\bar{x}, \bar{y}, C), \chi_1(x_i)$
18: $T_u(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \leftarrow T_u(\bar{x}, \bar{z}, C_1), D_{v_1}(\bar{z}, \bar{y}, C_2), \chi_1(x_i)$
19: $T_u(\bar{x}, \bar{x}, C_0(\varphi)) \leftarrow (\bar{x}, \bar{x}), \chi_2(x_i)$
20: $D_u(\bar{x}, \bar{y}, C) \leftarrow T_u(\bar{x}, \bar{y}, C), \chi_2(y_i)$
21: **end if**
22: $Q.enque(u.parent)$
23: **end while**

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, but the computation of element C is different from the sequence node. In the case of ordering constraints on activity nodes α and β, if one branch has α and the other one has β, since they could be executed in any order ($\alpha \rightarrow \beta$ or $\beta \rightarrow \alpha$), the ordering constraint cannot be satisfied for all execution paths. In the alternating constraints, a parallel schema might satisfy a constraint, only if one branch has no α nor β, and in the chain constraints if there is any α or β in either branch, the constraint will never be satisfied since the activity nodes of the other branch can be executed immediately after the α or β node. As a result, in the algorithm, we treat the parallel composition as a sequence one, but the computation of C is different for sequence and parallel nodes.

Algorithm 1 shows the bottom-up construction of the auxiliary data stores. We first add all the activity nodes to queue Q. Then, we choose a node from Q, let us call it u, since we have not added any other types of nodes to the queue yet, u is an activity node. For each transition $(\bar{x}, \bar{y})$ of u, we add a tuple $(\bar{x}, \bar{y}, C)$ to D_u (lines 6-7) where C is constructed depending on the class of constraint φ. When the auxiliary data is constructed, we add the parent node of u to queue Q (line 22). If the chosen node u is a sequence node



(a) Cardinality Constraint **(b) Existence Constraint**

Figure 5: Tree of Fig. 1b Augmented with Auxiliary Data

(line 9), we join tuples in $D_{v_1}(\bar{x}, \bar{z}, C_1)$ and $D_{v_2}(\bar{z}, \bar{y}, C_2)$ relations to construct $D_u(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi))$. Here, function f_Q is used to construct C using the auxiliary data of child nodes and the class of constraint φ. For a conditional node, we just collect tuples in D_{v_1} whose input states are in χ (satisfy the condition) and tuples in D_{v_2} whose input states are not in χ (lines 12-13). Unary relations $\chi_1(x_i)$ and $\chi_2(x_i)$ consist of sets of states χ and $S - \chi$ respectively. The auxiliary data of the parallel nodes is computed similar to the sequence nodes, except that we use a different function $f_P(C_1, C_2, \varphi)$ to compute C. If the node u is a loop node with a child node v_1, we first collect tuples in D_{v_1} whose input states are in χ in a temporally relation T_u (line 17). These paths can enter to the loop schema. Then in the second rule (line 18) we recursively construct the transitive closure of T_u 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 ($x_i \notin \chi$). Here $C_0(\varphi)$ refers to the element C with only one path where the path does not visit any nodes, i.e., for the cardinality constraints $C_0(\varphi) = \{0\}$, for the existence constraints $C_0(\varphi) = [0, 0, 0, 1]$, for the ordering and alternating constraints $C_0(\varphi) = [0, 0, 0, 0, 1, 0]$, and for the chain constraints $C_0(\varphi) = [0, 0, 0, 0, 0, 0, 1]$. Finally, in line 20 we select the paths of T_u that can go out of the loop ($y_i \notin \chi$) 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 $(\bar{x}, \bar{y}, C)$ where $\bar{x}$ and $\bar{y}$ are states (here, a, b, or c) and C counts the execution paths. In Fig. 5a the auxiliary data stores for the cardinality constraint regarding of activity α_4 (for example $Ub(\alpha_4) = 1$) are shown. For the activity 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_Q where for the cardinality constraints, $f_Q(C_1, C_2, \varphi) = \{m + n | m \in C_1, n \in C_2\}$ which basically adds the counts of two child nodes. Here $D_9 = (b, c, \{0, 0\})$ where (b, c) is the join of the transitions of nodes 1 and 2. Since (b, c) is resulted by two different paths and neither of them go through node 4, element C for D_9 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 $Ex(\alpha_3, \alpha_4)$ 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 $f_Q(C_1, C_2, \varphi)$ 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. □

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 store D'_u, Relations D^+_u and D^-_u, and Constraint class φ

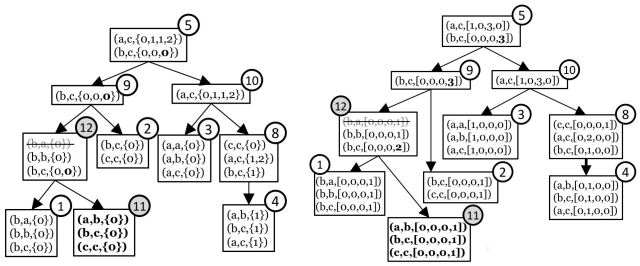
Output: Process Tree $T(P)$ with *updated* auxiliary data stores

1: Let Q be an empty queue
2: Add all activity nodes within process P to Q
3: $curr = u$
4: **while** $curr.parent \neq null$ **do**
5: $curr = curr.parent$
6: Let v_1 (and v_2) be the child node(s) of $curr$ where v_1 is the updated node
7: **if** $curr$ is a *sequence* node **then**
8: $D^+_{curr}(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \leftarrow D^+_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2)$
9: **else if** $curr$ is a *conditional* node and $L(curr) = (x_i, \chi)$ **then**
10: $D^+_{curr}(\bar{x}, \bar{y}, C) \leftarrow D^+_{v_1}(\bar{x}, \bar{y}, C), \chi_1(x_i)$
11: **else if** $curr$ is a *parallel* node **then**
12: $D^+_{curr}(\bar{x}, \bar{y}, f_P(C_1, C_2, \varphi)) \leftarrow D^+_{v_1}(\bar{x}, \bar{z}, C_1), D_{v_2}(\bar{z}, \bar{y}, C_2)$
13: **else if** $curr$ is a *loop* node where $L(curr) = (x_i, \chi)$ **then**
14: $T^+_{curr}(\bar{x}, \bar{y}, C) \leftarrow D^+_{v_1}(\bar{x}, \bar{y}, C), \chi_1(x_i)$
15: $T^+_{curr}(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \leftarrow T^+_{curr}(\bar{x}, \bar{z}, C_1), D'_{v_1}(\bar{z}, \bar{y}, C_2), \chi_1(z_i)$
16: $T^+_{curr}(\bar{x}, \bar{y}, f_Q(C_1, C_2, \varphi)) \leftarrow D^*_{v_1}(\bar{x}, \bar{z}, C_1), T^+_{curr}(\bar{z}, \bar{y}, C_2), \chi_1(x_i)$ (where $D^* = D' - D^+$)
17: $D^+_{curr}(\bar{x}, \bar{y}, C) \leftarrow T^+_{curr}(\bar{x}, \bar{y}, C), \chi_2(y_i)$
18: **end if**
19: **end while**

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^+ and D^- of each node are then used to construct: (1) the updated auxiliary data store D' of the node, and (2) the relations D^+ and D^- of the parent node. We first explain the



(a) Cardinality Constraint **(b) Existence Constraint**

Figure 6: Incremental Construction of Auxiliary Data

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 resulted (new) activity node where for each transition $(\bar{x}, \bar{y})$, a tuple $(\bar{x}, \bar{y}, C)$ 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'_u is the updated auxiliary data of node u, and D^+_u and D^-_u are the tuples which are added to or removed from D_u 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 $D^+_{v_1}$ and D_{v_2} is suffice 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 $D^+_{v_1}$ whose input states are in χ (unary relation $\chi_1(x_i)$). 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 $D^+_{v_1}$ whose input states are in χ 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 $(y_i \notin \chi)$ 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 $L(11) = (\alpha_{11}, \{o\}, \{(a, b), (b, c), (c, c)\})$, 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 $Ex(\alpha_3, \alpha_4)$. 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 1 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 $D^-_9 = \emptyset$ 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^+_5 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 $Lb(\alpha)$, all c's in bag C should be greater than or equal to $Lb(\alpha)$. $Ub(\alpha)$ checks c's to be less than or equal to $Ub(\alpha)$, $Rng(\alpha)$ 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 α but not β, so $C[0]$ should be equal to 0. To satisfy $coEx(\alpha, \beta)$, in addition to the previous condition, $C[1]$ has to be 0, means there is no path consisting α but not β. $Res(\alpha, \beta)$ needs all occurrences of α to be followed by a β, 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 $D_5 = \{(a, c, \{0, 1, 1, 2\}), (b, c, \{0, 0\})\}$, there is path with more than on occurrences of α_4, thus $Ub(\alpha_4) = 1$ is not verified. Existence constraint $Ex(\alpha_3, \alpha_4)$ is also not verified because D_5 is $\{(a, c, [1, 0, 3, 0]), (b, c, [0, 0, 0, 2])\}$ and there is path that goes through α_3, but not α_4. □

5 Experimental Evaluations

In this section, several experiments are conducted to evaluate the performance of the DecSerFlow constraints checking approaches. Three main types of algorithms, the Spin-based, the bottom up and the incremental construction, are implemented. The Spin-based algorithm is implemented as explained in Section 2.4 and the other two algorithms are implemented in Java. All algorithms are executed on a computer with 8G RAM and dual 2.9 GHz Intel processors.

The data sets (i.e., process schemas) used in experiments are randomly generated considering the following 5 parameters: number of activities (#A), number of objects (#O), average number of states per object (#S), average number of transitions per activity (#T), and number of loops (#L).

In the experiments we measure the impact of each parameter independently by fixing four of the above parameters and changing the remaining one. To measure the impact of process size (#A) sequential process models with #O=3, #S=10, and #T=3 are considered, and

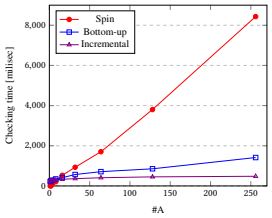


Figure 7: Process size

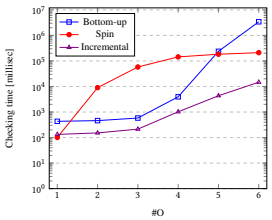


Figure 8: Objects

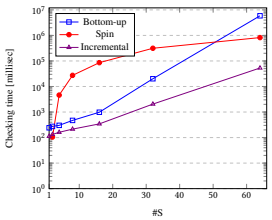


Figure 9: States

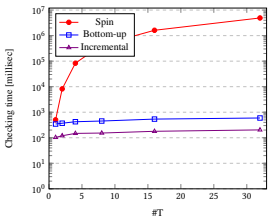


Figure 10: Transitions

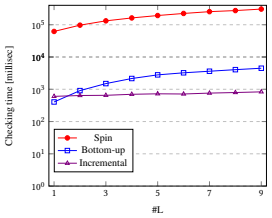


Figure 11: Loops

to perform experiments regarding the last four parameters, process models consisting of all sequential, parallel, conditional, and loop fragments with #A=10, #O=3, #S=10, #T=3, and #L=1 is considered, and each time one of the #O, #S, #T, or #L is changed.

Each experiment records the time needed for an algorithm to check a set of 10 input constraints (2 constraints from each of the 5 classes) on an input schema. Input constraints are chosen in such a way that eight of them are verified, but two of them are not. Note that in the bottom-up construction we collect the time to construct tree and auxiliary data, and check the constraint, whereas in the incremental algorithm we collect the time to update tree, update auxiliary data, retrieve data from and store data into database, and check the constraint after a change operation. In order to collect more accurate results, each experiment is done 9 times, each time with a different change operation (there are totally 9 different change operations) to obtain an average time result with the same parameters. We now report the findings.

Expanding the size of process affects the verification time more in the Spin-based approach than the other approaches

We compute the times needed for the approaches for sequential input schemas with n activities, ($1 \leqslant n \leqslant 256$). Fig. 7 shows the results of all three approaches where although for small processes Spin takes less time, by increasing the process size ($\#A \geqslant 8$), the incremental algorithm shows a better performance. Note that in very small processes ($\#A \leqslant 4$) the incremental algorithm takes more time even more than the bottom-up construction, since it has some overhead to store and retrieve data.

Bottom-up approach has an exponential growth rate in the size of the state space

The state space is affected by two parameters: objects and states. Fig. 8 and Fig. 9 show the results of all three approaches for schemas with $1 \leqslant \#O \leqslant 6$ and $1 \leqslant \#S \leqslant 64$ respectively (in Fig. 8, $\#S = 10$ and in Fig. 9, $\#O = 3$). Notice the logarithmic scale for the y axis, that we use to better highlight the behavior of various algorithms. In comparison to the bottom-up, Spin takes less time to verify the process with large state space, because the bottom-up construction has an exponential grow rate in terms of the number of objects. For example if $\#O = 3$ and $\#S = 10$, there will be totally 10^3 possible state relations and in the worst case 10^6 possible transitions which means to join the auxiliary data of two nodes, 10^{12} comparisons might be needed ($O((|S|^{|O|})^4)$). However, since the incremental approach computes only the updated tuples, its performance is better than both Spin and the bottom-up algorithms.

Spin-based algorithm is more expensive in terms of the number of transitions and loops

Fig. 10 and Fig. 11 show the time needed for all three approaches to verify the schemas with $1 \leqslant \#T \leqslant 32$ and $1 \leqslant \#L \leqslant 10$ 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, Univeristy of California Santa Barbara, for his appreciative contribution and insightful comments provided in the formative stages of this research.

References

[1] A. Awad, G. Decker, and N. Lohmann. 2009. Diagnosing and repairing data anomalies in process models. In *BPM*. Springer, 5–16.

[2] A. Awad, G. Decker, and M. Weske. 2008. Efficient compliance checking using BPMN-Q and temporal logic. In *BPM*. Springer, 326–341.

[3] Ahmed Awad, Matthias Weidlich, and Mathias Weske. 2009. Specification, verification and explanation of violation for data aware compliance rules. In *Service-Oriented Computing*. Springer, 500–515.

[4] K. Batoulis and M. Weske. 2017. Soundness of decision-aware business processes. In Int Conf *on Business Process Management*. Springer, 106–124.

[5] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. 2012. Verification of GSM-based artifact-centric systems through finite abstraction. In *International Conference on Service-Oriented Computing*. Springer, 17–31.

[6] Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, and Dino Mandrioli. 2014. Incremental syntactic-semantic reliability analysis of evolving structured workflows. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 41–55.

[7] Dengler F. Jennings B. Khalaf R. Bruno, G., S. Nurcan, M. Schmidt R. Prilla, M.. Sarini, and R. Silva. 2011. Key challenges for enabling agile BPM with social software. *Journal of Software Maintenance and Evolution: Research and Practice* 23, 4 (2011), 297–326.

[8] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. 1998. Workflow evolution. *Data & Knowledge Engineering* 24, 3 (1998), 211–238.

[9] E. Damaggio, A. Deutsch, R. Hull, and V. Vianu:. 2011. Automatic Verification of Data-Centric Business Processes. In *BPM*. 3–16.

[10] M. De Leoni and W. van der Aalst. 2013. Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In *Business Process Management*. Springer, 113–129.

[11] G. Dong and J. Su. 1998. Arity Bounds in First-Order Incremental Evaluation and Definition of Polynomial Time Database Queries. *J. Comput. Syst. Sci.* 57, 3 (1998), 289–308.

[12] Rik Eshuis and Akhil Kumar. 2016. Converting unstructured into semi-structured process models. *Data & Knowledge Engineering* 101 (2016), 43–61.

[13] Rik Eshuis, Alex Norta, and Raoul Roulaux. 2016. Evolving process views. *Information and Software Technology* 80 (2016), 20–35.

[14] G. Governatori and S. Sadiq. 2009. The journey to business process compliance. In *Handbook of Research on Business Process Modeling*. IGI Global, 426–454.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *SIGMOD*. ACM, 157–166.

[16] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Ario Santoso, and Dmitry Solomakhin. 2013. Verification of Semantically-Enhanced Artifact Systems. In *International Conference on Service-Oriented Computing*. Springer, 600–607.

[17] M. Hashmi, G. Governatori, H. Lam, and M. T. Wynn. 2018. Are we done with business process compliance: state of the art and challenges ahead. *Knowledge and Information Systems* (2018), 1–55.

[18] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[19] R. Hull, J. Su, and R. Vaculin. 2013. Data Management Perspectives on Business Process Management: Tutorial Overview. In *SIGMOD*. ACM, 943–948.

[20] Sh. Ji, B. Li, and D. Qiu. 2016. Incremental Verification of Evolving BPEL-Based Web Composite Service. *Chinese J. of Electronics* 25, 1 (2016), 6–12.

[21] Bartek Kiepuszewski, Arthur Harry Maria Ter Hofstede, and Christoph J Bussler. 2000. On structured workflow modelling. In *International Conference on Advanced Information Systems Engineering*. Springer, 431–445.

[22] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. 2012. Structuring acyclic process models. *Information Systems* 37, 6 (2012), 518–538.

[23] Manfred Reichert and Peter Dadam. 1998. ADEPTflex-Supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems* 10, 2 (1998), 93–129.

[24] M. Reichert and B. Weber. 2012. *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media.

[25] A. Rozinat and W. Van der Aalst. 2008. Conformance checking of processes based on monitoring real behavior. *Information Systems* 33, 1 (2008), 64–95.

[26] O. Sokolsky and S. Smolka. 1994. Incremental model checking in the modal mu-calculus. In *Int. Conf. on Computer Aided Verification*. Springer, 351–363.

[27] E. R. Taghiabadi, V. Gromov, Fahland D., and W. van der Aalst. 2014. Compliance checking of data-aware and resource-aware compliance requirements. In *OTM* Int. Conf. *on the Move to Meaningful Internet Systems*. Springer, 237–257.

[28] W.M.P. van der Aalst and M. Pesic. 2006. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *Proc. of the 3rd Int. Workshop on Web Services and Formal Methods, WS-FM*. 1–23.

[29] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. 2009. The refined process structure tree. *Data & Knowledge Engineering* 68, 9 (2009), 793–818.

[30] B. Weber, M. Reichert, and S. Rinderle-Ma. 2008. Change Patterns and Change Support Features – Enhancing Flexibility in Process-aware Information Systems. *Data & Knowledge Engineering* 66, 3 (2008), 438–466.

[31] B. Weber, M. Reichert, and S. Rinderle-Ma. 2008. Change patterns and change support features–enhancing flexibility in process-aware information systems. *Data & knowledge engineering* 66, 3 (2008), 438–466.

[32] M. Weske. 2012. *Business Process Management: Concepts, Languages, Architectures (2nd Ed.)*. Springer.

[33] M. T. Wynn, H. Verbeek, W. van der Aalst, A. ter Hofstede, and D. Edmond. 2009. Business process verification–finally a reality! *Business Process Management Journal* 15, 1 (2009), 74–92.

[34] G. Yang, M. Dwyer, and G. Rothermel. 2009. Regression model checking. In *IEEE* Int. Conf. *on Software Maintenance, 2009. (ICSM)*. IEEE, 115–124.