Version management for business process schema evolution

Xiaohui Zhao\textsuperscript{a,}\textsuperscript{*}, Chengfei Liu\textsuperscript{b}

\textsuperscript{a} Information Systems Discipline, University of Canberra, Canberra, Australia
\textsuperscript{b} Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia

\textbf{A B S T R A C T}

The current business environment changes rapidly, dictated by user requirements and market opportunities. Organisations are therefore driven to continuously adapt their business processes to new conditions. Thus, management of business process schema evolution, particularly process version control, is in great demand to capture the dynamics of business process schema changes. This paper aims to facilitate version control for business process schema evolution, with an emphasis on version compatibility, co-existence of multiple versions and dynamic version shifts. A multi-level versioning approach is established to specify dependency between business process schema evolutions, and a novel version preserving graph model is proposed to record business process schema evolutions. A set of business process schema updating operations is devised to support the entire set of process change patterns. By maintaining sufficient and necessary schema and version information, our approach provides comprehensive support for navigating process instance executions of different and changing versions, and deriving the process schema of a certain version. A prototype is also implemented for the proof-of-concept purpose.

\textcopyright{} 2013 Elsevier Ltd. All rights reserved.

\section{1. Introduction}

In the current business globalisation setting, varying market opportunities have been described as “change has become the only certainty” [1]. In such a turbulent environment, organisations have to adapt their business processes to emerging opportunities and changes continuously [2,3], in order to survive and thrive. Driven by this urgent demand, organisations are seeking new technologies to help manage their dynamic, expanding and changing business processes [4,5].

Technically, these requirements call for support on process schema evolution, process instance updating, and version control, inter alia. In particular, the capability for updating running process instances to the latest schema version, and modifying the process schema without suspending running instances is highly sought after [6]. In practice, this requirement is further complicated by temporary and concurrent process schema evolutions, since they result in many business process schema versions and their business process instances. As such, innovative version management solutions are in demand to harmonise the co-existence of business processes and instances belonging to different versions.

In our previous work [7], we tackled the handover of running instances from an old process schema to a new schema, i.e., between only two process schema versions, with other colleagues. In this paper we concentrate on management of multiple process schema versions and version shifts. A multi-level evolution diagram is proposed to specify process schema evolutions and their dependencies, while a version preserving graph is defined to record process schema evolutions. This approach allows dynamic process schema evolutions and co-existence of diverse process schema versions, and supports the entire set of classical process change patterns [8,9]. Particularly, our approach contributes to the current process version management by:

1. Proposing a process versioning approach to specify the process schema evolutions and versions in the version space.
(2) Establishing a novel version preserving graph (VPG) model to capture sufficient and necessary process schema evolution information and update the process schema on the fly, in the schema space.

(3) Defining a set of updating operations to support dynamic process schema evolutions following various process change patterns.

(4) Supporting extracting process schemas of different versions and navigating process instances of changing process schema versions in a dynamic manner.

(5) Implementing a proof-of-concept prototype.

The remainder of this paper is organised as follows: Section 2 addresses the version issues in business process schema evolutions with a motivating example, and analyses the requirements for process schema version management with a study of process schema evolution strategies and patterns. Section 3 defines the notion of process schema and introduces a multi-level versioning approach to specify the dependency between process schema evolution and process schema versions. Section 4 proposes a version preserving graph (VPG) model, and explains how to record process schema changes in a version preserving graph with a set of process schema updating operations. Section 5 discusses process instance navigation and process schema retrieval with the VPG model, and introduces the implemented prototype. Section 6 lists the work related to business process instance navigation and process schema retrieval, and discusses the advantages and limitations of our approach. Conclusion remarks and future work are in Section 7.

The reported work in this paper is based on our previous research on process schema evolutions [10], with significant extensions on process schema version designation, process schema evolution analysis, supports to classical process change patterns, model justification and prototype implementation.

2. Motivation and backgrounds

2.1. Motivating example

Business process adaptation and evolution are becoming more common and frequent. Here, we use a manufacturing scenario as a motivating example to illustrate how process schemas evolve. A factory has multiple pipelines of the same type, two of which are pipeline A and pipeline B as shown in Fig. 1. At the beginning, each pipeline follows the same production process schema, which is marked as version 0. The production process includes activities “schedule production”, “produce using work centre #1”, “quality checking” and “packaging”. To increase the production capacity, the factory may add another work centre, e.g., work centre #2, in parallel with work centre #1 to each pipeline. This change upgrades both pipelines A and B from version 0 to version 1. Besides such a permanent process schema evolution, a pipeline may temporarily adjust itself to adapt to its practical situation. For example, the work centre #1 of pipeline A may come across a malfunction, and therefore it has to be removed from the pipeline for maintenance. Yet, pipeline A attempts to keep its production output by fixing defective products during the absence of work centre #1. Therefore, pipeline A will temporarily change from version 1 from version 1.1 as shown in Fig. 1. The numbering of versions will be discussed in Section 3.2.

Fig. 2 sets out the different business process schemas for the evolving pipelines in Business Process Modelling Notations (BPMN) format. The business process schemas for pipelines of versions 0, 1 and 1.1 are shown in Fig. 2(a)–(c), respectively. The other business process schemas in Fig. 2 indicate the business process schemas resulting from further evolutions. For example, pipeline A of version 1.1 may put task “fixing defective products” in a loop to keep fixing the defective products, if they fail to pass the testing. Correspondingly, the process schema will evolve to version 1.1.1 as shown in Fig. 2(d), where a Loop Control gateway is deployed to control when to exit the loop body. For pipeline B of version 0, its work centre #1 may also endure a temporary maintenance, yet pipeline B uses manual labour to replace its work centre #1. In this case, its process schema will evolve from version 0 to version 1.2 as shown in Fig. 2(e). In addition, suppose a thorough technical upgrade to work centre #2 of all pipelines can significantly improve the quality of its manufactured products; therefore the products made by work centre #2 do not require quality checking. Owing to this upgrading, the process schemas for pipelines other than A and B evolve from version 1 to version 2 as shown in Fig. 2(f). For pipeline A, if the upgrading to work centre #2 occurs after pipeline A’s process schema evolves to version 1, then the process schema will evolve again to version 2.1 as shown in Fig. 2(g). If the upgrading to work centre #2 occurs after pipeline A’s process schema evolves to version 1.1, the process schema will evolve to version 2.1.1 as shown in Fig. 2(h). For pipeline B’s process schema of version 1.2, the upgrading to work centre #2 changes it to version 2.2 as shown in Fig. 2(i). Further, when work centre #1 of pipeline B comes back from maintenance, pipeline B’s process schema will revert to version 2 as shown in Fig. 2(f).

Besides such forward evolution, a process schema may possibly change back to a previous version. For example, after pipeline A’s process schema evolves to version 1.1 and before it moves to version 2.1 (i.e., before the technical upgrading of work centre #2), the process schema may change back to version 1 if work centre #1 comes back from maintenance.

This example shows, instead of a simple linear evolution pattern, the actual process schema evolution is subject to many factors, such as unit replacement, technology upgrading or external changes. Some factors, such as unit replacement, external changes, etc., may only result in temporary evolution to a business process schema. Yet, other factors, such as technology upgrades, may bring permanent evolutions to a business process schema. Further, combinations of different cases result in more diverse evolutions to a process schema.

2.2. Process schema evolution strategies and patterns

To systematically describe process evolutions and their influences to a process schema, we first investigate strategies and patterns of process schema evolutions.
In work [11], Ploesser et al. classified process schema evolutions into three strategic types, viz., temporary substitution, temporary adaptation and continuous evolution. The first two strategies cater for process fragment replacement, or the structural adaptation of a business process in response to an anticipated yet temporary event. These two strategies are motivated by emergency response planning, contingency planning, logistics, manufacturing processes, or supporting processes that are subject to temporary disruptions. The third strategy caters for the continuous evolution from the current state of a business process to its future, instigated by a permanent change in the process environment. This is motivated by the observation of continuous improvements in manufacturing environments or shop floor processes. Among these three types of process changes, we consider that a permanent improvement stands out as the main stream of process schema evolution, while a temporary substitution or adaptation serves as a side evolution. Based on the evolution reasons discussed in this work, we have further refined process schema evolutions into revision evolutions and variant ones, which are to be detailed in next section.

At technical level, how a process schema changes itself during an evolution has been investigated by Weber et al. In their work [8,9], they have summarised 14 typical patterns for structural process schema evolutions (named

Fig. 1. Pipeline changes in the production scenario.
process change patterns) and four patterns for the process schema evolutions in predefined regions, based on empirical evidences from large case studies. The former 14 patterns allow users to structurally change process schemas, while the latter four patterns focus on incomplete process modelling. Since the latter four patterns are not in line with the topic of our paper, we only concentrate on the 14 structural change patterns.

Here, we list these 14 change patterns in Table 1. More details can be found in [8].

These 14 patterns are highly abstract process schema change operations. Under certain pre-conditions, such as single-entry-single-exit condition (discussed in Appendix), this set of change patterns is applicable to any part of a business process schema while ensuring the structural correctness of the resulting process schemas [12]. Therefore, these change patterns serve as a cornerstone for process schema evolution management. In Section 4 of this paper, we define a set of process schema updating operations to fully support these change patterns, and thereby justify our model’s capability of handling various process schema evolutions.

2.3. Requirements for process schema version management

Business process schema evolution has been extensively addressed in literature [13–15] from managerial and organisational perspectives, with corresponding methodologies and case studies on process reengineering and improvement. Some enterprise software vendors, such as IDS Scheer AG and Oracle J.D.Edwards, have also investigated business process schema evolutions from a technical perspective, and categorised business process schema evolutions in terms of their motivations and influences [16]. Based on
these academic literature and industry practices, we elicit the following operational features of business process schema evolutions as the base propositions for our process schema evolution and version management.

a) Business process schema evolution is in accordance with the temporary substitution, adaptation and permanent improvement strategies, rather than significant re-designs;

b) All business process schema evolutions can be facilitated with the discussed process change patterns;

c) For a business process, it mainly evolves along a single forward trail driven by permanent improvements; and

d) For a business process, it may have concurrent yet different schema evolutions introduced by temporary substitutions or adaptations. These different evolutions may possibly occur simultaneously with the permanent improvement, and thereby create some side evolution trails, as discussed in the motivating example.

These points feature the dynamic process evolution variations. To initiate the thinking of process version management, we make an analogy to version control in software configuration management. As enterprise software products grow more complex, software products become comprehensive off-the-shelf packages that have to be configured before deployment [17,18]. Following the “Design by Reuse” paradigm, software components are pre-configured into several reference systems as general solutions. Such reference systems will be further configured to adapt to customers’ requirements [19,20]. To facilitate software configuration, software version control accurately records the composition of versioned software components (which are evolving into many revisions and variants), maintains the consistency between interdependent components, reconstructs previously recorded software configurations, etc. [21,22]. From the modelling perspective, a product space and a version space are typically used to store software component details and their inter-relations, as well as the versions of those components and version dependencies respectively.

In comparison to software configuration, a business process schema describes process tasks and the control flows between these tasks. Therefore, instead of software components, these tasks and control flows will be stored in a product/schema space of process change management. The versions of these tasks and control flows and their dependencies will be stored in a version space. Different from software customisation, process schema evolutions are driven by planned improvements (such as technical upgrades to production equipment) or reactions to unpredictable changes (such as machine malfunctions). Thus, process changes are dynamic. In addition, at run time a process schema can be shared by multiple instances, and later on these instances may evolve differently according to their concrete situations, such as the two pipelines in the motivating example. This results in many minor variants on the basis of a common process schema, and the co-existence of running process instances with different versions. Consequently, it requires a finer granularity of change and version description for better reusability.

Based on the above discussion, the following points are summarised as objectives of our proposed process version management methodology:

1. Process schema changes should be recorded in the schema space on the fly;
2. Process schemas of different versions, and process instances of different versioned schemas should be allowed to coexist in the same schema space;
3. Process schema evolutions should be recorded in an efficient manner in the schema space; and
4. A process schema of any version should be retrievable on demand.

To support these points, we propose a multi-level evolution diagram and a version preserving graph to

<table>
<thead>
<tr>
<th>Change pattern name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Insert process fragment</td>
<td>A process fragment is added to a process schema.</td>
</tr>
<tr>
<td>2. Delete process fragment</td>
<td>A process fragment is deleted from a process schema.</td>
</tr>
<tr>
<td>3. Move process fragment</td>
<td>A process fragment is moved from its current position in a process schema to another position within the same schema.</td>
</tr>
<tr>
<td>4. Replace process fragment</td>
<td>A process fragment is replaced by another process fragment in a process schema.</td>
</tr>
<tr>
<td>5. Swap process fragment</td>
<td>Two existing process fragments are swapped in a process schema.</td>
</tr>
<tr>
<td>6. Extract process fragment</td>
<td>From a given process schema a process fragment is extracted and replaced by a corresponding sub process.</td>
</tr>
<tr>
<td>7. Inline subprocess</td>
<td>A sub process to which one or more process schemas refer to is dissolved and the corresponding sub process graph is directly embedded in the parent schemas.</td>
</tr>
<tr>
<td>8. Embed process fragment in loop</td>
<td>Adds a loop construct to a process schema which surrounds an existing process fragment.</td>
</tr>
<tr>
<td>9. Parallelise process fragment</td>
<td>Process fragments which have been confined to be executed in sequence so far are parallelised in a process schema.</td>
</tr>
<tr>
<td>10. Embed process fragment in conditional branch</td>
<td>An existing process fragment shall be only executed if certain conditions are met.</td>
</tr>
<tr>
<td>11. Add control dependency</td>
<td>An additional control edge is added to a process schema.</td>
</tr>
<tr>
<td>12. Remove control dependency</td>
<td>A control edge is removed from a process schema.</td>
</tr>
<tr>
<td>13. Update condition</td>
<td>A transition condition in the process schema is updated.</td>
</tr>
<tr>
<td>14. Copy process fragment</td>
<td>A process fragment is copied from its current position in a process schema to another position of the same schema.</td>
</tr>
</tbody>
</table>

Table 1: Fourteen Structural process change patterns.
record process schema evolutions in schema space and version space, respectively. Based on the above process change patterns, a set of run-time modification operations is developed to support run-time process schema modification. Special model elements are deployed to separate business tasks from process control flow changes, and thereby prevent the interference between different process schema evolutions. We also apply a change-based versioning method to record the differences between versions for enhancing information reusability and storage efficiency. Two extraction strategies are discussed to realise the process schema retrieval from the version preserving graph.

3. Process schema and version representation for business process change management

Business process schema evolution management relies on the recording of process schema evolutions in both schema space and version space, as discussed in Section 2. Correspondingly, this section is to introduce the definition of process schema, process schema evolution types and relations, and a version numbering mechanism, respectively.

3.1. Process schema

Following business process modelling standards, like Event-driven Process Chain (EPC), Business Process Modelling Notation (BPMN) [23] etc., we use gateways to represent the control flow structure between tasks of a business process. The definitions of tasks and gateways are given as follows.

**Definition 3.1.** (Task). A task represents an atomic activity of a business process. To precisely describe the execution order information of task $n_1$, we represent its pre-execution point and post-execution point as $n_1$ and $n_1$, respectively.

As shown in Fig. 3, $n_1$ and $n_1$ are graphically represented as a shadowed diamond and an empty diamond respectively. Each task has its unique pre-execution point and post-execution point. The rationale of using these pre/post-execution points will be discussed in Section 4.4.

**Definition 3.2.** (Gateway). Gateways are dedicated control flow constructs for a business process. Here, we define five types of gateways, namely Xor-Split, Xor-Join, And-Split, And-Join, Loop Control, to represent different control flow structures.

Like a task, each gateway also has its unique pre-execution point and post-execution point, as shown in Fig. 4(a). Here, the post-execution point of an Xor-Split or And-Split gateway has multiple outgoing arcs, which indicate the multiple branches, while its pre-execution point has only one incoming arc. Similarly, the pre-execution point of an Xor-Join or And-Join gateway has multiple incoming arcs, while its post-execution point has only one outgoing arc. For a loop structure, a special gateway, i.e., Loop Control gateway, is deployed to control when to exit the loop body. This gateway also has its own pre/post-execution points. This gateway is not a standard BPMN element, yet it is derived from IBM’s BPMN do-while element supported by its WebSphere Business Modeller 6.1 [24]. In our model, it is used to represent a typical do-while loop. Fig. 4(b)–(d) show samples of Xor-Split/Join, And-Split/Join and Loop structures, where we can see that outgoing arcs from the post-execution point of Loop Control gateway may attach conditions to restrict the corresponding control flow.

**Definition 3.3.** (synchronisation link). A synchronisation link denotes a control dependency between tasks on different parallel branches. For example, in Fig. 4(c), link $l$ is a synchronisation link, which specifies that task $n_2$’s execution is after $n_1$’s execution. Such a synchronisation...
link arises from the dependencies between tasks in an And-Split/Join structure, and it is already used as a standard control flow construct in many business process modelling languages, such as Business Process Execution Language for Web Services (WS-BPEL) [25].

Definition 3.4. (business process schema). A business process represents a collection of related and structured tasks that produce a specific service or product. The schema of a business process \( bp \) can be defined as a directed graph in the form of tuple \((T, G, P, A, s, t)\), where

- \( T \) is the set of tasks;
- \( G \) is the set of gateways;
- \( s \in T \) and \( t \in T \) are the starting point and the terminating point of \( bp \), and thereby \( s \) has no pre-execution point and \( t \) has no post-execution point;
- \( P \) is the set of pre/post-execution points of the tasks/gateways/starting point/terminating point of \( bp \);
- \( A = (\{i\} \cup G) \times P \times (\{s\} \cup G) \cup P \times P \) is the set of arcs, which link above components together. Each arc represents the execution order from its source node to its target node. Set \( A \) also includes synchronisation links in And-Split/Join structures.

Two business process schemas with different forms may be equivalent in structure. To precisely describe the equivalence between two business process schemas, we first introduce the definition of trace which has been adopted by Kinderle-Ma et al. to define the semantics of their proposed process change patterns in [12].

Definition 3.5. (Trace). Let \( P S \) be the set of all process schemas and \( N \) be the set of all nodes (including tasks and gateways but excluding pre/post-execution points) based on which process schema \( P S \) is specified. Let \( \varsigma \) denote the set of all possible traces producible on process schema \( S \). A particular trace \( \delta \in \varsigma \) is then defined as \( \delta = (n_1, n_2, \ldots, n_k) \) where the temporal order of \( n_i \) in \( \delta \) reflects the order in which nodes were completed over \( S \). Further, \( |\delta| \) indicates the number of nodes in trace \( \delta \), and \( \delta(i) \) indicates the \( i \)th item of trace \( \delta \).

Definition 3.6. (process schema equivalence). Two process schemas \( bp_1 \) and \( bp_2 \) are said to be equivalent if the following condition holds.

\[
\forall \delta_1 \in \varsigma(bp_1), \exists \delta_2 \in \varsigma(bp_2) : \delta_1 = \delta_2, \text{ i.e., } \delta_1(i) = \delta_2(i) \text{, where } |\delta_1| = |\delta_2|, \text{ and vice versa.}
\]

Two equivalent process schemas are represented as \( bp_1 \equiv bp_2 \). This relation will be used in defining Evolution Dependency and Evolution Composition in Section 3.2.

3.2. Process schema evolutions

The motivating example in Section 2 shows different process schema evolutions that result in different process schema versions. This section defines the concept of process schema evolution, and specifies the relations between various evolutions.

A process schema evolution denotes an enforcement of a process change pattern on a process schema. According to the process change strategies discussed in Section 2, we classify the process schema evolutions into variant evolutions and revision evolutions:

1. A variant evolution denotes a process schema evolution driven by a temporary substitution or adaptation. Such a variant evolution can change the process schema either from the current version to a new version or from the current version back to a previous version.
2. A revision evolution denotes a process schema evolution driven by a permanent improvement. Such revision evolutions on a process schema cannot be recovered.

Technically, a process schema evolution can be defined as follows.

Definition 3.7. (process schema evolution). A process schema evolution corresponds to an operation \( \varepsilon = (ptn, N_1, N_2, X) \) on a given process schema \( bp \), where

- \( ptn \{ \text{Pattern 1, Pattern 2, ..., Pattern 14} \} \) indicates the change pattern that \( \varepsilon \) will enforce to \( bp \).
- \( N_1 \) and \( N_2 \) are two sets of candidate nodes for applying the change pattern. For example, Pattern 1 “Insert Process Fragment” requires two boundary nodes for the insertion, while Pattern 2 “Delete Process Fragment” only requires the nodes of the fragment for removal.
- \( X \) is a set of node(s) or arc(s) to insert into \( bp \) during the evolution, if needed. For example, Pattern 1 “Insert Process Fragment” can save the fragment to insert in \( X \); while Pattern 11 “Add Control Dependency” can save the synchronisation link in \( X \).
- For process schema \( bp \), its result process schema of evolution \( \varepsilon \) is represented as \( \varepsilon(bp) \).

An evolution \( \varepsilon = (ptn, N_1, N_2, X) \) is said to be enabled for process schema \( bp = (T, G, P, A, s, t) \), if the following two conditions hold. Only an enabled evolution is allowed to be applied to a given process schema.

\[
\begin{align*}
&\text{(1) } \forall n_1 \subseteq bpT \cup bpG \text{ and } \\
&\quad \exists n_2 \subseteq bpT \cup bpG \\
&\text{(2) } \forall n_1 \neq \phi \exists f_1 \in \varsigma(bp) \text{ and } \\
&\quad \exists n_2 \neq \phi \exists f_2 \in \varsigma(bp) \text{ and }
\end{align*}
\]

The Node Existence Condition guarantees process schema \( bp \) contains the tasks and gateways needed by evolution \( \varepsilon \) to apply the underlying process change pattern. The Single-Entry-Single-Exit Condition is needed as it is a pre-condition of process change patterns to guarantee the structural correctness of the resulting process schema as mentioned in Section 2.2. Further, the rationale of this condition is detailed in the Appendix.

These two conditions determine whether an evolution is applicable to a process schema. It becomes more complex when detecting whether more evolutions are applicable to a process schema, as evolutions may interfere with each other and such interferences may result in different degrees of dependencies or conflicts. According to such interferences,
we summarise the following four binary relations between evolutions.

Evolution dependency (≺): Evolution \( e_x \) is said to be dependent on evolution \( e_y \) on process schema \( bp \), represented as \( e_x \preceq e_y \). If \( e_x \) is enabled for \( e_x(bp) \), yet not enabled for \( bp \).

Conflict-free (/): Evolutions \( e_x \) and \( e_y \) are said to be conflict-free, and represented as \( e_x \parallel e_y \). If \( e_x(bp) \) and \( e_y(bp) \) are both enabled by \( e_y \) for \( bp \), yet not enabled for \( bp \).

Evolution conflict (⊥): Two evolutions \( e_x \) and \( e_y \) on process schema \( bp \) are said to be conflicting, and represented as \( e_x \perp e_y \). If \( e_x(bp) \) and \( e_y(bp) \) are both enabled for \( bp \), yet not enabled for \( bp \).

Evolution composition (⊙): A composite evolution \( e_k \) represents the combination of two dependent or conflict-free evolutions, e.g., \( e_i \) and \( e_j \) where \( e_i \not\perp e_j \), and these evolutions can apply together to a process schema. This relation is represented as \( e_k = e_i \circ e_j \). A composite evolution \( e_k \) can be used to represent the result evolution of the composition.

Suppose \( bp \) is a process schema, we have

\[
\begin{align*}
\text{If } e_i \preceq e_j & \quad \text{then } e_k(bp) \equiv e_j(e_i(bp)) \quad \text{Rule (3.1): } e_x \preceq e_y \Rightarrow e_x \perceq e_y \text{ (symmetry of conflicting evolution)} \\
\text{If } e_i \parallel e_j & \quad \text{then } e_k(bp) \equiv e_j(e_i(bp)) \quad \text{Rule (3.2): } e_x \parallel e_y \Rightarrow e_x \parallel e_y \text{ (symmetry of conflict-free evolution)} \\
\end{align*}
\]

These rules are useful for guiding the evolution representation on the Multi-Level Evolution Diagram (MLED) which is introduced in next section. The proof of this theorem is given in the Appendix.

3.3. Business process version designation

For an evolving process schema, a version represents a state of the process schema. During the lifecycle of a business process schema, it may have several versions corresponding to its experienced adaptations and improvements. Some workflow products [26,27] simply use incremental numbers to mark the process schema evolutions, yet lack the support on evolution dependencies and the strategies of enforcing such evolutions. To well represent complex process schema evolutions, a novel versioning approach consisting of a multi-level evolution diagram (MLED) is proposed in this section with a special version numbering mechanism.

3.3.1. Multi-level evolution diagram

Fig. 5 shows an MLED sample, where possible variant evolutions are represented on a series of pages, while revision evolutions are represented as the step from one page to another. According to proposition (c) in Section 2.3, revision evolutions are connected in sequence and construct a single forward trail, which connects all pages from the initial process schema, i.e., \( V_0 \) shown in Fig. 5. \( V_1, V_2, \ldots \) represent the resulting base process schemas of pages \( P_1, P_2, \ldots \) after revision evolutions \( e_b, e_{b_1}, \ldots \) respectively.

On page \( P_1 \), variant evolutions \( e_1 \) and \( e_2 \) represent two different evolutions from base process schema \( V_1 \). These two variant evolutions are drawn as two arrows starting from the central point (delegating base process schema \( V_1 \)). A variant evolution \( e_3 \) depending on \( e_2 \) is represented as an arrow connecting \( e_2 \)’s target point to \( e_3 \)’s target point.

Once a new revision evolution \( e_b \) occurs and leads to a new page \( P_2 \), the variant evolutions on previous pages can be projected to \( P_2 \), if they do not conflict with \( e_b \). For example, variant evolutions \( e_1, e_2 \) and \( e_3 \) will be projected to \( P_2 \) when revision evolution \( e_b \) occurs. Yet, any previous variant evolution \( e_x \) conflicting with the current revision evolution is not allowed to be projected to the new page, neither is any evolution comprising \( e_x \) or depending on \( e_x \), as regulated by Rule (3.3) and Rule (3.4). For example, suppose \( e_3 \) conflicts with next revision evolution \( e_0, e_3 \) will not appear on the next page. If \( e_2 \) conflicts with \( e_x \), neither \( e_x \) nor \( e_2 \) will be projected to next page. As such, on each page there are no
isolated arrows, and therefore each arrow, i.e., a variant
evolution, must lie on a path from the central point.

On each page, conflict-free or dependent variant evolu-
tions may combine together into composite evolutions.
For example, on pages \( P_1 \) and \( P_2 \), dependent evolutions \( e_2 \) and \( e_3 \) compose into \( e_4 \), i.e., \( e_4 = e_2 \oplus e_3 \), which is represented
as an arrow linking the source point of \( e_2 \) to the target point
of \( e_3 \). On page \( P_3 \), conflict-free evolutions \( e_1 \) and \( e_5 \) compose
into \( e_6 \), i.e., \( e_6 = e_1 \oplus e_5 \), which is represented as an arrow
linking the common source point of \( e_1 \) and \( e_5 \) to a new
target point. These composite variant evolutions can be
projected to next page, if they are not conflicting with the
revision evolution, according to Rule (3.5).

Definition 3.8. (evolution path). An evolution path indicates
how a process schema evolves from one version to
another version through a sequence of process schema evolutions.
In an MLED, such an evolution path corresponds
to a distinct path comprising a series of connected arrows.

Note that an MLED does not contain counter-evolutions,
i.e., \( \forall x \in \Xi \forall y \sim bp \equiv x \left( x \left( bp \right) \right) \). A counter-evolution is treated as
a reversion to its previous version, and we do not set an
arrow for it. This guarantees that on each page, the evolution
arrows form a lattice structure without cycles, and the central point serves as the root of the lattice.

3.3.2. Process schema version designation and evolution
identification

According to MLED, we define the following guidelines
for designating versions for different evolutions:

Versioning along the Forward Trail: The process schemas resulted from the revision evolutions along the forward trail
are assigned with incremental and non-negative integer versions. Version 0 means the initial process schema.

Versioning on a Page: The resulting process schema of a
variant evolution is assigned with a unique version by
appending a subdigit to the version number of its ancestor process schema.

This versioning mechanism generates a string of integers separated by dots, e.g., “\( x_1.x_2. \ldots x_n \)”, for each process
schema version. In this string, \( x_1 \) represents a revision
evolution and the others represent variant evolutions. For
example, version 2.4 in the MLED of Fig. 5 indicates the
process schema resulted from revision evolutions \( e_4 \) and \( e_5 \)
as well as variant evolution \( e_6 \). In addition, as \( e_6 \) is a
composite evolution consisting of \( e_1 \) and \( e_5 \), the process
schema of version 2.4 can be further interpreted into a schema resulted from evolutions \( e_4, e_5, e_1 \) and \( e_6 \).

These version numbers can also identify each involved
evolution, as defined below.

For a variant evolution, its ID is the difference between its
result version and its source version; and

For a revision evolution, its ID is the process schema
version of its arriving page.

This identification method provides a unique ID for each
revision. For example, variant evolution \( e_1 \) appears on
both pages \( P_1 \) and \( P_2 \), yet the difference between its result
version and the beginning version on different pages is the
same, i.e., \( 1.2 \rightarrow 0.2 \) on page \( P_1 \) and \( 2.2 \rightarrow 2 \rightarrow 0.2 \) on \( P_2 \).

Table 2 lists the identifications for all evolutions involved
in Fig. 5. In this table, a composite evolution is decomposed
into its constituent evolutions.

4. Version control in business process change
management

The versioning approach discussed in Section 3 solves
the issue of representing process schema versions and
evolutions in the version space. In this section, a novel
graph model is proposed to store and maintain the structural
to changes to a process schema in the schema space.

4.1. Version preserving graph

To graphically describe a business process schema, we
extend the conventional directed graph to represent pro-
cess schemas of different versions. In particular, a version
set, a version mapping and a binary relation are designed
for version preservation. This extended graph is named a
version preserving graph (VPG).
Definition 4.1. (version preserving graph). A VPG stores the process structure of all versions that have occurred for a given process schema \( bp \). Such a VPG can be defined as tuple \( ( N, A, V, f, R, s, t) \), where

- \( N \) is the set of nodes, where each node \( n \in N \) represents a task, a gateway, or its pre/post-execution point;
- \( A \) is the set of arcs, which is defined the same as in the definition of process schema;
- \( V \) is the set of evolution IDs, such as “1”, “2”, “0.1”, “0.0.1”, etc.;
- \( f: N \cup A \rightarrow V \) is a mapping that assigns each node or arc in the graph with a proper version number;
- \( R \) is a binary relation, i.e., \( R = \{(a_1, a_2)\mid a_1, a_2 \in A \land a_1 \text{ and } a_2 \text{ in an exclusive relation}\} \). With this binary relation, exclusive relations between arcs in a VPG can be easily represented. Note, here \( R \) only records the exclusive relations that are caused by versioning, not by business constraints.

<table>
<thead>
<tr>
<th>Evolution name</th>
<th>Corresponding evolution identification</th>
<th>Corresponding process schema shift in Fig. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 )</td>
<td>0.2</td>
<td>Fig. 2(b) → Figure 2(e)</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>0.1</td>
<td>Fig. 2(b) → Fig. 2(c)</td>
</tr>
<tr>
<td>( e_3 )</td>
<td>0.01</td>
<td>Fig. 2(f) → Fig. 2(g)</td>
</tr>
<tr>
<td>( e_4 = e_2 \oplus e_3 )</td>
<td>0.11 (( = 0.1 \oplus 0.01 ))</td>
<td>Fig. 2(b) → Fig. 2(d)</td>
</tr>
<tr>
<td>( e_5 )</td>
<td>0.3</td>
<td>Fig. 2(f) → Fig. 2(h)</td>
</tr>
<tr>
<td>( e_6 = e_1 \oplus e_5 )</td>
<td>0.4 (( = 0.2 \oplus 0.3 ))</td>
<td>N/A</td>
</tr>
<tr>
<td>( e_8 )</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>( e_9 )</td>
<td>3</td>
<td>N/A</td>
</tr>
</tbody>
</table>

In summary, this graph represents a business process schema with nodes and arcs, keeps their evolution identifications in mapping \( f \) and stores exclusive relations between arcs using relation \( R \). Exclusive relations, version numbers and pre/post-execution points are the main differences of a VPG from other process models. Version numbers are already discussed in Section 3.2, while the transitivity of exclusive relations and the rationale of pre/post-execution points will be addressed in Sections 4.3 and 4.4, respectively.

Fig. 6 shows the VPG example according to the motivating example in Section 2. In this graph, each node represents a task, gateway, pre/post-execution point, or a starting/terminating point, and the versions of nodes and arcs are marked aside as labels. The small curves crossing arcs denote the exclusive relation among arcs.

This graph contains the information of three process schemas, viz., schemas of versions 0, 1 and 1.1. This information enables the VPG to navigate the execution of process instances belonging to these versions. For example, a process schema of version 1.1 can be regarded as resulting from evolutions 1 and 0.1. In this VPG, an instance of process schema version 1.1 first goes through starting point \( s \), and node \( n_1 \) which has evolution ID 0. From node \( n_1 \), the instance bypasses arc \( o_0 \) and flows along arc \( a_1 \), as the ID (1) of \( a_1 \) is higher than the ID (0) of arc \( o_0 \) which is in the exclusive relation with \( a_1 \). After passing And-Split gateway \( a_{g_1} \), the process instance splits into two branches, where one branch flows into node \( n_5 \), and the other flows to \( n_2 \) via arc \( o_1 \). Similarly, the second flow goes forward along arc \( b_1 \) (which has evolution ID 0.1) but not arc \( o_1 \) (which has evolution ID 0), since \( b_1 \)'s evolution ID has a higher priority than \( o_1 \)'s. As such, the execution of process instance with version 1.1 can be easily navigated in this graph, and so are process instances of version 0 or 1. The details on how to set up the evolution IDs

![Fig. 6. An example of VPG.](image-url)
of arcs and nodes, and the priority of selecting arcs will be discussed in Section 4.3 and Section 5 respectively.

4.2. Process schema updating Operations

To support complex process version shifts in practical scenarios well, we claim that run-time process schema updating (during the process schema evolution) should meet the following requirements:

- **Dynamic**: Process schema version shifts may occur at any time, and therefore process schema updating should be operable at both build time and run time.
- **Information preservable**: Process schema of old versions are needed for auditing the execution of process instances (against the process schema version at that time). Thus, the process schema information of old versions should be preserved during schema modification.
- **Restorable**: Process instances may sometimes be changed back to a previous version, particularly in cases of temporary substitution and temporary replacement. Consequently, the process schema updating should be restorable.

According to the process change patterns discussed in Section 2.2, we provide details on implementing these patterns within the context of a version preserving graph (VPG) in Tables 3–6. The procedure for each process change pattern is illustrated by an example graph, where a region bordered by dashed lines denotes a process fragment, and the newly inserted arcs and nodes are assigned evolution ID vid. An inclined curve between two arcs stands for an exclusive relation. Below each example graph, the corresponding set operations to a VPG are also given.

For pattern 1 “insert process fragment”, the case of “parallel insertion” can be easily adjusted to support the case of “conditional insertion” by replacing And-Split/Join gateways, ag1 and ag2, with Xor-Split/Join gateways.

For pattern 2 “delete process fragment” and pattern 4 “replace process fragment”, the to-be-changed process fragment is not removed from the graph, but is “short circuited” by a new arc having a new version. By this means, the original process fragment can be preserved for process instances with old versions.

For pattern 3 “move process fragment”, the case of “parallel insertion” can be easily adjusted to support the case of “conditional insertion” by replacing And-Split/Join gateways, ag1 and ag2, with Xor-Split/Join gateways. For

### Table 3

Process schema updating operations according to process change patterns 1, 2 and 4.

<table>
<thead>
<tr>
<th>1. Insert process fragment</th>
<th>2. Delete process fragment</th>
<th>4. Replace process fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequential insertion</strong></td>
<td><strong>Parallel insertion</strong></td>
<td></td>
</tr>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
<td><img src="image9" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image10" alt="Diagram" /></td>
<td><img src="image11" alt="Diagram" /></td>
<td><img src="image12" alt="Diagram" /></td>
</tr>
</tbody>
</table>

N=V∪{m1, m2, m3, ..., m6};
create arcs b1=(m1, m2);
create arcs b2=(m3, m6);
A=A∪{b1, b2, c1, ..., c3};
V=V∪{vid1};
f=f∪{(b1, vid1), (b2, vid1), (c1, vid1), (c2, vid1)};
f=f∪{(m1, vid1), (m2, vid1)};
R=R∪{(a0, b1)}.

N=V∪{m1, m2, m3, ..., m6};
create arcs b1=(m1, m2);
create arcs b2=(m3, m6);
A=A∪{b1, b2, c1, ..., c3};
V=V∪{vid1};
f=f∪{(b1, vid1), (b2, vid1), (c1, vid1), (c2, vid1)};
f=f∪{(m1, vid1), (m2, vid1)};
R=R∪{(a0, b1)}.

N=V∪{m1, m2, m3, ..., m6};
create arcs b1=(m1, m2);
create arcs b2=(m3, m6);
A=A∪{b1, b2, c1, ..., c3};
V=V∪{vid1};
f=f∪{(b1, vid1), (b2, vid1), (c1, vid1), (c2, vid1)};
f=f∪{(m1, vid1), (m2, vid1)};
R=R∪{(a0, b1)}.
pattern 3 “move process fragment” and pattern 5 “swap process fragment”, the movement or swapping of involved process fragment(s) is realised by adjusting the execution dependencies by adding extra arcs and setting up exclusive relations. In this way, we can support these patterns with minimal graph changes.

For pattern 8 “embed process fragment in loop” and pattern 10 “embed process fragment in conditional branch”, proper gateways will be inserted to represent the required loop or branching structure. The designated evolution IDs to related arcs can navigate the execution of process instances with different versions.

For pattern 12 “remove synchronisation link”, a new arc is inserted to “short circuit” both the to-remove arc, i.e., $a_0$ in the example, and the to-keep arc, i.e., $a_1$. Arc $a_1$ cannot serve two evolutions, as an arc can only carry one evolution ID. Owing to this characteristic, we combine pattern 11 and pattern 12 together to support a composite pattern “replace synchronisation link”, which simply adds a new arc in an exclusive relation to the old arc. For pattern “update condition”, the old arc is replaced with a new one which is attached with the new condition.

4.3. Updating VPG

When a process schema evolution occurs, we need to update the VPG using the operations discussed in the previous section. This procedure is detailed in this section. The version numbering mechanism has been discussed in Section 3.2. Exclusive relations play an important role in distinguishing the outgoing paths from a node. In particular, when updating a VPG, the exclusive relation(s) introduced by a new evolution may interfere with the exclusive relations between existing arcs, owing to the transitivity of exclusive relation. Thus, we investigate the transitivity of this relation first.

Transitivity of exclusive relation: For two arcs, if each of them is in an exclusive relation with a third arc, these two arcs are also in an exclusive relation. Given exclusive relation $R$, we can formalise this property as $(a_1Rb_1) \land (a_1Rc_1) \Rightarrow b_1 Rc_1$.

4.3.1. Explanation

The exclusive relation between two arcs implies the intention of replacing the original control flow and therefore arcs $b_1$ and $c_1$ represent two replacing schemes on $a_1$. 

---

| Process schema updating operations according to process change patterns 3 and 5. |
|---------------------------------|---------------------------------|
| Sequential movement | Parallel movement |
| Sequential movement | Parallel movement |

Table 4

3. Move process fragment

<table>
<thead>
<tr>
<th>Sequential movement</th>
<th>Parallel movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1=(p_1, \bullet p_2)$, $b_2=(n_1, \bullet p_3)$, $b_3=(p_1, \bullet n_1)$; $A=\text{Add}(b_1, b_2, b_3)$; $\text{transit}(b_1, b_2)$; $\text{transit}(b_2, b_3)$; $f=\text{transit}((b_1, \text{vid}), (b_2, \text{vid}), (b_3, \text{vid}))$; $R=\text{transit}((a_1, b_1), (a_2, b_2), (a_3, b_3))$.</td>
<td></td>
</tr>
<tr>
<td>$b_1=(p_1, \bullet p_2)$, $b_2=(n_1, \bullet p_3)$, $b_3=(p_1, \bullet n_1)$; $A=\text{Add}(b_1, b_2, b_3)$; $\text{transit}(b_1, b_2)$; $\text{transit}(b_2, b_3)$; $f=\text{transit}((b_1, \text{vid}), (b_2, \text{vid}), (b_3, \text{vid}))$; $R=\text{transit}((a_1, b_1), (a_2, b_2), (a_3, b_3))$.</td>
<td></td>
</tr>
</tbody>
</table>

4.3. Updating VPG

When a process schema evolution occurs, we need to update the VPG using the operations discussed in the previous section. This procedure is detailed in this section. The version numbering mechanism has been discussed in Section 3.2. Exclusive relations play an important role in distinguishing the outgoing paths from a node. In particular, when updating a VPG, the exclusive relation(s) introduced by a new evolution may interfere with the exclusive relations between existing arcs, owing to the transitivity of exclusive relation. Thus, we investigate the transitivity of this relation first.

Transitivity of exclusive relation: For two arcs, if each of them is in an exclusive relation with a third arc, these two arcs are also in an exclusive relation. Given exclusive relation $R$, we can formalise this property as $(a_1Rb_1) \land (a_1Rc_1) \Rightarrow b_1 Rc_1$. 

4.3.1. Explanation

The exclusive relation between two arcs implies the intention of replacing the original control flow and therefore arcs $b_1$ and $c_1$ represent two replacing schemes on $a_1$. 

As these two replacing schemes cannot be applied simultaneously, $b_1$ and $c_1$ are inherently in an exclusive relation.

Based on this transitivity of exclusive relation, we introduce how to update the VPG to reflect dynamic process schema evolutions. Basically, the VPG updating procedure comprises steps of inserting new arcs and nodes, assigning proper evolution IDs and setting exclusive relations, as described as follows.

Procedure of updating new evolution $\varepsilon_5$ into the corresponding VPG.

1. **Assign evolution ID vid to all to-be-added nodes and arcs belonging to evolution $\varepsilon_5$, according to the versioning method introduced in Section 3.3.2.**
2. **Insert the to-be-added nodes and arcs of $\varepsilon_5$ to the VPG.**
3. **For every arc $a_i=(n_1, n_2)\in\mathcal{A}$ where $\mathcal{A}$ is the set of the to-be-added arcs of $\varepsilon_5$.**
   - If node $n_1$ already has an existing outgoing arc $a_2$ with evolution ID $uid$ in the VPG where $\varepsilon_5\leq\varepsilon_5$ and evolution $\varepsilon_5$ has ID $uid$, then
     - Set arcs $a_1$ and $a_2$ in an exclusive relation;
     - Update the derived exclusive relations with other arcs, according to the Transitivity of Exclusive Relation.

This updating process first assigns the evolution IDs to the new nodes and arcs, then inserts the new nodes to the VPG and in the end iteratively checks the new arcs to set the exclusive relations. Using the production process discussed in Section 2 as an example, Fig. 7 illustrates how this process schema is updated during a series of evolutions. The initial process schema is represented as the VPG shown in Fig. 7(a), where all nodes and arcs are marked with evolution ID 0. When the business process evolves to version 1 as an additional work centre is inserted, the VPG will be updated to Fig. 7(b) following process change pattern “parallel insert process fragment”. The inserted arcs $a_1, \ldots, a_{13}$, node $n_5$, two And-Split/Join gateways $ag_1$ and $ag_2$, and their pre/post-execution points are all assigned with evolution ID 1. Later on, when the business process for pipeline $A$ evolves to version 1.1, as task “produce with work centre #1” is replaced by task “fix defective products”, the VPG will be updated to Fig. 7(c) following pattern “replace process fragment”. The added arcs $b_1, \ldots, b_4$, node $n_7$ and its pre/post-execution points are assigned with evolution ID 0.1. Again, when pipeline $A$ puts task “fix defective products” in a loop, a loop control, its pre/post-execution points and related arcs are inserted with evolution ID 0.0.1, as shown in Fig. 7(d). In the meantime, the business process for another pipeline may replace task “produce work centre #1” with “manual production”, and thereby evolves to version 1.2 as shown in Fig. 7(e). The added arcs $c_1, \ldots, c_4$, node $n_8$ and its pre/post-execution points are assigned with evolution ID 0.2.
Table 6
Process schema updating operations according to process change patterns 11 to 13.

|------------------------------|---------------------------------|------------------------------------|---------------------|
| ![Diagram 1](image1)  
Create sync link \( a_1 = (n_1, n_2); \)  
\( A = A \cup \{a_1\}; \)  
\( V = V \cup \{vid\}; \)  
\( f = f_1(\{a_1, vid\}); \)  
\( R = R_0(\{a_2, a_1\}); \) | ![Diagram 2](image2)  
Create arc \( a_2 = (n_1, n_3); \)  
\( A = A \cup \{a_2\}; \)  
\( V = V \cup \{vid\}; \)  
\( f = f_1(\{a_2, vid\}); \)  
\( R = R_0(\{a_2, a_1\}); \) | ![Diagram 3](image3)  
Create sync link \( a_1 = (n_1, n_3); \)  
\( A = A \cup \{a_1\}; \)  
\( V = V \cup \{vid\}; \)  
\( f = f_1(\{a_1, vid\}); \)  
\( R = R_0(\{a_2, a_1\}); \) | ![Diagram 4](image4)  
Create arc \( a_1 = (xog, n_3); \)  
\( A = A \cup \{a_1\}; \)  
\( V = V \cup \{vid\}; \)  
\( f = f_1(\{a_1, vid\}); \)  
\( R = R_0(\{a_2, a_1\}); \) |
Fig. 7. VPG sample.
According to the transitivity of exclusive relation, arc \( c_1 \) is set in an exclusive relation with \( b_1 \), next to \( a_1 \). When handling the evolution to version 2, arcs \( a_1 \), \( a_2 \), and \( a_3 \) with evolution ID 2 are added following pattern "sequential move process fragment" as shown in Fig. 7(f).

4.4. Theoretical discussion on VPG model

Pre/post-execution points are the major distinction of our VPG model to other process modelling approaches. Here, we discuss about the motivation for deploying these pre/post-execution points.

4.4.1. Rationale for deploying pre/post-execution points

As an information-preserving model, our VPG model changes a business process schema only by adjusting its control flows. In this model, pre/post-execution points only act as placeholders without business meanings. However, these pre/post-execution points take over the control flow modification, and thereby separate tasks and gateways from control flow changes. This separation prevents potential inter-influences between VPG updating and historical control flow keeping.

Take the VPG in Fig. 8(a) as an example: task \( n_2 \) is to replace task \( n_1 \), which has two incoming arcs, \( a_1 \) and \( a_2 \), to its pre-execution point \( n_1 \), and two outgoing arcs, \( a_3 \) and \( a_4 \), from its post-execution point \( n_1 \). At node \( n_1 \), arc \( b_1 \) overrides arc \( a_2 \), and thereby redirects the routing to \( n_2 \) while bypassing the routing to \( n_1 \). The new path now becomes \( n_1 \rightarrow n_2 \rightarrow n_2 \rightarrow n_1 \), while \( n_1 \) and \( n_2 \) on this new path naturally inherit their original control flows, i.e., arcs \( a_1, a_2, a_3 \) and \( a_4 \) without redirecting these arcs to \( n_2 \). Note, here \( n_2 \) has its own pre/post-execution points instead of using \( n_1 \)'s. These points can help preserve the control flows related to \( n_2 \) during future modifications to \( n_2 \).

Fig. 8(b) shows an example of removing a task. In a VPG, task removal is achieved by adding a new arc to short circuit the task. As shown in Fig. 8(b), arc \( b_1 \) links \( a_1 \) to \( n_1 \), and creates a new path \( n_1 \rightarrow n_1 \). On this new path, \( n_1 \) and \( n_1 \) keep the original control flows, i.e., arcs \( a_1, a_2, a_3 \) and \( a_4 \). Without such pre/post-execution points, these control flows have to connect to \( n_1 \) itself, and will get lost when removing \( n_1 \).

From these examples, we can see that pre/post-execution points play an important role in keeping historical control flow information. Any removal or replacement of task/gateway will not eliminate these pre/post-execution points, and therefore related historical control flow information can be inherited.

In addition, both pre- and post-execution points are necessary to separate a node from the interference caused by control flow changes to its neighbouring nodes. Therefore, these two types of points cannot be combined into one type. Fig. 8(c) and (d) illustrate a scenario in which such single-type points mess up control flow changes of neighbouring nodes. Suppose we intend to apply the sequential movement operation (the first change pattern in Table 4) to this process schema, and change the execution order from \( n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4 \) to \( n_1 \rightarrow n_3 \rightarrow n_2 \rightarrow n_4 \). To realise this change, arcs \( x, y, z \) together with related exclusive relations are inserted to link \( n_1 \) to \( n_3 \), \( n_2 \) to \( n_4 \), and \( n_3 \) to \( n_2 \), respectively, as shown in Fig. 8 (d). This finally results in a loop between the point after \( n_1 \) and the point before \( n_4 \), owing to the interference between arcs \( x, y, z \). In comparison, our solution for the sequential movement operation (please refer to Table 4) avoids such interferences by confining control flow changes to the pre/post-execution points of the object node without touching the ones of neighbouring nodes. Note, as pre-/post-execution points are affected by interferences instead of object nodes, some pre-/post-execution points may be left over after some node removal modification.

As a compact model, a VPG is expected to record all version information with minimal model space. In this dilemma, information sufficiency and necessity are two preferred properties for our approach.


Property 2. (Information necessity). A VPG contains no redundant information to construct a process schema of a particular version in the multi-level evolution diagram.

4.4.2. Justification

The relationship between process schema evolutions and the resulting structural changes to a process schema are recorded by MLED and VPG, respectively. Each version in the MLED corresponds to a series of arcs and nodes recorded in the VPG, and vice versa. Each path in the MLED indicates an evolution scheme from the base version, and by collecting the IDs of these evolutions we can derive a process schema of a given corresponding version from the VPG. In this context, Property 1 can be interpreted as requiring an evolution path from the initial version of the process schema to the target one in the MLED. As discussed in Section 3.3.1, pages are connected by a unidirectional forward trail, and on each page the variant evolutions form a lattice structure with the base process schema as the root. In this structure, each schema version is connected from the base version of the page by at least one path, and therefore Property 1 holds.

Property 2 implies that any two evolution paths from the same starting version to different ending versions generate different process schemas. On each page, this can be further interpreted by two evolution paths ending at different

---

**Fig. 8.** Examples for pre/post-execution points.
versions generating different process schemas. Since this corresponds to a composite variant evolution on a page, we focus on how composite evolutions are represented in the MLED. As discussed in Section 3.3.1, if a composite evolution comprises two conflict-free evolutions, these three evolutions are represented by three individual arrows with a common starting version but different target versions. In this situation, the three arrows construct three different paths, and each of them generates a different process schema. If a composite evolution $e_p = e_1 \otimes e_2$ and $e_1 \otimes e_2$, then variant evolution $e_1$ and $e_2$ are represented as two connected arrows while $e_p$ is represented as an arrow linking $e_1\text{s}$ source version to $e_2\text{s}$ target version. In this way, these three evolutions construct two paths, yet they share the same starting version and ending version in the MLED. Thus, Property 2 holds.

These two properties guarantee that a VPG provides necessary and sufficient information to derive a process schema of any version that has occurred. Based on this foundation, the next section discusses how to use a VPG to navigate process instances of different and changing versions, and extract a process schema of a given version at run time.

5. Run-time version management

Run-time process schema version management tackles three main issues, i.e., process instance navigation, process schema retrieval and process instance data migration. Our previous work [7] has already addressed the last issue and this paper concentrates on the first two issues with the proposed process version control approach. With respect to process instance navigation and process schema retrieval, exclusive relation and evolution IDs play an important role in determining process components and routing for process instances/schemas of different versions. In particular, when process instance navigation or process schema retrieval runs to a node with several outgoing arcs of different evolution IDs in an exclusive relation, the selection of proper outgoing arc(s) follows the priority list defined below.

**Evolution ID selection priority:** For a given version in the format of $x_1.x_2...x_n$, the selection priority between related evolution IDs is given in a descending order:

1. If process schema evolution with ID $0.0...x_k$ is a composite evolution, then find evolution IDs in $\text{decomposedVIDs}(0.0...x_k)$, otherwise find evolution ID $0.0...x_k$;
2. If process schema evolution with ID $0.0...x_{k+1}$ is a composite evolution, then find versions in $\text{decomposedVIDs}(0.0...x_{k+1})$;
   otherwise find evolution ID $0.0...x_{k+1}$;
3. ... …
4. If process schema evolution with ID $x_1.x_2$ is a composite evolution, then find evolution ID in the form of $x_1$;
5. Evolution ID in the form of $x_1$;
6. Evolution ID in the form of $x_{k+1}$;
7. ... …
8. Evolution ID 0.

The versions not on this list will not be considered, while function $\text{decomposedVIDs}(vid)$ returns the set of evolution IDs of constituent evolutions belonging to composite evolution with ID $vid$. Basically, this priority list follows the reverse order of evolution appearances in possible evolution paths in MLED. The later an evolution appears the higher priority it has.

5.1. Process instance navigation

Process instance navigation is a typical application of run-time version management, which is responsible for directing a process instance of a certain version to flow through the VPG via proper routing and adjusting its routing when the instance shifts its version.

Within a VPG, once a process instance of version $v$ is initialised, it starts from starting point $s$, and then flows to other nodes by following the Evolution ID Selection Priority List to select proper outgoing arcs, if these arcs are not involved in any exclusive relation. When it comes to multiple outgoing arcs in an exclusive relation, the instance will choose the arc with the highest prioritised evolution ID. For example, in the VPG shown in Fig. 9, the routing for a process instance of version 2.2 is as follows:

1. Flow from $s$ to node $n_1$ via the intermediate pre/post-execution points;
2. After $n_1\bullet$, the instance flows to $\bullet ag_1$, via arc $c_1$, as arc $a_1$'s evolution ID holds the higher priority compared to $o_1s$;
3. After $ag_1\bullet$, the instance splits into two branches, one of which goes to $n_2$ and the other goes to $n_3$;
4. After $\bullet n_2$, the first branch flows to arc $c_1$, as $c_1$'s evolution ID has the highest priority compared to the ones of $b_1$ and $o_1$;
5. This branch goes through $n_6$ to $n_2\bullet$, and afterwards it chooses arc $d_2$ to get to $n_3$, and then changes to arc $d_2$ after $n_3\bullet$;
6. The second branch passes node $n_5$ and converges with the first branch at $\bullet ag_2$; and
7. The instance proceeds to $ag_2\bullet$, and goes forwards along arc $d_1$ to $n_4$ and then ends at the terminating point $t$.

If an instance changes its version during its execution, the navigation will also change to use the new version to select proper outgoing arc(s). Exclusive relationships between arcs and the evolution IDs of arcs and nodes together provide the semantic information for navigating process instances.

5.2. Process schema retrieval

A VPG maintains the information of all versions of a business process schema. In practice, many business process management operations only refer to the business process schema of a specific version, e.g., the operations of initiating instances, reviewing business processes. Accordingly, the VPG is expected to dynamically extract the business process schema of a given version. Basically, such schema extraction can be achieved with two different strategies, backward assembling and top-down exploration. The next two subsections discuss these two strategies, respectively.
backward assembling strategy

Given a requested version, the most direct strategy is to start by collecting the nodes and arcs having evolution IDs with the highest priority and continue to lower ones, according to the Evolution ID Selection Priority List discussed in Section 5. In this strategy, before collecting nodes and arcs of the next highest priority, we need to delete all the arcs that are in an exclusive relation with any collected arc. This removal may result in some unreachable nodes, i.e., nodes with no incoming arcs, or broken Split/Join, Loop structures. These unreachable nodes or broken structures need to be deleted as well as those arcs connecting to these nodes.

This collecting and removing process continues until all arcs or nodes with the versions on the priority list are handled. For example, suppose we extract a process schema of version 1.1 from the VPG shown in Fig. 7(c). According to the priority list, ID 0.1 has the highest priority, ID 1 has the second, and ID 0 has the least. Following a priority descending order, we start with evolution ID 0.1, and therefore arcs $b_1, ..., b_4$, node $n_7$, and its pre- and post-execution points are first selected. As arc $b_1$ is in an exclusive relation with arc $a_1$, arc $a_1$ needs to be removed. This removal results in node $n_2$ having no incoming arcs, and therefore node $n_2$ should be removed, together with its outgoing arc. After this, the evolution ID of the highest priority drops to 1, and therefore arcs $a_1, ..., a_{12}$, nodes $ag_1, ag_2, n_5$, and their pre/post-execution points are collected. In the meantime, arc $a_2$ is removed as it is in an exclusive relation with arc $a_2$. Then, the evolution ID of the highest priority drops to 0, and nodes $s$, $t$, $n_1$, $n_3$, and $n_4$ as well as their pre/post-execution points are collected, together with the related arcs. Now, the assembling finishes, and the collected nodes and arcs construct the structure of the process schema of version 1.1. Algorithm 1 formalises the extraction procedure using this strategy.

In this algorithm, the following functions are used.

- Function $\text{relatedArcs}(M, A\text{Temp})$, referenced in line 3 in Algorithm 1, returns the set of arcs that are in an exclusive relation with arcs in set $A\text{Temp}$ in VPG $M$;
- Function $\text{in}(M, n)$, referenced in line 7, returns the in-degree of node $n$ in VPG $M$;
- Function $\text{outArcs}(M, n)$, referenced in line 8, returns a set of node $n$'s outgoing arcs in VPG $M$;
- Function $\text{linkedNode}(M, a)$, referenced in line 6, returns the node that arc $a$ links to in VPG $M$;
- Function $\text{highestVer}(M, v)$, referenced in line 17, returns the evolution ID with the currently highest priority in VPG $M$ with version $v$;
- Function $\text{tasks}(N\text{Target})$, referenced in line 18, returns all tasks from node set $N\text{Target}$;
- Function $\text{points}(N\text{Target})$, referenced in line 18, returns all pre/post-execution points from node set $N\text{Target}$;
- Function $\text{gateways}(N\text{Target})$, referenced in line 18, returns all gateways from node set $N\text{Target}$.

**Algorithm 1.** Backward assembling

| Input | $M$—The VPG for a business process schema  
| Output | $v$—The requested version |
| 1. | $A\text{Temp}=\emptyset$; $N\text{Target}=\emptyset$; |
| 2. | add the nodes and arcs of version $\text{highestVer}(M, v)$ to sets $N\text{Temp}$ and $A\text{Temp}$, respectively; |
| 3. | $A=\text{relatedArcs}(M, A\text{Temp})$; /*$N\text{Temp}$ and $A\text{Temp}$ keep the nodes and arcs of the current latest version in a temporary VPGM. $A$ stores the arcs that are in the exclusive relation with the ones in $A\text{Temp}$. */ |
| 4. | while $(A\neq\emptyset)$ do |
| 5. | pick arc $a\in A$; /*choose node $n$ that arc $a$ links to in temporary VPG $M$*/ |
| 6. | $n=\text{linkedNode}(M, a)$; |
| 7. | if $\text{in}(M, n)=1$ then |
| 8. | $A=A-\text{outArcs}(M, n)$; /*if $n$ only has one incoming arc, i.e., arc $a$, in temporary VPG $M$, this */ |
| 9. | delete $n$ from $M$; /*$n$ is inaccessible to the process schema of version $v$, and therefore*/ |
| 10. | end if |
| 11. | $\text{VPG M. The exploration shall continue by spreading though $n$'s outgoing arcs, and therefore these arcs are added to $A^*$} |
| 12. | delete $a$ from $A$ and $M$; |
| 13. | $N\text{Target}=N\text{Target}\cup N\text{Temp}$; |
Algorithm 2. Top-down exploration.

\[
\text{Input M—The VPG for a business process schema} \\
\text{v—The requested version} \\
\text{Output Q—The graph for the business process schema of the requested version} \\
1. NTarget = ∅; ATarget = ∅; \quad \text{\texttt{\# NTarget and ATarget keep the reserved nodes and arcs.}} \\
2. NTemp = \{M\}; \quad \text{\texttt{\# NTemp keeps the set of nodes for searching. First, push starting node } s \text{ in it.}} \\
3. while (NTemp ≠ ∅) do \\
4. for each \( n \in \text{NTemp} \) do \\
5. \( A = \text{coupledArcs}(M, n); \) \\
6. \( \text{ATemp} = \text{outArcs}(M, n) - A; \) \\
7. \( \text{\# For node } n, \text{ put its outgoing arcs that are free of exclusive relation into ATemp.} \) \\
8. \( \text{pick arc } a \in A; \) \\
9. \( A = \text{correlatedArcs}(M, a, s(a)); \) \\
10. \( \text{ATemp} = \text{ATemp} \cup \text{pickPriorityArc}(A, v); \) \\
11. \( A = A - A; \) \\
12. end while \\
13. NTarget = NTarget ∪ \{n\}; \\
14. end for \\
15. NTemp = ∅; \\
16. for each \( a \in \text{ATemp} \) do \\
17. \( \text{NTemp} = \text{NTemp} \cup \text{checkNodes}(M, a) - \text{NTarget}; \) \\
18. end for \\
19. ATarget = ATarget ∪ \text{ATemp}; \\
20. end while \\
21. Q = \{\text{tasks}(\text{NTarget}), \text{points}(\text{NTarget}), \text{gateways}(\text{NTarget}), \} \\
\text{ATarget, Ms, Mt};
\]

This algorithm uses sets NTemp and ATemp to keep the newly collected nodes and arcs respectively. Set A temporarily stores the arcs to be deleted from the VPG. After picking an arc \( a \) in set A, the algorithm will check whether \( a \) is the only incoming arc to its linked node \( n \). If so, node \( n \) will be deleted along with \( a \), from the VPG, and the outgoing arcs of \( n \) will be added to set \( A \) for future checking. The collected nodes and arcs will be inserted into the resulting graph by moving the elements in NTemp and ATemp to NTarget and ATarget, respectively. VPG M is a temporary VPG. The iterative checking is conducted following the descending version order, i.e., from the nodes and arcs of the requested version progressively to the ones of the lowest version (version 0). During each checking, some inaccessible nodes and arcs to the process schema of the given version will be removed from M.

5.2.2. Top-down exploration strategy

Unlike backward assembling strategy, top-down exploration strategy searches for the requested version from the top of a VPG. Each outgoing arc which has an evolution ID in the Evolution ID Priority List with regard to the requested version will be collected provided it is not in an exclusive relation with any other arcs. As to the arcs in an exclusive relation, we need to select the arc which has the highest priority among the exclusively coupled peers. The arcs and nodes with a version that is not in the priority list will not be considered at all.

Let us extract the process schema of version 1.1 from the VPG shown in Fig. 7(c) again. The extraction process starts from \( s \), and then comes through \( a_1, t_1 \) and \( n_1 \) to node \( n_2 \), which has two outgoing arcs, i.e., \( a_1 \) of version 1 and \( a_0 \) of version 0 in an exclusive relation. Thus, \( a_1 \) is selected, and then we arrive to the And-Split gateway \( a_2 \). One of its two branches leads to the And-Join gateway \( a_3 \) via node \( n_3 \); while the other branch leads to node \( n_2 \), which has two outgoing arcs \( a_1 \) and \( b_1 \) in an exclusive relation. As \( b_1 \) holds evolution ID 0.1, the searching will follow \( b_1 \) while bypassing \( a_1 \). We apply this checking to all arcs encountered that are in an exclusive relation when we trace down through the VPG to node \( t \), and finally we can obtain the nodes and arcs for process schema version 1.1. Algorithm 2 formalises the extraction procedure under this strategy.

In addition to the functions defined in Section 5.2.1, this algorithm also uses the following functions:

- Function \text{coupledArcs}(M, n), referenced in line 5, returns a set of node \( n \)'s outgoing arcs that are in an exclusive relation in VPG \( M \);
- Function \text{correlatedArcs}(M, a), referenced in line 9, returns the set of arcs that are in an exclusive relation with arc \( a \) in VPG \( M \);
- Function \text{pickPriorityArc}(A, v), referenced in line 10, returns the arc with the highest priority evolution ID with regard to version \( v \) among set \( A \);
- Function \text{checkNodes}(M, a), referenced in line 17, returns the set of nodes that arc \( a \) links to in VPG \( M \).

This algorithm starts searching from the starting node \( s \), and collects the includable nodes and arcs in sets NTarget and ATarget. When the search arrives at the outgoing arcs of collected nodes, the algorithm (line 4 to line 9) determines whether these arcs are collectable by checking the priority list and exclusive relation. The search moves on to the nodes which are linked from the newly collected arcs and checks whether these nodes have been collected before to reduce potential redundant processing. Finally, the search terminates when it arrives at node \( t \).

5.2.3. Strategy analysis

The time complexity of both algorithms is \( o(n^2) \), where \( n \) is the number of nodes in the VPG. The backward
assembling strategy needs to check most nodes and arcs in the VPG. When removing an arc that is in an exclusive relation with a collected arc, it may leave some nodes with no incoming arcs, and therefore these nodes and all their outgoing arcs will be removed as well. As such, this may result in a chain reaction, which will affect many nodes and arcs in the graph, no matter whether these nodes or arcs are really useful for the extraction or not. In fact, the process schema of version \( v \) represents the result of a series of evolutions, while other evolutions do not contribute to this schema version at all. For example, the evolution from Fig. 2(b) to (c) is totally irrelevant to the process schema of version 2.2. Nevertheless, the backward assembling strategy still processes the nodes and arcs with evolution ID 0.1 during the extraction of schema version 2.2. With respect to identifying irrelevant arcs and nodes, the top-down exploration strategy is relatively intelligent. When the top-down exploration strategy comes across a splitting structure, it leaves all irrelevant arcs untouched as long as they are not on the priority list with regard to the requested version. Thereby, the searching of irrelevant nodes or arcs can be sidestepped.

In the backward assembling strategy, the arc and node collecting operation is processed faster than that in the top-down exploration strategy. This is because it directly picks up the arcs and nodes with certain evolution IDs from the VPG without tracing down the whole control flow structure. Therefore, this strategy works efficiently in the case where the arcs and nodes of the requested version cover a large portion of the VPG. This is because in the backward assembling strategy the first search can significantly reduce the size of the VPG, and therefore simplifies the following searches.

### 5.3. Prototype implementation

For proof of concept, we have implemented a process schema version management prototype using Java API for XML Processing (JAXP). This prototype maintains a VPG as a large XML Process Definition Language (XPDL) with extensions on exclusive relations between arcs and evolution IDs. Though the VPG can be continuously changing (simulated by updating the XPDL file constantly), the prototype enables users to extract a process schema of any version out of the evolving VPG. With this function, the prototype can collaborate with the business process engine to initialise or guide the execution of process instances belonging to different versions. The extracted process schema can be converted into a new XPDL document, which can be graphically displayed as a BPMN diagram in BizAg Process Modeller. This prototype also provides the function of comparing process schemas belonging to different versions. Fig. 10 gives a screenshot of the implemented prototype.

### 6. Related work and discussion

#### 6.1. Related work

Business process version management is closely related to workflow evolution research. Casati et al. [28] presented a workflow modification language (WFML) to support business process schema evolutions. They also devised three main policies, viz., abort, flush and progressive, to manage case evolution. WFML defines a set of declaration primitives and flow primitives for the changes of workflow variables and flow structures. Chiu, Li, and Karlapalem applied the extended object-oriented meta-modelling to support the flexible definition and adaptive features of online workflow evolution and exception handling [29].

Work on adaptive workflows addressed run-time modifications for the purpose of dynamic exception handling. In this area, Hamadi and Benatallah [30] proposed a self-adaptive recovery net (SARN) to support workflow adaptability during unexpected failures. This net extends the classical Petri net by deploying recovery tokens and recovery transitions to represent the dynamic adaptability of a business process, and a set of operations is used to modify the net structure.

In Sadiq et al.’s work on process constraints for flexible workflows [31], they proposed the concept of “pockets of flexibility” to allow ad hoc changes and/or building of workflows, for highly flexible processes. A set of rules specifies that how partially defined workflow fragments are refined at run time.

In AgentWork [32], Muller et al. used temporal estimates to determine which remaining parts of running workflows were affected by an exception and thereby predeterminately performed suitable adaptations. An Event/Condition/Action rule model was proposed to automatically detect logical failures and determine necessary workflow adaptations.

Hallerbach, Bauer and Reichert have discussed the configuration and management of process variants in their Provop framework [33]. Their work aims at handling multitudes of variants that may exist for a process, and at supporting process variant configuration via a set of high-level change patterns. Works on configurable reference process models [34] provided a framework for process configuration. An extended event-driven process chain (EPC) model is used as the reference model to preserve all process variants, i.e., alternative parts of a process model. This reference model can be configured into a concrete process model by selecting proper process variants and removing others, according to customers’ requirements. Later, van der Aalst shifted this work to WF-Net, and provided a formal method for preserving correctness during process configuration [35]. This reference process model targets build-time configuration, and relies on a complex extraction method [36] to check the correctness of the resulting process model. In comparison, our work targets the dynamics introduced from run-time process schema evolutions. Owing to different purposes, the two approaches create their base process graphs in different ways. Our approach records differences between the original process schema and the new schema, and marks the differences with proper evolution IDs. Process change patterns are adopted as standard updating operations to guarantee the correctness of the resulting process schema. The configurable reference process model approach distinguishes between alternative parts in the base process graph for future configuration decisions, and these alternative parts are predefined in a static manner. Technically, in our work the MLED is deployed to trace process schema evolutions.
which may occur concurrently or sequentially, and a special version numbering mechanism is responsible for identifying relationships between these evolutions. In a VPG graph, pre/post-execution points separate tasks and gateways from interferences between different process schema evolutions. With all these supports, it is much simpler to extract a process schema of a given version from the VPG and the extraction will not result in incorrect control flow structures.

In project ADEPTflex [37,38], Rinderle, Reichert and Dadam carried out extensive studies on process schema evolution, including common workflow type and instance changes, disjoint and overlapping process changes. Their work formally specified change operations to process schemas and workflow instances, as well as related migration policies for handling potential conflicts. In their subsequent work, ADEPT2 process management system [39], a change history is kept to store the applied change operations. Based on this change log, they applied purging techniques to extract the effective changes to an instance-specific process schema. However, it does not touch the transformation between subsequent or sibling versions, let alone the compatibility of multiple process versions.

Timestamp approach provides a simple version management solution, which assigns process versions with the creation time of the process version. As the time sequence cannot be overlapped, this approach fails to represent concurrent versions. In addition, each process version is saved as an individual model, which seriously damages information reuse [40,41].

Kradolfer and Geppert [42] presented a framework for dynamic workflow/process schema evolution based on workflow type versioning and workflow migration. In their work, a version tree was proposed to represent workflow schema evolutions, and to keep track of the resulting history. Nevertheless, the version tree only provides primitive supports for version management. Typically, to re-assign a previous version to a running workflow instance, this method has to perform a series of inverse modification operations along the version tree to achieve that version. In our VPG approach, version re-assignment can be realised by switching to the requested version directly.

Based on a version-stamp method, Lee et al. [40] proposed a version control solution to process version changes. Instead of keeping the offset between different versions, they used version blocks to describe the differences between the task groups of different versions. This approach can represent process structural evolutions in a space efficient way, but the time complexity for analysing and partitioning version blocks is considerably higher than ours.

6.2. Comparison and discussion

Table 7 lists the comparison between existing process version management approaches and our VPG approach.

In summary, our approach contributes to business process schema evolution management with the following supports:

- **Evolution versioning**: The proposed versioning mechanism usefully presents the process schema evolutions driven by temporary substitution/adaptation and permanent improvement. The multi-level evolution diagram helps analyse and specify the dependencies between evolutions.
- **Dynamic version shifting**: Within a VPG, a running process instance can change from one version to another. In addition, any new evolution can dynamically update the VPG using the 14 process change patterns proposed in [8], and thereby our approach can provide a comprehensive solution for various process changes.
6.3. Limitations

Nevertheless, the migration to our proposed version management method may bring some tradeoffs, which can be potential limitations. Some tradeoffs and deduced limitations are summarised below, although they may be outweighed by many advantages offered by our methodology:

(1) Model complexity: A VPG keeps all version information and therefore the graph is relatively more complicated than any business process schema of a single version. Nevertheless, the VPG is mainly used to automate business process execution and version shifts by workflow engines, rather than for humans to read. In addition, the extracted business process schemas of respective versions can be provided for human users instead of the original VPG, to improve model readability.

(2) Increasing model space: As a VPG records structural changes of all versions that have occurred, its scale and complexity will increase over time as a business process changes of all versions that have occurred, its scale and therefore the graph is relatively more complicated. Nevertheless, the migration to our proposed version management approach may bring some tradeoffs, which can be outweighed by many advantages offered by our methodology.

7. Conclusions and future work

This paper discussed version management issues in the context of business process schema evolution. A versioning method was proposed to represent different business process schema evolutions and the dependencies between them. A novel version preserving graph model was proposed to capture dynamic modifications to a business process schema via a set of updating operations. Strategies for extracting a business process schema of a given version from a version preserving graph were also discussed, and a prototype was implemented for proof-of-concept.

Future work includes the investigation on data dependency in addition to structural dependency among process schema evolutions and an assessment framework for service continuity in the context of process changes.

Appendix

In Definition 3.4 of Section 3.1, a business process schema is defined as a special graph. Within the context of a process schema, we have given an adapted definition of single-entry-single-exit process fragment, on the basis of its original definition in [44].

Definition A1. (Single-Entry-Single-Exit Fragment). For process schema \( bp=(T, G, P, A, s, t) \), an SESE fragment \( fg \) is a non-empty sub-graph \( (V, E) \) of \( bp \), i.e., \( V \subseteq T \cup G \cup P \) and \( E=Ar \cap (V \times V) \) such that there exist edges \( e \) and \( e' \in \) with \( \{e\}=(\{s\}u(P(V)) \times \{V\}P) \) and \( \{e'\}=(\{V\}P \times \{t\}u(P(V)) \). Edges \( e \) and \( e' \) are called the entry and the exit of \( fg \), respectively. For business process schema \( bp \), \( \xi bp \) denotes the set of all available SESE fragments in \( bp \).

The following theorem is proposed in Section 3.2, and here we give the proof sketch for it.

Theorem. Derivation of evolution relations

\[
\begin{align*}
\text{Rule (3.1): } & \varepsilon_s L \varepsilon_y \Rightarrow \varepsilon_y L \varepsilon_x \quad \text{(symmetry of conflicting relation)} \\
\text{Rule (3.2): } & \varepsilon_s / \varepsilon_y \Rightarrow \varepsilon_y / \varepsilon_x \quad \text{(symmetry of conflict-free relation)} \\
\text{Rule (3.3): } & \varepsilon_s L \varepsilon_y \Rightarrow \lnot \varepsilon_y \oplus \varepsilon_y \\
\text{Rule (3.4): } & (\varepsilon_s L \varepsilon_y) \land (\lnot \varepsilon_y L \varepsilon_z) \Rightarrow \lnot \varepsilon_y \oplus \varepsilon_y \oplus \varepsilon_z
\end{align*}
\]

Table 7

Comparison between existing version management approaches and our VPG approach.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Support process change patterns/operations?</td>
<td>Creation and deletion</td>
<td>Not mentioned</td>
<td>Support 8 primitive process change patterns (Note, they use another set of process change patterns, which are mainly derived from workflow patterns [43])</td>
</tr>
<tr>
<td>Support multiple versions?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support concurrent (temporary) versions?</td>
<td>Yes</td>
<td>Hard to describe concurrent versions</td>
<td>Yes</td>
</tr>
<tr>
<td>Versioning method</td>
<td>Number series</td>
<td>Time stamps</td>
<td>Combinations of version blocks</td>
</tr>
<tr>
<td>Space efficiency</td>
<td>Need to trace along a path of the version tree</td>
<td>Direct and dynamic shift</td>
<td>Direct yet static shift</td>
</tr>
<tr>
<td></td>
<td>Good, as the change operations can be reused to generate new versions</td>
<td>Space consuming, since each version is stored as an individual process model</td>
<td>Excellent with capability of reusing any possible arc and node</td>
</tr>
</tbody>
</table>

Rule (3.5): \((x \otimes y)_L \land (x \otimes y)_L \land (x \otimes y)_L \land (x \otimes y)_L \Rightarrow x \otimes y \otimes z\)

**Proof.** Rules (3.1) and (3.2) are obvious.

Rule (3.3) can be proven directly by the definition of evolution composition, since \(\otimes\) is only applicable to non-conflicting evolutions.

With these three rules, we can further explore the relations among more process schema evolutions.

Rule (3.4) determines whether three evolutions can be composed together, in the case that two out of three evolutions are conflicting, but these two are not conflicting with the third evolution. From Rule (3.3), it is easy to see that condition \(x \otimes y\) already determines that \(x \otimes y\) does not hold. Therefore, \(x \otimes y \otimes z\) cannot hold either. Rule (3.4) is very useful in guiding evolution compositions, which is one of the main topics of Section 3.3.

Rule (3.4) does not answer whether three non-conflicting can be composed together or not. Rule (3.5) specifies the conditions for such compositions. The proof of Rule (3.5) is given by enumerating all possible situations. Owing to the conditions on the left side, there are only five possible situations for the three evolutions, namely \(x \otimes y\), \(x \otimes y\), \(x \otimes y\), \((x \otimes y)\land (x \otimes y)\), and \((x \otimes y)\land (x \otimes y)\). In the first two situations, composite evolution \(x \otimes y\) certainly holds. In situation \((x \otimes y)\land (x \otimes y)\), it is easy to see that \(x \otimes y\). Further, \(x \otimes y\) can be obtained as \(x \otimes y\) and \(x \otimes y\) are not conflicting. In situation \((x \otimes y)\land (x \otimes y)\), first we can have \((x \otimes y)\) as condition \(x \otimes y\) is given in the left part of Rule (3.5). Then, we have \(x \otimes y\), and from that we can get \(x \otimes y\) from situation \((x \otimes y)\land (x \otimes y)\). Note the last item on the left side, i.e., \(x \otimes y\), indicates a loop evolution situation, which is not allowed for composition, because there is not any independent starting evolution that can trigger the whole evolution chain.

These rules together can guide the evolution composition, and thereby help the construction of the Multi-Level Evolution Diagram (MLED) in Section 3.3.

**References**


