Workflow modeling for virtual processes: an order-preserving process-view approach

Duen-Ren Liu*, Minxin Shen

Institute of Information Management, National Chiao Tung University, 1001 Ta Hsueh Rd., Hsinchu 300, Taiwan

Received 15 October 2000; received in revised form 10 March 2002; accepted 20 April 2002

Abstract

Conducting workflow management allows virtual enterprises to collaboratively manage business processes. Given the diverse requirements of the participants involved in a business process, providing various participants with adequate process information is critical to effective workflow management. This work describes a novel process-view, i.e., an abstracted process which is derived from a base process to provide process abstraction, for modeling a virtual workflow process. The proposed process-view model enhances the conventional activity-based process models by providing different participants with various views of a process. Moreover, this work presents a novel order-preserving approach to derive a process-view from a base process. The approach proposed herein can preserve the original ordering of activities in the base process. Additionally, a formal model is presented to define an order-preserving process-view. Finally, an algorithm is proposed for automatically generating an order-preserving process-view. The proposed approach increases the flexibility and functionality of workflow management systems.

Keywords: Workflow management; Process abstraction; Virtual workflow; Process modeling

1. Introduction

Workflow management via workflow management systems (WFMSs) not only facilitates electronic commerce, but also allows virtual enterprises to collaboratively manage business processes. As an effective process management tool, WFMSs allow businesses to analyze, simulate, design, enact, control and monitor their overall business processes [1,2]. The support of a WFMS allows various participants to collaborate in effectively managing a workflow-controlled business processes. The participants represent particular positions in a company or particular companies in a supply chain. In practice, these participants possess different needs and levels of authority when obtaining information on business processes. To facilitate effective workflow management, a WFMS should provide various participants with adequate process information.

For example, a high-level manager may require aggregated information on a process, while a marketing manager may not have the authority or need to know each specific step of the production flow. These requirements create the
need for a flexible process model capable of providing appropriate processes abstraction for various roles within an enterprise. Furthermore, interorganizational coordination via WfMSs has become a critical success factor for businesses in rapidly fluctuating and complex business environments. Besides the interoperability issues of heterogeneous WfMSs, in a WfMS-supported supply chain (or called multi-enterprise process [3]), each participatory organization wants to conceal its own processes from other organizations, and different organizations require different supply chain information. In sum, providing aggregated information or encapsulating sensitive data requires the development of a workflow model capable of offering adequate abstracted processes for different levels, units, and organizations.

Despite notational differences, activity-based methodologies are extensively used process modeling techniques, and have been extensively adopted for commercial products, research projects, and standards, e.g., MQSeries Workflow [4], Ultimus [5], METEOR [6], and workflow management coalition (WFMC) process definition metamodel [7]. A typical activity-based approach designs a workflow through a top-down decomposition procedure. This stepwise refinement allows a modeler to define a process more easily and completely than do one-step approaches. However, resultant layered process definitions do not always fit into an organizational hierarchy, despite providing several different levels of hierarchical abstraction. Therefore, hierarchically decomposing a process may not provide each organizational level with an appropriate view of that process. Despite forcing a process modeler to follow an organizational hierarchy while decomposing a process, different organizational units (divisions/companies) may have difficulties in obtaining adequate abstractions of the process/supply chain they participate in. The activity-based approach cannot adequately provide different participants with varied abstracted processes.

The activity-based approach should be enhanced to provide different process abstractions. Several formal process modeling techniques, including process algebras and Petri Nets [8–11], can provide process abstractions by renaming activities to silent activities that are not observable. Such abstraction is considered as partial abstraction since it provides partial observability of a process. Although useful in satisfying some of the needs of process abstractions, partial abstraction may be unable to adequately address the needs of managers or collaborative parties who require aggregated information on a process.

Based on the notion of views in database management systems (DBMSs), this work proposes a novel virtual workflow process, i.e., a process-view, in a WfMS. A process-view, i.e., an abstracted process derived from an implemented base process, is employed to provide aggregate abstraction of a process. During workflow build time, a process modeler does not need to be concerned with process abstraction, and can focus solely on process design, using a top-down decomposition procedure to define the process in detail. The modeler can then use a process-view definition tool to define multiple abstracted processes, i.e., process-views. During run time, creating a process instance initiates its corresponding process-view instances. Each participant can retrieve and monitor appropriate process information via the related process-view instance. Therefore, coordination within an organization or across multiple organizations can be improved.

Although process design is a specialized and top-down procedure, process-view design is a generalized and bottom-up procedure. Process-views allow a WfMS to provide various aggregated views of a process for different levels or departments in an organization or for different organizations in a supply chain. Several approaches can be adopted to construct a process-view. This work describes a novel order-preserving approach in which the constructed process-view can preserve the original ordering of activities in the base process. A formal model is also presented to define an order-preserving process-view. Theoretical analysis is performed herein, indicating that the defined process-view satisfies the order-preserving property. Moreover, an algorithm is proposed to automatically generate an order-preserving process-view.
The remainder of this paper is organized as follows. Section 2 formally defines business processes. Section 3 then describes and defines a process-view. Next, Section 4 presents the proposed order-preserving approach to construct a process-view. Section 5 then discusses and compares related work on workflow modeling. Conclusions and future work are finally made in Section 6. Appendix A provides proofs of all lemmas.

2. Workflow model: a base process

A process that may have multiple process-views is termed a base process herein. Activity-based workflow models generally use activities and dependencies to describe a process. Dependencies prescribe the ordering relationships between activities within a process. According to WfMC [7], the following six ordering structures may appear in business processes. Sequence: An activity has a single subsequent activity. AND-SPLIT: An activity splits into multiple parallel activities that are all executed. XOR-SPLIT: An activity splits into multiple mutually exclusive alternative activities, only one of which is followed. AND-JOIN: Multiple parallel executing activities join into a single activity. XOR-JOIN: Multiple mutually exclusive alternative activities join into a single activity. Loop: One or more activities are repeatedly executed until the exit condition is satisfied.

The above ordering constructs are not arbitrarily combined. For example, AND-SPLIT must pair with AND-JOIN, and XOR-SPLIT must pair with XOR-JOIN. Wrong combinations of ordering structures may cause structural conflicts such as deadlock and non-reachability. Verification issues are beyond the scope of this work. Please refer to Woflan [12] and FlowMake [13] to verify the correctness of process definitions. This work assumes that the given process definitions are structurally correct.

Moreover, a well-structured loop in a process definition should have a single entry and a single exit, as the iteration statements in programming languages. Allowing multiple entries/exits makes the complex control flow hard to understand, and induces ambiguities in the evaluation of exit conditions. Leymann et al. [14] also claim that race conditions may occur in arbitrary loops. Thus, this work prescribes that a loop must be single-entry and single-exit.

A graphical representation of a process resembles a directed graph [15] in which each node is an activity and each directed edge is a dependency. This work uses a rectangle to denote an activity and a solid arrow to represent a dependency in a process graph. Moreover, a blank arrow indicates a loop dependency used to construct a loop structure. Fig. 1 depicts a sample process. A loop dependency from \( a_4 \) to \( a_2 \) indicates that activities \( a_2, a_3, \) and \( a_4 \) form a loop in which \( a_2 \) is the entry point and \( a_4 \) is the exit point. The single-entry loop is converted into a multi-entry loop, for example, by adding a dependency from \( a_1 \) to \( a_3 \).

2.1. Formal model

To elucidate the derivation of process-views from base processes, a formal model is introduced first for describing base processes. The model is revised from the standard WfMC process definition language [7] and focuses only on activities and dependencies to simplify the discussion.

**Definition 1** (Process and sub-process). A process \( P \) is a 2-tuple \( \langle A, D \rangle \), where \( A \) is a set of activities and \( D \) is a set of dependencies and loop dependencies. A process \( P = \langle A', D' \rangle \) is a sub-process of \( P \) if \( A' \subseteq A \) and \( D' \subseteq D \).

**Definition 2** (Dependency). A dependency is a 4-tuple \( \langle \text{activity } x, \text{ activity } y, \text{ type } R, \text{ condition } C \rangle \), denoted by \( \text{dep}(x, y, R, C) \), where \( x, y \in A \). Condition \( C \) represents the constraints that determine whether routing can proceed from \( x \) to \( y \). The dependency \( \text{dep}(x, y, R, C) \) is an outgoing dependency of \( x \) and an incoming dependency of \( y \).
Activity \( x \) is called the preceding activity and \( y \) is called the succeeding activity in \( \text{dep}(x, y, R, C) \). Type \( R \) indicates that \( \text{dep}(x, y, R, C) \) is not a loop dependency.

**Definition 3** (Loop dependency and loop). A loop dependency is a 4-tuple \( \langle \text{activity } \text{ex}, \text{activity } \text{ei}, \text{type } \text{L}, \text{loop condition } \text{LC} \rangle \), denoted by \( \text{dep}(	ext{ex}, \text{ei}, \text{L}, \text{LC}) \), where \( \text{ex}, \text{ei} \in \text{A} \). Type \( \text{L} \) refers to a loop dependency. Additionally, \( \text{dep}(	ext{ex}, \text{ei}, \text{L}, \text{LC}) \) implies that there exists a loop, denoted by \( \text{lp}(\text{ei}, \text{ex}) = \langle \text{A}_{\text{lp}(\text{ei}, \text{ex})}, \text{D}_{\text{lp}(\text{ei}, \text{ex})} \rangle \), where \( \text{A}_{\text{lp}(\text{ei}, \text{ex})} = \{\text{ei}, \text{ex}, \text{and all activities between } \text{ei} \text{ and } \text{ex} \} \), and \( \text{D}_{\text{lp}(\text{ei}, \text{ex})} = \{\text{dep}(\text{x}, \text{y}, \text{R}, \text{C}_{\text{xy}}) \text{ or } \text{dep}(\text{x}, \text{y}, \text{L}, \text{LC}_{\text{xy}}) \} \text{ where } \text{x, y} \in \text{A}_{\text{lp}(\text{ei}, \text{ex})} \}; \text{ei} \) is the entry point and \( \text{ex} \) is the exit point. Notably, a loop is here restricted to having one entry and one exit point. All activities of \( \text{A}_{\text{lp}(\text{ei}, \text{ex})} \) are repeatedly executed until \( \text{LC} \) is evaluated as false.

**Definition 4** (Activity). An activity \( \text{ba} \) is a 3-tuple \( \langle \text{SPLIT\_flag}, \text{JOIN\_flag}, \text{SC} \rangle \), where

1. **SPLIT\_flag** may be “NULL”, “AND”, or “XOR”. “NULL” means that \( \text{ba} \) has a single outgoing dependency (Sequence). “AND” means that \( \text{ba} \) has multiple outgoing dependencies labeled with identical conditions (AND-SPLIT). “XOR” means that \( \text{ba} \) has multiple outgoing dependencies associated with mutually exclusive conditions (XOR-SPLIT).

2. **JOIN\_flag** may be “NULL”, “AND”, or “XOR”, and is used to derive SC. “NULL” indicates that this activity has a single incoming dependency (Sequence). Given multiple incoming dependencies, “AND” indicates that \( \text{ba} \) can be started if the conditions of all incoming dependencies are satisfied (AND-JOIN), while “XOR” indicates that \( \text{ba} \) can be started if one of its incoming dependencies has satisfied conditions and the others are associated with unsatisfied conditions (XOR-JOIN).

3. **SC** is the starting condition of \( \text{ba} \). SC explicitly expresses the condition whether \( \text{ba} \) can be started, and is derived from the JOIN\_flag and the condition fields of incoming dependencies. First, if JOIN\_flag is NULL, SC equals the condition (C) of the incoming dependency.

Secondly, if JOIN\_flag is XOR, SC equals the Boolean XOR combination of the conditions of all incoming dependencies. Finally, if JOIN\_flag is AND, SC equals the Boolean AND combination of the conditions of all incoming dependencies.

**Enacting a process.** During run-time, an execution of a process is called a process instance. For \( \text{dep}(x, y, R, C) \), \( C \) is not evaluated until \( x \) is completed. The evaluation of \( C \) is either true or false. The fact that \( x \) is completed and \( C \) is evaluated as true is one precondition that determines whether \( y \) can be started. For convenience, “a dependency is evaluated as true/false” states that the dependency’s condition field is evaluated as true/false.

The SC field of an activity \( \text{ba} \) is evaluated when all incoming dependencies of \( \text{ba} \) have been evaluated. \( \text{ba} \) can be started when \( \text{SC} \) is evaluated as true. Note that \( \text{SC} \) is derived from the JOIN\_flag and the condition fields of \( \text{ba} \)’s incoming dependencies. If \( \text{ba} \) is started, then its outgoing dependencies are evaluated after the completion of \( \text{ba} \). If \( \text{SC} \) is evaluated as false, then \( \text{ba} \) is not executed in the process instance, and the outgoing dependencies of \( \text{ba} \) are evaluated as false.

An activity is called a fired activity in a process instance if its \( \text{SC} \) is evaluated as true and it is executed in the process instance. Moreover, an activity is called a non-fired activity in a process instance if its \( \text{SC} \) is evaluated as false and it is not executed in the process instance. Notably, an activity that is non-fired in a process instance may be fired in other process instances. For convenience, “an activity is evaluated as fired/non-fired” states that the activity’s \( \text{SC} \) is evaluated as true/false.

For \( \text{dep}(\text{ex}, \text{ei}, \text{L}, \text{LC}) \), after \( \text{ex} \) is completed, \( \text{LC} \) is (re)evaluated to decide whether the loop \( \text{lp}(\text{ei}, \text{ex}) \) is repeated. If \( \text{LC} \) is evaluated as true, then \( \text{lp}(\text{ei}, \text{ex}) \) is repeated. Notably, each execution of \( \text{lp}(\text{ei}, \text{ex}) \) starts a new iteration and initiates a new execution context of \( \text{lp}(\text{ei}, \text{ex}) \). The activities and (loop) dependencies of \( \text{lp}(\text{ei}, \text{ex}) \) are reset for re-execution and re-evaluation when a new iteration of \( \text{lp}(\text{ei}, \text{ex}) \) is started. Thus, the activities and (loop) dependencies within a new iteration are
initially non-evaluated. The activities of \( lp(ei, ex) \) are re-evaluated as fired/non-fired in an iteration of \( lp(ei, ex) \). Additionally, the completion of an activity that belongs to \( lp(ei, ex) \) means that the activity is completed in an iteration of \( lp(ei, ex) \). The activity may be started/completed again in the follow-on iterations of \( lp(ei, ex) \). Besides, the completion of \( lp(ei, ex) \) means that the entire loop stops; that is, \( LC \) is evaluated as false and \( lp(ei, ex) \) is not repeated. The starting of \( lp(ei, ex) \) means the starting of \( ei \) in the first iteration of \( lp(ei, ex) \).

Notably, where \( lp(ei, ex) \) is a nested loop if it belongs to another loop which is called the outer loop. For the two levels of nesting, the above is also applied to each iteration of the outer loop. Furthermore, the above can be recursively applied to the case of more than two levels of nesting.

If a process contains loops, then two revisions are needed for the entry and exit points of those loops. (1) For \( dep(x, y, R, C) \), if \( x \) is the exit point of a loop (i.e., \( x \) is also associated with a loop dependency), then \( C \) is evaluated when \( x \) is completed and the loop condition is evaluated as false (i.e., the loop stops). Moreover, if \( x \) is evaluated as non-fired, then \( C \) and the loop condition are evaluated as false. (2) For an activity \( ba \), if \( ba \) is the entry point of a loop, then whether \( ba \) can be started depends on both \( SC \) and the loop condition. In the first iteration of the loop, \( ba \) is started/fired if \( SC \) is evaluated as true. In the follow-on iteration(s), \( ba \) is started/fired if the loop condition is evaluated as true (i.e., the loop is repeated).

For convenience, in the rest of this work, \( dep(x, y, \ldots) \) indicates a situation in which type and condition field are free.

**Definition 5** (Adjacent). Two distinct activities \( x \) and \( y \) are adjacent if \( dep(x, y, \ldots) \) exists.

**Definition 6** (Path). A path of length \( k \) from an activity \( x \) to an activity \( y \) in a process \( P = \langle A, D \rangle \) is a sequence of activities \( a_0, a_1, \ldots, a_k \) such that \( x = a_0, y = a_k \), and \( dep(a_{i-1}, a_i, \ldots) \in D \) for \( i = 1, 2, \ldots, k \). The length of the path is the number of (loop) dependencies on the path. The path contains the activities \( a_0, a_1, \ldots, a_k \) and the (loop) dependencies \( dep(a_{i-1}, a_i, \ldots) \) for \( i = 1, 2, \ldots, k \).

**Definition 7** (Loop-derived sub-process). Given a process \( P = \langle A, D \rangle \), if \( dep(ex, ei, L, LC) \) exists in \( P \), then a loop \( lp(ei, ex) \) exists in \( P \). A sub-process can be derived by excluding \( dep(ex, ei, L, LC) \) from \( lp(ei, ex) \). That is, for \( lp(ei, ex) \) in \( P \), a sub-process \( LP(ei, ex) = \langle ALP(ei, ex), DLP(ei, ex) \rangle \) where \( ALP(ei, ex) = \{x | x \ belongs \ to \ lp(ei, ex) \} \), and \( DLP(ei, ex) = \{dep(x, y, \ldots) | x, y \in ALP(ei, ex) \} \). \( lp(ei, ex) \) can be viewed as a repeatedly executed \( LP(ei, ex) \). \( LP(ei, ex) \) is called a loop-derived sub-process that is derived from \( lp(ei, ex) \).

To clarify the target for discussion, \( lp \) and \( LP \) are used in parts of this work to denote a loop and a loop-derived sub-process, respectively.

**Definition 8** (Ordering relation). Given a process \( P = \langle A, D \rangle \) and a sub-process \( P' = \langle A', D' \rangle \) of \( P \), the ordering relation between an activity \( x \) and an activity \( y \) in \( P' \) is defined as follows:

1. If there exists a path from \( x \) to \( y \) in \( P' \), then the ordering of \( x \) is higher than \( y \), i.e., \( x \ precedes \( y \). Their ordering relation is denoted by \( x > y \) or \( y < x \) which means \( x > y \) (or \( y < x \)) holds in \( P' \).
2. If no path exists from \( x \) to \( y \) or from \( y \) to \( x \) in \( P' \), then \( x \) and \( y \) are ordering independent, i.e., \( x \) and \( y \) proceed independently. Their ordering relation is denoted by \( x \parallel y \) which means \( x \parallel y \) holds in \( P' \).

Definition 8.1 implies that if another activity \( z \in A' \) exists, such that \( x > y \) and \( y > z \) hold in \( P' \), then \( x > z \) holds in \( P' \). Notably, the ordering relations may hold in \( P \) but do not hold in the sub-processes. For example, in Fig. 1, the ordering relations among \( a_2, a_3, \) and \( a_4 \) in \( P \) are \( a_2 > a_3, a_2 < a_3, a_2 > a_4, a_2 < a_4, a_3 > a_4, \) and \( a_3 < a_4 \) because the ordering relations can be derived from the paths that include \( dep(a_4, a_2, L, LC_{a_2}) \). However, only the ordering relations \( a_2 > a_3, a_2 > a_4, \) and \( a_3 > a_4 \) hold in \( LP(a_2, a_4) \), derived from \( lp(a_2, a_4) \), because the ordering relations are derived from the paths that exclude \( dep(a_4, a_2, L, LC_{a_2}) \). The semantics of “\( a_2 < a_3 \) holds in \( P' \)” can be elucidated as \( a_3 \) is executed in the iteration of \( lp(a_2, a_4) \) before \( a_2 \) is executed in the follow-on iteration of \( lp(a_2, a_4) \) in an instance of \( P \). The semantics of “\( a_2 > a_3 \) holds in \( LP(a_2, a_4) \)” is elucidated as \( a_2 \) is executed before \( a_3 \) in each iteration of \( lp(a_2, a_4) \).
3. Virtual process: a process-view

Views in DBMSs are virtual tables generated from either physical tables or previously defined views. Similarly, process-views are generated from either physical processes (base processes) or other process-views, and are considered virtual processes. During design time, a process modeler defines various process-views based on the roles of participants. During run time, a WfMS initiates all process-view instances if their base process is initiated. Process-views allow a process modeler to flexibly provide different roles (i.e., different levels or departments within an organization or different organizations in a supply chain) with appropriate views of an implemented process. This ability implies that a modeler can provide only the information that participants need to know, while filtering and concealing information as desired. Fig. 2 illustrates this concept.

Assume that the base process in Fig. 2 is a manufacturing process. Marketers do not need to know every step in the process, although they must know the progress of order fulfillment to serve their customers. A process modeler can design an appropriate process-view for the marketing department as follows: \(a_1, a_2, a_3\) are mapped into \(va_1\); \(a_4\) and \(a_5\) are mapped into \(va_2\); \(a_6\) and \(a_7\) are mapped into \(va_3\). When a customer places a new order, the WfMS initiates a new manufacturing process instance and corresponding process-view instances. Marketers can use the information from the process-view instance to serve customers. A case study in Section 4.4 demonstrates more applications of process-views.

Like process design, the design of a process-view must first identify any activities within it and then arrange them based on dependencies and ordering structures. However, an “activity” in a process-view is not performed, but rather is used to express the progress information of a set of activities. Hence, to differentiate the terminology used in base process and process-view, this work uses the terms virtual activity and virtual (loop) dependency for the process-view, and the terms base activity and base (loop) dependency for the base process. While a virtual activity is derived from a bottom-up aggregation of a set of activities within a process, a base activity is generated from a top-down decomposition of a business process. A process modeler develops a process definition and then defines process-views.

3.1. Formal model

Definition 9. (Process-view). A process-view \(VP\) is a 2-tuple \(\langle VA, VD \rangle\), where \(VA\) is a set of virtual activities and \(VD\) is a set of virtual dependencies and virtual loop dependencies. During run-time, an execution of a process-view is called a process-view instance.

A process-view has a corresponding base process from which it is derived. A virtual activity is an abstraction of a set of base activities and correlative base (loop) dependencies. A virtual dependency connects two virtual activities in a process-view, and a virtual loop dependency constructs a loop structure in a process-view.
According to the different properties of a base process, various approaches can be developed to derive VA and VD. Section 4 presents an approach that preserves the original execution order of a base process.

Regardless of how VA and VD are derived, the definitions of path, loop (denoted by $vlp(\ldots)$), loop-derived sub-process-view (denoted by $VLP(\ldots)$) and ordering relations for a process-view are similar to Definitions 3, 6, 7 and 8. For example, the definition of “path” for a process-view can be obtained by replacing “activity/(loop) dependency/process” in Definition 6 with “virtual activity/virtual (loop) dependency/process-view”. Therefore, those definitions are omitted herein for brevity.

Fig. 3 illustrates the relationship between the components of the novel model. Base process relevant data defines the data created and used within each process instance during workflow enactment [7]. Similarly, the produced and consumed data of a process-view is termed process-view relevant data. Since a virtual activity is an abstraction of a set of base activities, the produced/consumed data of a virtual activity is the set of data that is produced/consumed by the base activities belonging to the virtual activity.

4. Order-preserving approach

Execution order is an important property of business processes, particularly continuous manufacturing processes such as chemical materials, integrated circuit (IC), and steel. This section first introduces three rules that a process-view must follow to preserve the ordering property. Then virtual activities and (loop) dependencies in an order-preserving process-view are formally defined based on these rules. Essential activities, i.e., activities that a modeler wants to conceal or aggregate in a virtual activity, are proposed to simplify the procedure of defining a virtual activity. Next, a liquid crystal display (LCD) production flow is used to illustrate the application of process-views. Finally, novel algorithms that automatically generate legal virtual activities and virtual (loop) dependencies are also proposed herein.

4.1. Basic rules

The following introduces three rules for defining virtual activities.

4.1.1. Rule 1 membership

A virtual activity can be viewed as a set of activities of a base process. Restated, a virtual activity comprises base activities. For further abstraction, a virtual activity may comprise other previously defined virtual activities. As illustrated in Fig. 4, member activities of a virtual activity may include base activities, virtual activities, or both. The membership among base activities and virtual activities is defined transitively. That is, if a base activity $ba$ is a member of a virtual activity...
va\textsubscript{1}, and \( va_{1} \) is a member of another virtual activity \( va_{2} \), then \( ba \) is also a member of \( va_{2} \).

4.1.2. Rule 2 atomicity

A virtual activity is an atomic unit of processing. The following first considers the case of base processes that do not contain loops to explain clearly the atomicity rule. Notably, a virtual activity may contain other virtual activities according to the membership rule. The behavior of a virtual activity is determined by its member base activities since base activities are the actual execution units. For example, if \( va_{1} \) contains \( va_{2} \) and \( ba_{1} \), and \( va_{2} \) contains \( ba_{2} \) and \( ba_{3} \), then member base activities of \( va_{1} \) are \( ba_{1} \), \( ba_{2} \), and \( ba_{3} \).

In a base process, each base activity is executed atomically; that is, the starting of a base activity implies all its preceding base activities have been evaluated and the fired ones are completed in the base process instance. Thus, the claim that “each virtual activity proceeds atomically in a process-view” rests on three requirements.

1. A virtual activity is started if one member base activity is started, and is completed if all member base activities have been evaluated and each fired member base activity is completed.

2. In a process-view instance, a virtual activity is evaluated as fired if one member base activity is evaluated as fired, and is evaluated as non-fired if all member base activities are evaluated as non-fired.

3. The starting of a virtual activity implies that all its preceding virtual activities have been evaluated and each preceding fired virtual activity is completed in the process-view instance.

Requirements 1 and 2 describe how to decide the behavior of a virtual activity, and requirement 3 specifies the behavior of a process-view. Each virtual activity is an indivisible unit in a process-view, and, thus, can be decided as started if one member base activity is started. As stated in Section 2.1 (enacting a process), whether a base activity is either fired or non-fired in a base process instance is unknown until the \( SC \) of the base activity has been evaluated. Consequently, a virtual activity can be decided as completed when the starting condition of each member base activity has been evaluated and each fired member base activity is completed. Moreover, the starting of a virtual activity occurs after the completion of its preceding fired virtual activities in a process-view instance.

Consider three process-views \( VP_{a} \), \( VP_{b} \), and \( VP_{c} \) as shown in Fig. 5 where each virtual activity follows requirements 1 and 2. When \( a_{3} \) is started, \( a_{1} \) and \( a_{2} \) should be completed in the base process. Under such conditions, three process-views behave differently from each other. In \( VP_{a} \), \( va_{2} \) is started (because \( a_{3} \) is started) and \( va_{1} \) has been completed. This behavior follows atomicity property. Contrarily, in \( VP_{b} \), \( va_{1} \) is not completed since whether \( a_{4} \) is fired is currently unknown, i.e., the starting condition of \( a_{4} \) is not yet evaluated. Meanwhile, \( va_{2} \) is completed and \( va_{3} \) is started. Restated, the current status is that \( va_{1} \) is started, \( va_{2} \) is completed, and \( va_{3} \) is started. This behavior is incompatible with the ordering relations of \( VP_{b} \). Therefore, \( VP_{b} \) violates the atomicity property since it does not satisfy requirement 3. The behavior of \( VP_{c} \) is similar to that of \( VP_{b} \). However, \( va_{1} \) of \( VP_{b} \) cannot be decided as completed until the end of the base process instance, while \( va_{1} \) of \( VP_{c} \) can be decided as completed when \( a_{3} \) is completed. Notably, the virtual activities defined in \( VP_{d} \) are identical to those defined in \( VP_{c} \). However, these activities have different ordering relations. \( VP_{d} \) also violates
the atomicity property because $va_1$ is not completed when $va_2$ is started.

As another example, in Fig. 6, the starting of virtual activity $va_3$ implies that $va_2$ is completed since $va_2 > va_3$. If the $SPLIT$ flag of $a_1$ is AND and the $JOIN$ flag of $a_4$ is AND, then $a_4$ cannot be started until $a_2$ and $a_3$ are completed. The completion of $va_2$ implies that $a_2$ and $a_3$ are completed. However, if the $SPLIT$ flag of $a_1$ is XOR and the $JOIN$ flag of $a_4$ is XOR, then $a_4$ cannot be started until $a_2$ or $a_3$ are completed. The completion of $va_2$ implies that either $a_2$ or $a_3$ is completed, while the other is non-fired in the base process instance.

**Base processes that contain loops.** The repeatable property distinguishes the base activities that belong to a loop from the other ones. The operational semantics of a virtual activity should not infer a new repeatable behavior that does not occur in the base process since a virtual activity is only an abstraction of a set of base activities. Restated, although capable of concealing and revealing original behavior, an abstraction should not imply new behavior. Hence, four cases are possible to define a virtual activity with respect to a loop $lp$ of a base process. First, the virtual activity does not contain base activities that belong to $lp$. Second, the virtual activity contains the entire $lp$. Third, the virtual activity only contains partial base activities of $lp$. Fourth, the virtual activity contains not only partial (not all) base activities of $lp$, but also some base activities that do not belong to $lp$.

The second definition, although concealing the repeatable behavior, does not infer new one, even if it contains the entire $lp$ and some base activities that do not belong to $lp$. Regarding the fourth definition, the repeatable behavior of the virtual activity corresponds to the repeated execution of $lp$. Such repeatable behavior implies that the base activities that do not belong to $lp$ are also involved in the repeatable semantics of $lp$ since a virtual activity is an atomic unit. Hence, the fourth definition imposes repeatable semantics on the base activities that do not belong to $lp$, and is not a reasonable abstraction.

The above atomicity requirements are extended to tackle the base process that contains loops, and are summarized below. For clarity, the following
describes the case of base processes that do not have nested loops. However, this rule can be recursively applied to handle nested loops.

Summary of atomicity rule. Four cases are possible to define a virtual activity \( va \) with respect to a loop \( lp \) in the base process.

Case 1: \( va \) does not contain base activities that belong to \( lp \).

Case 2: \( va \) contains the whole \( lp \). The whole \( lp \) is viewed as a member base activity; it is started when \( lp \) starts and is completed when \( lp \) stops; it is evaluated as fired/non-fired if the \( SC \) of the entry point of \( lp \) is evaluated as true/false.

1. \( va \) is started if one member base activity is started, and is completed if all member base activities have been evaluated and each fired member base activity is completed.
2. In a process-view instance, \( va \) is evaluated as fired if one member base activity is evaluated as fired, and is evaluated as non-fired if all member base activities are evaluated as non-fired.
3. The starting of \( va \) implies that all its preceding virtual activities have been evaluated and each preceding fired virtual activity is completed in the process-view instance.

Case 3: \( va \) only contains partial base activities of \( lp \) in the base process; accordingly, \( va \) belongs to a loop \( vlp \) in the process-view. A correspondence exists between \( vlp \) and \( lp \), just as a process-view has a corresponding base process. The completion of \( va \) that belongs to \( vlp \) means that \( va \) is completed in an iteration of \( vlp \) in the process-view instance; \( va \) may be started/completed again in the follow-on iterations of \( vlp \) in the process-view instance. Similarly, the completion of member base activities means that these base activities are completed in an iteration of \( lp \) in the base process instance. Member base activities may be started/completed again in the follow-on iterations of \( lp \) in the base process instance.

1. Starting and completion of \( va \):
   \( ◦ \) Starting: \( va \) is started in an iteration of \( vlp \) in the process-view instance if one member base activity is started in an iteration of \( lp \) in the base process instance.
   \( ◦ \) Completion: \( va \) is completed in an iteration of \( vlp \) in the process-view instance if all member base activities have been evaluated and each fired member base activity is completed in an iteration of \( lp \) in the base process instance.

2. In an iteration of \( vlp \) in the process-view instance, \( va \) is evaluated as fired if one member base activity is evaluated as fired in an iteration of \( lp \) in the base process instance, and is evaluated as non-fired if all member base activities are evaluated as non-fired in an iteration of \( lp \) in the base process instance.
3. In an iteration of \( vlp \) in the process-view instance, the starting of \( va \) implies that all its preceding virtual activities have been evaluated and each preceding fired virtual activity is completed.

Case 4: \( va \) cannot be defined as the case that contains not only partial (not all) base activities of \( lp \), but also some base activities that do not belong to \( lp \).

Discussion. In case 2, a virtual activity encapsulates a whole loop, thus it is reasonable to view the loop as a member activity. That is, the behavior of repeated execution of the loop is hidden by the virtual activity. Therefore, the completion of the member activity refers to the completion of the whole loop; that is, the end of the cyclic execution of the loop. However, a virtual activity in case 3 does not hide a whole loop. Hence, the virtual activity must reveal that its member activities are repeatedly executed. That is, repeatable behavior should be preserved. Accordingly, in case 3, the process-view must contain a loop that corresponds to the loop of the base process. That is, member base activities of the virtual activities that belong to \( vlp \), must belong to \( lp \). On the other hand, if a loop exists in the process-view, then a corresponding loop can be found in the base process.

Consider the two process-views \( VP_a \) and \( VP_b \) as shown in Fig. 7. Case 1 determines the behavior of \( va_1 \) and \( va_3 \) in \( VP_a \). Similarly, case 1 determines the behavior of \( va_1 \) and \( va_4 \) in \( VP_b \). However, the behavior of \( va_2 \) in \( VP_a \) is determined by case 2 since \( va_2 \) contains a whole loop \( lp(a_2, a_4) \). In \( VP_a \), \( va_2 \) is started if \( a_2 \) is started, and it is completed
when \( lp(a_2, a_4) \) stops and \( a_5 \) is completed. As to \( VP_b \), case 3 determines the behavior of \( va_2 \) and \( va_3 \). Completion of \( va_2 \) in an iteration of the loop \( vlp(va_2, va_3) \) in \( VP_b \) means \( a_2 \) and \( a_3 \) are completed in one iteration of the loop \( lp(a_2, a_4) \) in the base process instance. In the follow-on iteration(s), \( va_2 \) is started/completed again. Notably, \( vlp(va_2, va_3) \) of \( VP_b \) corresponds to \( lp(a_2, a_4) \) of the base process.

4.1.3. Rule 3 order preservation

Briefly, this rule states that a process-view must preserve the original ordering relations of a base process. Order preservation provides a syntactical constraint that ensures that a process-view follows the atomicity property. The following first explains order preservation and then summarizes this rule.

A situation in which the ordering relation between two virtual activities is “\( > \)” in a process-view infers that the implied ordering relation between the respective members of these virtual activities is “\( > \)” due to the atomicity property of virtual activity. Corresponding inferences also hold for the ordering relations “\( < \)” and “\( \sim \)” respectively. For example, the process-view in Fig. 6 shows that the ordering relation between \( va_1 \) and \( va_2 \) is \( va_1 > va_2 \). “\( va_1 > va_2 \)” infers that “\( > \)” is the implied ordering relation between any member of \( va_1 \) and any member of \( va_2 \) because a virtual activity is an atomic unit; that is, \( va_1 > va_2 \) implies \( a_1 > a_2 \) and \( a_1 > a_3 \). Notably, the implied ordering relations may not conform to the ordering relations of the base process; that is, the implied ordering relations may not hold in the base process.

Consider the base process depicted in Fig. 8(a), which seeks to define a virtual activity that must contain activities \( a_{11} \) and \( a_{22} \). Figs. 8(b) and (c) provide two possible definitions. In the base process, three branches proceed independently and autonomously, while the ordering relation between \( a_{13} \) and \( a_{22} \) is \( a_{13} \sim a_{22} \). However, if a virtual activity is defined as in Fig. 8(b), then \( a_{11}, a_{12}, a_{21} \), and \( a_{22} \) are viewed as an atomic unit since they are members of the same virtual activity. The ordering relation \( \text{virtual activity} > a_{13} \) infers an implied ordering relation \( a_{22} > a_{13} \). This implied ordering relation does not hold in the base process. Hence, the virtual activity must contain all activities in branches 1 and 2, as shown in Fig. 8(c), to preserve the original ordering relations.

The implied ordering relation \( a_{22} > a_{13} \) means that \( a_{13} \) must wait for the completion of \( a_{22} \) before it can start. Perhaps \( a_{13} \) is started after \( a_{11}, a_{12}, a_{21} \), and \( a_{22} \) are completed in the base process instance. In this situation, the virtual activity defined in Fig. 8(b) satisfies the atomicity property. However, a definition cannot assure the atomicity property; that is, the property may not hold for the other base process instances. If implied ordering relations conform to the original ordering relations, then the progress expressed in virtual activities necessarily satisfies the atomicity property.

The definition of a virtual activity must maintain repetitive execution order of a loop structure. In Fig. 9, for example, each numbered dotted rectangle is a possible virtual activity definition. While alternatives 1 and 2 are valid, alternatives 3 and 4 alter the original ordering relations. Alternative 3 creates an implied ordering relation \( a_3 > a_1 \); that is, \( a_1, a_2, \) and \( a_3 \) may be executed repetitively (\( a_1 \) and \( a_2 \) are viewed as an atomic unit). Likewise, alternative 4 creates an implied ordering relation \( a_4 > a_2 \); that is, \( a_4 \) may be executed before the condition that \( a_2 \) and \( a_3 \) be repetitively executed, is satisfied.
Fig. 10 seems to show a valid process-view definition because implied ordering relations derived from VP can be found in BP. However, the loop-derived sub-process-view \( VLP(\text{va}_2, \text{va}_3) \) does not preserve the ordering relations of the loop-derived sub-process \( LP(\text{a}_2, \text{a}_5) \). When \( a_2 \) is completed and \( a_3 \) is started in the iteration of \( lp(\text{a}_2, \text{a}_5) \), \( \text{va}_2 \) is not completed (because whether \( a_4 \) is fired is currently unknown) and \( \text{va}_3 \) is started in the iteration of \( vl(\text{va}_2, \text{va}_3) \). Atomicity is violated in this situation. The process-view must contain a loop structure that corresponds to the loop of the base process when a virtual activity of a process-view does not hide an entire loop of a base process, as stated in the atomicity rule. In such situations, to keep the atomicity property, the loop-derived sub-process-view must preserve the ordering relations of the corresponding loop-derived sub-process.

In Fig. 10, \( a_2 > a_3 \), \( a_3 > a_4 \) and \( a_4 > a_5 \) hold in \( LP(\text{a}_2, \text{a}_5) \). However, “\( \text{va}_2 > \text{va}_3 \) holds in \( VLP(\text{va}_2, \text{va}_3) \)” implies that \( a_2 > a_3 \), \( a_4 > a_3 \), \( a_2 > a_5 \), and \( a_4 > a_5 \). The implied ordering relation “\( a_4 > a_3 \)” does not hold in \( LP(\text{a}_2, \text{a}_5) \). Therefore, \( VLP(\text{va}_2, \text{va}_3) \) does not preserve the ordering relations of \( LP(\text{a}_2, \text{a}_5) \).

**Summary of order preservation rule.** If the ordering relation between two virtual activities is “\( > \)” in a process-view/loop-derived sub-process-view, then the implied ordering relation between the respective members of these virtual activities is also “\( > \)”. Corresponding inferences also hold for the ordering relations “\( < \)” and “\( \infty \)”, respectively. The implied ordering relations between the respective members of two virtual activities must hold in the corresponding base process/loop-derived sub-process.

Restated, (1) Given a process-view VP, derived from a base process BP, for any two virtual activities \( \text{va}_i \) and \( \text{va}_j \) in VP: “\( \text{va}_i > \text{va}_j \) holds in VP” implies that “\( a_x > a_y \)” for all \( a_x \) contained by \( \text{va}_i \) and all \( a_y \) contained by \( \text{va}_j \); moreover, “\( a_x > a_y \)” must hold in BP for all \( a_x \) contained by \( \text{va}_i \) and all \( a_y \) contained by \( \text{va}_j \). (2) Given a loop-derived sub-process-view VLP, derived from the corresponding loop-derived sub-process LP, for any two virtual activities \( \text{va}_i \) and \( \text{va}_j \) in VLP: “\( \text{va}_i > \text{va}_j \) holds in VLP” implies that “\( a_x > a_y \)” for all \( a_x \) contained by \( \text{va}_i \) and all \( a_y \) contained by \( \text{va}_j \); moreover, “\( a_x > a_y \)” must hold in LP for all \( a_x \) contained by \( \text{va}_i \) and all \( a_y \) contained by \( \text{va}_j \).
Statements (1) and (2) also hold for the ordering relations “<” and “≤”, respectively.

**Discussion.** The atomicity rule describes the operational semantic property of a process-view. The order preservation rule provides a syntactical constraint that ensures a process-view follows the atomicity property. This approach is called order-preserving because implied ordering relations, derived from a process-view/loop-derived sub-process-view, conform to the ordering relations of the base process/loop-derived sub-process. A *legal* virtual activity in an order-preserving process-view must follow above three rules. Therefore, the behavior of a process-view whose virtual activities are legal can be determined by the behavior of its base process.

Consider a situation in which a member activity ba of a virtual activity va is started. Therefore, va is started. The following elucidates that the preceding fired virtual activities of va are completed in a process-view of which each virtual activity follows the above three rules. Consider any fired virtual activity va′ with a higher order than va in the process-view. According to the order preservation rule, all member base activities of va′ must have a higher order than ba since va′ > va. In the base process, all base activities that precede ba have been evaluated and the fired ones are completed before ba is started since base activities are executed atomically. Thus, in the process-view, any virtual activity with a higher order than va can be determined as completed before ba is started because all of its fired member activities have been completed.

Contrarily, the starting of a member activity of an arbitrarily defined virtual activity that violates order preservation, cannot ensure that preceding virtual activities have been completed. Consider the same situation in which a member activity ba of a virtual activity va is started. The preceding virtual activities that violate order preservation contain some member base activities that do not have a higher order than ba. In the base process, all base activities with a higher order than ba are completed. However, base activities without a higher order than ba may be neither completed nor evaluated. Consequently, in the process-view, the virtual activities with higher order than va cannot be decided as completed because some of their member activities may not be completed before the starting of ba. Several virtual activities that are not ordering independent may be executed concurrently. The operational semantics of such a process-view violates the atomicity rule.

### 4.2. Formal model

The rules that a process-view should comply with have been introduced above. In the following, virtual activities and virtual (loop) dependencies in an order-preserving process-view are formally defined.

**Definition 10** (Virtual activity). Given a base process $BP = \langle BA, BD \rangle$, a virtual activity $va$ is a 5-tuple $\langle A, D, SPLIT\_flag, JOIN\_flag, SC \rangle$, where

1. $A$ is a non-empty set, and its members follow three rules:

   (a) Members of $A$ are base activities that are members of $BA$ or other previously defined virtual activities that are derived from $BP$. 

![Fig. 10. Illustrative example of order preservation in the loop structure.](image-url)
(b) The starting and completion of \(va\) are determined by the starting and completion of members of \(A\), according to the atomicity rule.

(c) Let \(BP' = \langle BA', BD' \rangle\) be \(BP\) or any loop-derived sub-process \(LP(e_i, e_x)\) where \(e_i, e_x \in BA\). For any \(x \in BA'\) and \(x \notin A\): if there exists a \(y \in A\) and \(y \in BA'\) such that \(x \succ y\) holds in \(BP'\), then \(x \succ z\) holds in \(BP\) for all \(z \in A\) and \(z \in BA'\); if there exists a \(y \in A\) and \(y \in BA'\) such that \(x \prec y\) holds in \(BP'\), then \(x \prec z\) holds in \(BP\) for all \(z \in A\) and \(z \in BA'\). That means the ordering relations between \(x\) and all members (base activities) of \(A\) that belong to \(BP'\) are identical.

2. \(D = \{dep(x, y, \ldots) | x, y \in A \text{ and } dep(x, y, \ldots) \in BD\}\).

3. \(SPLIT\_flag\) may be “NULL” or “MIX”. NULL suggests that \(va\) has a single outgoing virtual dependency while MIX indicates that \(va\) has multiple outgoing virtual dependencies.

4. \(JOIN\_flag\) may be “NULL” or “MIX”. NULL suggests that \(va\) has a single incoming virtual dependency while MIX indicates that \(va\) has multiple incoming virtual dependencies.

5. \(SC\) is the starting condition of \(va\).

The \(SPLIT\_flag\) and \(JOIN\_flag\) cannot simply be described as AND or XOR since \(va\) is an abstraction of a set of base activities that may be associated with different ordering structures. Therefore, MIX is used to abstract the complicated ordering structures. A WIMS evaluates \(SC\) to determine whether \(va\) can be started. Section 4.5 further discusses \(JOIN\_flag\), \(SPLIT\_flag\), and the derivation of \(SC\). Members of \(A\) are called \(va\’s member activities\), and members of \(D\) are called \(va\’s member dependencies\). To save space, the abbreviated notation \(va = \langle A, D \rangle\) is employed below to represent a virtual activity.

**Definition 11** (Virtual dependency). Let \(va_i = \langle A_i, D_i \rangle\) and \(va_j = \langle A_j, D_j \rangle\) be two distinct virtual activities that are derived from a base process \(BP = \langle BA, BD \rangle\). A virtual dependency from \(va_i\) to \(va_j\) is \(vdep(va_i, va_j, R, VC_{ij}) = \{dep(a_x, a_y, R, C_{xy}) | a_x \in A_i, a_y \in A_j\}\), and \(dep(a_x, a_y, R, C_{xy}) \in BD\}\), where the virtual condition \(VC_{ij}\) is a Boolean combination of \(C_{xy}\).

**Definition 12** (Virtual loop dependency). Let \(va_i = \langle A_i, D_i \rangle\) and \(va_j = \langle A_j, D_j \rangle\) be two distinct virtual activities that are derived from a base process \(BP = \langle BA, BD \rangle\). A virtual loop dependency from \(va_i\) to \(va_j\) is \(vdep(va_i, va_j, L, VLC_{ij}) = \{dep(a_x, a_y, L, LC_{xy}) | a_x \in A_i, a_y \in A_j\}\), and \(dep(a_x, a_y, L, LC_{xy}) \in BD\}\), where the virtual loop condition \(VLC_{ij}\) equals the loop condition \(LC_{xy}\).

Notably, a loop (and a loop-derived sub-process-view) in the process-view has a corresponding loop (and a loop-derived sub-process) in the base process, as a process-view has a corresponding base process (see the atomicity rule). Therefore, a virtual loop dependency of the process-view only contains one loop dependency of the base process. Section 4.5 further discusses the relationship between \(VC VLC\) and \(C(\text{LC})\) with respect to virtual (loop) dependency. In the following, Theorem 1 proves that the process-view, defined according to Definitions 10–12, preserves original ordering relations. The following lemmas support the proof of Theorem 1. These lemmas are proved in Appendix A.

**Lemma 1.** Consider a process-view \(VP = \langle V A, V D \rangle\), where members of \(VA\) follow Definition 10 and members of \(VD\) follow Definitions 11 and 12, derived from a base process \(BP = \langle BA, BD \rangle\). For any two distinct virtual activities \(va_i \in VP\), \(va_j \in VP\), and \(va_i = \langle A_i, D_i \rangle\) and \(va_j = \langle A_j, D_j \rangle\): if there exist a base activity \(a_i \in A_i\) and a base activity \(a_j \in A_j\) such that \(a_i > a_j\) holds in \(BP\), then \(va_i > va_j\) holds in \(VP\).

**Lemma 2.** Consider a process-view \(VP = \langle V A, V D \rangle\), where members of \(VA\) follow Definition 10 and members of \(VD\) follow Definitions 11 and 12, derived from a base process \(BP = \langle BA, BD \rangle\). Let \(LP\) be a loop-derived sub-process of \(BP\); let \(VLP\) be a loop-derived sub-process-view of \(VP\), \(VLP\) corresponds to \(LP\). For any two distinct
virtual activities $v_{ai}$ and $v_{aj}$ in VLP, $v_{ai} = \langle A_i, D_i \rangle$ and $v_{aj} = \langle A_j, D_j \rangle$: if there exist a base activity $a_i \in A_i$ and a base activity $a_j \in A_j$ such that $a_i > a_j$ holds in LP, then $v_{ai} > v_{aj}$ holds in VLP.

**Lemma 3.** Consider a process-view $VP = \langle VA, VD \rangle$, where members of $VA$ follow Definitions 10 and members of $VD$ follow Definitions 11 and 12, derived from a base process $BP = \langle BA, BD \rangle$. For any two distinct virtual activities $v_{ai}$ and $v_{aj}$ in $VP$, $v_{ai} = \langle A_i, D_i \rangle$ and $v_{aj} = \langle A_j, D_j \rangle$: if $v_{ai} > v_{aj}$ holds in $VP$, then there exist a base activity $a_i \in A_i$ and a base activity $a_j \in A_j$ such that $a_i > a_j$ holds in $BP$.

**Lemma 4.** Consider a process-view $VP = \langle VA, VD \rangle$, where members of $VA$ follow Definitions 10 and members of $VD$ follow Definitions 11 and 12, derived from a base process $BP = \langle BA, BD \rangle$. Let $LP$ be a loop-derived sub-process of $BP$; let $VLP$ be a loop-derived sub-process-view of $VP$: $VLP$ corresponds to $LP$. For any two distinct virtual activities $v_{ai}$ and $v_{aj}$ in VLP, $v_{ai} = \langle A_i, D_i \rangle$ and $v_{aj} = \langle A_j, D_j \rangle$: if $v_{ai} > v_{aj}$ holds in VLP, then there exist a base activity $a_i \in A_i$ and a base activity $a_j \in A_j$ such that $a_i > a_j$ holds in $LP$.

**Theorem 1.** Given a process-view $VP = \langle VA, VD \rangle$, as derived from a base process $BP = \langle BA, BD \rangle$, if members of $VA$ follow Definition 10 and members of $VD$ follow Definitions 11 and 12, then $VP$ preserves the ordering relations of $BP$, and any loop-derived sub-process-view $VLP$ preserves the ordering relations of its corresponding loop-derived sub-process $LP$.

**Proof.** Let $v_{ai}$ and $v_{aj}$ be two different virtual activities in $VP$, $v_{ai} = \langle A_i, D_i \rangle$ and $v_{aj} = \langle A_j, D_j \rangle$. The following proves that $VP$ preserves the ordering relations of $BP$: that is, the implied ordering relations between the respective members of $v_{ai}$ and $v_{aj}$ conform to the ordering relations of the base process $BP$.

**Case 1: $v_{ai} > v_{aj}$ holds in $VP$.** The proof needs to show that $a_i > a_j$ holds in $BP$ (according to atomicity rule) for all $a_i \in A_i$ and all $a_j \in A_j$. Given that $v_{ai} > v_{aj}$ holds in $VP$, by Lemma 3, let $a_i$ be the member base activity of $v_{ai}$ and $a_j$ be the member base activity of $v_{aj}$ (i.e., $a_i \in A_i$ and $a_j \in A_j$), and $a_i > a_j$ holds in $BP$. Since $a_i > a_j$, $a_i \not\in A_j$ and $a_j \not\in A_i$, by Definition 10(1c), $a_i > a_j$ for all $a_i \in A_i$. Furthermore, for any $a_j \in A_j$: since $a_i > a_j$, $a_i \not\in A_j$ and $a_j \not\in A_i$, thus $a_i > a_j$ for all $a_j \in A_j$ (by Definition 10(1c)). Therefore, $a_i > a_j$ holds in $BP$ for all $a_i \in A_i$ and all $a_j \in A_j$.

**Case 2: $v_{ai} < v_{aj}$ holds in $VP$.** The proof is similar to Case 1 and is omitted.

**Case 3: $v_{ai} \equiv v_{aj}$ holds in $VP$.** The proof needs to show that $a_i \equiv a_j$ holds in $BP$ for all $a_i \in A_i$ and all $a_j \in A_j$. Assume that the proposition is false; that is, $a_i \not\equiv a_j$ does not hold in $BP$ for some $a_i \in A_i$ and some $a_j \in A_j$. Let $a_i$ be a member base activity of $v_{ai}$ and $a_j$ be a member base activity of $v_{aj}$ (i.e., $a_i \in A_i$ and $a_j \in A_j$), such that $a_i > a_j$ (or $a_j < a_i$) holds in $BP$. By Lemma 1, if $a_i > a_j$ holds in $BP$, then $v_{ai} > v_{aj}$ holds in $VP$, which contradicts with $v_{ai} \equiv v_{aj}$. Similarly, if $a_j < a_i$ holds in $BP$, then $v_{aj} < v_{ai}$ holds in $VP$, which also contradicts with $v_{ai} \equiv v_{aj}$. Therefore, $a_i \not\equiv a_j$ holds in $BP$ for all $a_i \in A_i$ and all $a_j \in A_j$.

Lemmas 2 and 4 are used to prove that any loop-derived sub-process-view of $VP$ preserves the ordering relations of its corresponding loop-derived sub-process of $BP$. The proof of this claim is similar to the above proof and is thus omitted. □

4.3. Essential activity

Previous sections have introduced three rules that allow a virtual process to preserve the ordering relations of a base process. However, a process modeler merely wishes to conceal sensitive activities or aggregate detailed activities. In addition to these activities, what activities must be included in order to form a legal virtual activity is not a primary concern, and should be supported by a process-view definition tool. These sensitive or detailed activities are called essential activities.

**Definition 13 (Essential activity).** Before defining a virtual activity, a modeler must select some activities that are essential to that virtual activity.
The chosen activities are called essential activities, and form an essential activity set EAS.

Many virtual activities contain the same essential activities and conform to Definition 10. These virtual activities have a “cover” relation with each other, and can be found to share a “minimal virtual activity”.

**Definition 14 (Cover).** Let EAS be an essential activity set. A virtual activity \( v_a = \langle A_i, D_i \rangle \) is said to cover a virtual activity \( v_a = \langle A_j, D_j \rangle \) if and only if \( A_j \subseteq EAS, A_i \supseteq A_j \) and \( D_i \supseteq D_j \).

**Definition 15 (Minimal virtual activity).** For an essential activity set EAS, a virtual activity \( \langle A, D \rangle \) is called a minimal virtual activity, denoted by \( \text{min}_\text{va}(EAS) \), if it does not cover other virtual activities and \( A \supseteq EAS \).

Given an EAS, a modeler must identify \( \text{min}_\text{va}(EAS) \). Besides essential activities, A only contains those activities needed to preserve the original ordering relations of the base process, i.e., the \( \text{min}_\text{va}(EAS) \) only contains essential and adequate information to abstract EAS. Adding more activities which are neither modeler selected nor order preservation needed, into A merely adds unnecessary information into \( \text{min}_\text{va}(EAS) \).

The procedure of defining an order-preserving process-view is summarized as follows: A process modeler must initially select essential activities. The process-view definition tool then automatically generates a legal minimal virtual activity that encapsulates these essential activities. The above two steps are repeated until the modeler determines all required virtual activities. The definition tool then generates all virtual (loop) dependencies between these virtual activities as well as the ordering fields (JOIN/SPLIT_flag) and SC of each virtual activity. The process-view definition tool can be implemented using the algorithm proposed in Section 4.5.

### 4.4. Illustrative examples

This section uses a manufacturing process as shown in Fig. 11 of a thin film transistor-liquid crystal display (TFT-LCD) company, named HiTEC, to demonstrate the applications of process-view.

The manufacturing process can be divided into three parts: Array, Cell, and Module. Initially, each cassette contains a load of glass substrates. The Array part, activities (1)–(14) in Fig. 11, produces TFT panels after repeated sputtering, stepping, developing, and etching on glass substrates. Next, TFT panels move to the Cell part. The Cell part, activities (16)–(40), attaches color filters to TFT panels, breaks them into pieces, and injects liquid crystal to produce LC cells. The Module part, activities (42)–(63), assembles LC cells, flexible printed circuits, chips, and casing to produce LCD modules. Finally, these LCD modules are delivered to downstream customers such as notebook computer and monitor manufacturers. Delivery activities, (41) and (64), are outsourced to a transport company.

HiTEC has three fabrication factories: FAB I, II, and III. FAB I and II produce Array and Cell Parts, while FAB III handles Module parts. According to the demands of different roles, a process modeler can design an appropriate view for a role without being restricted by the original process definitions. The following discussion provides two examples of the roles: a factory director and a marketer.

First, the application of the proposed methodology to defining a process-view is demonstrated. The factory director of FAB II must know the aggregated information and more about the Array and Cell processes. A process modeler can easily use the order-preserving approach to define a process-view for the director. For example, when the modeler wants to use a virtual activity to abstract activities (16), (19), (20), (22), and (25), the process-view definition can automatically derive a legal virtual activity. Fig. 12 displays the prototype system when generating a \( \text{min}_\text{va}(EAS) \), where \( EAS = \{16, 19, 20, 22, 25\} \). The tool automatically generates a \( \text{min}_\text{va}(EAS) \) based on the chosen EAS. Following the determination of all virtual activities, i.e., each base activity is contained by a virtual activity, the tool automatically generates a process-view, as shown in Fig. 13 where each virtual activity is annotated with its
member activity(s) in the braces. The process-view ensures the preservation of the original ordering relations.

Marketers must monitor the progress of the manufacturing process to improve their provision of customer service. The process-view shown in Fig. 11 informs marketers of the status of order fulfillment in the manufacturing department, for example, the Cell activity represents the progress of activities (16)–(40) in Fig. 11. Another scenario is HiTEC authorizing its customers to access process-views. Customers can actively monitor the progress of order processing through their process-views. A process modeler can use diverse customer requirements as a basis for designing various customized process-views to provide personalized service.

4.5. Algorithm

This section introduces algorithms to derive an order-preserving process-view. The algorithm first derives the member activities and dependencies of a minimal virtual activity based on the essential
activities specified by a modeler. Subsequently, how to derive virtual dependencies and the JOIN flag, SPLIT flag, and SC field of each virtual activity in the process-view is discussed.

4.5.1. Minimal virtual activity generator

For a given essential activity set $EAS$, Fig. 15 shows the algorithm capable of obtaining an $\text{min}_{\text{va}}(EAS) = \langle A, D \rangle$. Because $D$ can be derived from $A$ and $EAS$ is known, the members of $A$ must be identified. Let $BP'$ be the base process $BP$ or any loop-derived sub-process of $BP$. According to the definition of a virtual activity (Definition 10(1c)), a legal virtual activity $A$ must satisfy the order-preserving condition: the ordering relations between $x$ and all members of $A$ that belong to $BP'$ are identical for any base activity $x \in BA'$ and $x \notin A$.

As mentioned in Section 4.3, an activity $x$, $x \notin EAS$, is included in $A$ for order preservation. $EAS$ is obviously a starting point for identifying $x$. The algorithm begins from $EAS$, by initializing an activity set $TAS$ that equals $EAS$, to check whether $TAS$ is a legal (i.e., order-preserving) virtual
activity. If TAS is not legal, TAS is updated by including activities that violate the order-preserving condition. To determine which of the activities should be added into TAS to form a legal and minimal virtual activity, the algorithm considers the activities that are adjacent to members of TAS:

```plaintext
1) procedure VAGenerator (base process BP = (BA, BD), essential activity set EAS)
2) begin
3)   temp activity set TAS = EAS
4)   repeat
5)     an activity set TAS1 = TAS
6)     adjacent activity set AAS = \{ x | x, y ∈ BA, x ≠ TAS, y ∈ TAS and dep(x, y, _, _) ∈ BD \}
7)     a process set PS = { p | p is BP or a loop-derived sub-process of BP }
8)     while AAS is not empty do
9)       select an activity x from AAS
10)      remove x from AAS
11)     if ∃ y, z ∈ TAS, ∃ BP’ = (BA’, BD’) ∈ PS, and x, y, z ∈ BA’ such that (( x > y holds in BP’ but x > z does not holds in BP’ ) or ( x < y holds in BP’ but x < z does not holds in BP’ ) or ( x = y holds in BP’ but x = z does not holds in BP’ ))
12)     * The ordering relations between x and all base activities of TAS that belong to BP’ are not identical */
13)       then add x to TAS
14)     end if
15)     end while
16) until TAS == TAS1
17) an activity set A = TAS
18) a dependency set D = \{ dep(x, y, _, _) | x, y ∈ A, and dep(x, y, _, _) ∈ BD \}
19) min_va(EAS) = (A, D)
20) end
```

Fig. 15. The algorithm of a minimal virtual activity generator.

Since this virtual activity conforms to Definition 10, it is a legal virtual activity. Moreover, the algorithm checks the ordering relations from adjacent activities, creating a minimal virtual activity. The proof is shown in Theorem 2.

**Theorem 2.** Given a base process BP and an essential activity set EAS, the virtual activity va, as generated by the algorithm VAGenerator, is a legal (order-preserving) virtual activity, and also a minimal virtual activity of EAS.

**Proof.** The procedure determines whether adjacent activities of TAS satisfy the order-preserving condition. TAS is updated during the while loop (lines 8–15), by adding adjacent activities that violate the order-preserving condition. If TAS is updated, the repeat-until loop is repeated to check the order-preserving condition. The repeat-until loop (lines 4–16) continues to repeat until no more adjacent activity is added into TAS (line 16, TAS == TAS1), i.e., all adjacent activities of TAS satisfy the order-preserving condition. Finally, TAS is a legal virtual activity and A is set to equal TAS.

Following the determination of A, the members of D are those dependencies whose succeeding and preceding activity are both members of A (Definition 10(2)). The minimal virtual activity of EAS, \( \text{min}_va(EAS) \), equals \( \langle A, D \rangle \).
x is not adjacent to any members of A, such that x violates the order-preserving condition. Let y_1 and y_2 be the member activities of A that cause the violation of the condition.

**Case 1:** \( x > y_1 \) holds in BP, but \( x > y_2 \) does not hold in BP. Given that \( x > y_1 \) and x is not adjacent to any activity in A, there must exist a path from x to z and a path from z to y_1, where z is an adjacent activity of A. Thus, \( x > z \) and \( z > y_1 \) hold in BP. \( z > y_1 \) implies that \( z > y_2 \) also holds in BP since z satisfies the order-preserving condition. It further implies that \( x > y_1 \) and \( x > y_2 \) hold in BP, which contradicts the assumption that \( x > y_1 \) holds but \( x > y_2 \) does not hold in BP.

**Case 2:** \( x < y_1 \) holds in BP, but \( x < y_2 \) does not hold in BP. The proof is similar to Case 1 and is omitted.

**Case 3:** \( x \propto y_1 \) holds in BP, but \( x \propto y_2 \) does not hold in BP. Given that \( x \propto y_2 \) does not hold in BP, assume that \( x > y_2 \) holds in BP. There exist a path from x to z and a path from z to y_2, where z is an adjacent activity of A, since \( x > y_2 \) and x is not adjacent to any activity in A. Thus, \( x > z \) and \( z > y_2 \) hold in BP. \( z > y_2 \) implies that \( z > y_1 \) also holds in BP, since z satisfies the order-preserving condition. It further implies that \( x > y_1 \) and \( x > y_2 \) hold in BP, which contradicts the assumption.

The proof of the preservation of the ordering relations held in the loop-derived sub-processes of BP is similar to the above proof and is omitted.

Thus, the generated virtual activity \( va = \langle A, D \rangle \) conforms to Definition 10.

Next, the following proves that va is a minimal virtual activity of EAS. Assume that va is not minimal, then a legal and minimal virtual activity \( mva = \langle A_m, D_m \rangle \) exists such that \( A \supseteq A_m \). The procedure begins from EAS, by initializing TAS to be EAS, to check the ordering relations of the adjacent activities. TAS is updated during the while loop (lines 8–15), by adding adjacent activities that violate the order-preserving condition. Assume that a TAS is formed during the while loop of the procedure, where \( TAS \subset A_m \). Additionally, an adjacent activity x of TAS, \( x \notin A_m \), can be found such that x violates the order-preserving condition. Notably, an adjacent activity is added into TAS according to the if-condition (line 11). If the adjacent activity belongs to \( A_m \), then the process proceeds until an adjacent activity that does not belong to \( A_m \) can be found; otherwise, the procedure generates \( A_m \). Since the ordering relations between x and the members of TAS violate the order-preserving condition, there exist two activities \( y_1 \) and \( y_2 \) in TAS, such that the ordering relation between x and \( y_1 \) differs from the ordering relation between x and \( y_2 \) (the ordering relation means the ordering relation held in BP or a loop-derived sub-process of BP). Moreover, \( y_1 \) and \( y_2 \) belong to \( A_m \) since \( TAS \subset A_m \). Thus, the ordering relations between x and members of \( A_m \) also violate the order-preserving condition. Consequently, \( mva \) is not a legal virtual activity, which contradicts the assumption. Therefore, \( va \) is a minimal virtual activity. □

**Example 1.** This example illustrates the steps required to generate \( min_{va}(EAS) \), where \( EAS = \{a_{16}, a_{19}, a_{20}, a_{22}, a_{25}\} \) and the base process BP is the LCD manufacturing process described in Section 4.4. Part of the LCD process shown in Fig. 11 is redrawn in Fig. 16(a). The algorithm creates a copy of EAS, i.e., TAS, as an initial set (see line 3 in Fig. 15). The repeat-until loop uses TAS as a seed to identify the member activities of \( min_{va}(EAS) \) (see lines 4–16).

![Fig. 16. Two process examples.](image-url)
Initially, \( AAS = \{a_{15}, a_{17}, a_{18}, a_{21}, a_{23}, a_{24}, a_{26} \} \) (see line 6). This example only need to consider the ordering relations of \( BP \) since it does not contain loops (see line 7). The subsequent while loop verifies whether the adjacent activities satisfy the order-preserving condition (see lines 8–15). For example, activity \( a_{22} \) is added into \( TAS \) since \( a_{23} < a_{22} \) but \( a_{22} \not< a_{25} \). However, activity \( a_{15} \) is not added into \( TAS \) since the ordering relations between \( a_{15} \) and all members of \( TAS \) are “>”. After checking all members of \( AAS \), activities \( a_{17}, a_{18}, a_{21}, a_{24}, a_{23}, \) and \( a_{26} \) are added into \( TAS \).

Since \( TAS \) is updated (\( TAS \neq TAS_{1} \)), i.e., \( TAS \) does not follow the definition of virtual activity, the repeat-until loop repeats using the revised \( TAS \), \( \{a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23}, a_{24}, a_{25}, a_{26} \} \) (see line 16). During the next iteration, \( AAS = \{a_{15}, a_{27} \} \). \( TAS \) remains unchanged during the while loop since the members of \( AAS \) satisfy the order-preserving condition. Under the circumstances, the ordering relations between \( x \) and all members of \( TAS \) are identical in \( BP \) for any \( x \in BA \) and \( x \notin TAS \), i.e., \( TAS \) follows the definition of a virtual activity, and the repeat-until loop stops.

For the \( \text{min}_va(EAS) = \{A, D\} \), \( A \) equals \( TAS \) and the dependency set \( D \) contains the following dependencies: \( \text{dep}(a_{16}, a_{17}, R, \_), \text{dep}(a_{17}, a_{18}, R, \_), \text{dep}(a_{18}, a_{19}, R, \_), \text{dep}(a_{19}, a_{20}, R, \_), \text{dep}(a_{20}, a_{21}, R, \_), \text{dep}(a_{21}, a_{22}, R, \_), \text{dep}(a_{22}, a_{23}, R, \_), \text{dep}(a_{23}, a_{24}, R, \_), \text{dep}(a_{24}, a_{25}, R, \_), \text{dep}(a_{25}, a_{26}, R, \_) \) (see lines 18–19).

**Example 2.** This example illustrates how to derive the \( \text{min}_va(EAS) \) for the base process \( BP \) as shown in Fig. 16(b) and \( EAS = \{a_{5}, a_{7} \} \). Initially, \( AAS = \{a_{4}, a_{6}, a_{8} \} \), \( TAS = \{a_{5}, a_{7} \} \), and \( PS = \{BP, LP(a_{3}, a_{6}) \} \). \( a_{6} \) is added into \( TAS \) because \( a_{5} < a_{6} \) holds in \( BP \) but \( a_{5} > a_{6} \) does not hold in \( BP \). \( a_{8} \) is not added to \( TAS \) because \( a_{7} > a_{6} \) and \( a_{6} > a_{9} \) hold in \( BP \). Notably, \( a_{8} \) is added into \( TAS \) because \( a_{4} < a_{5} \) (derived from the path \( a_{5}, a_{6}, a_{2}, a_{4} \)) holds in \( BP \) but \( a_{4} < a_{7} \) does not hold in \( BP \). Therefore, \( TAS \) changes to \( \{a_{4}, a_{5}, a_{6}, a_{7} \} \) and repeat-until loop repeats. During the second iteration, \( AAS = \{a_{3}, a_{5} \} \). \( a_{5} \) is added into \( TAS \) because \( a_{4} > a_{5} \), \( a_{5} > a_{6} \), \( a_{5} > a_{7} \), and \( a_{7} > a_{8} \) hold in \( BP \). However, \( a_{3} \) is added into \( TAS \) because \( a_{3} < a_{6} \) (derived from the path \( a_{6}, a_{3} \)) holds in \( BP \) but \( a_{3} > a_{7} \) does not hold in \( BP \). Therefore, \( TAS \) is updated to \( \{a_{3}, a_{4}, a_{5}, a_{6}, a_{7} \} \) and the repeat-until loop repeats. During the third iteration, \( AAS = \{a_{1}, a_{2}, a_{4} \} \). TAS remains unchanged during the while loop since the ordering relations between each adjacent activity and the members of \( TAS \) are identical in \( BP \) and \( LP(a_{3}, a_{6}) \). Consequently, the repeat-until loop stops and \( A = \{a_{3}, a_{4}, a_{5}, a_{6}, a_{7} \} \), \( D = \{\text{dep}(a_{3}, a_{4}, R, \_), \text{dep}(a_{4}, a_{5}, R, \_), \text{dep}(a_{5}, a_{6}, R, \_), \text{dep}(a_{6}, a_{7}, R, \_), \text{dep}(a_{7}, a_{8}, R, \_) \} \).

**Example 3.** Given a base process \( BP \) as shown in Fig. 16(b) and \( EAS = \{a_{4}, a_{6} \} \), the generation of the min va(EAS) is explained below. Initially, \( AAS = \{a_{5}, a_{7} \} \), \( TAS = \{a_{4}, a_{6} \} \), and \( PS = \{BP, LP(a_{3}, a_{6}) \} \). According to the ordering relations of \( BP \), \( TAS \) is unchanged (for \( a_{5}: a_{5} > a_{4} \), \( a_{3} > a_{6} \), \( a_{5} < a_{4} \), \( a_{5} > a_{6} \)), \( a_{5} < a_{4} \), \( a_{5} < a_{6} \); for \( a_{5}: a_{5} > a_{4} \), \( a_{3} > a_{6} \), \( a_{5} < a_{4} \), \( a_{5} < a_{6} \); for \( a_{7}: a_{7} < a_{4} \), \( a_{7} < a_{6} \). However, according to the ordering relations of \( LP(a_{3}, a_{6}) \), \( a_{5} \) is added into \( TAS \) (for \( a_{5}: a_{5} > a_{4} \), \( a_{3} > a_{6} \); for \( a_{5}: a_{5} < a_{4} \), \( a_{5} > a_{6} \)). Therefore, \( TAS \) is updated to \( \{a_{4}, a_{5}, a_{6} \} \) and the repeat-until loop repeats. During the second iteration, \( AAS = \{a_{3}, a_{7} \} \). \( TAS \) remains unchanged during the while loop since the ordering relations of \( BP \) and \( LP(a_{3}, a_{6}) \) are preserved. Hence, for the min va(EAS), \( A = \{a_{4}, a_{5}, a_{6} \} \) and \( D = \{\text{dep}(a_{4}, a_{5}, R, \_), \text{dep}(a_{5}, a_{6}, R, \_) \} \).

### 4.5.2 Virtual dependency and virtual loop dependency

Following the generation of all virtual activities, the process-view definition tool derives virtual dependencies by Definition 11. First, the members of a virtual dependency must be determined, after which the VC field of each virtual dependency must be derived.

Given the virtual activity set \( VA \) of a process-view \( VP \) derived from a base process \( BP \), whether or not a virtual dependency is associated with two virtual activities can be determined. For any two distinct virtual activities \( va_{i} = \langle A_{i}, D_{i} \rangle \) and \( va_{j} = \langle A_{j}, D_{j} \rangle \), if \( \exists a_{s} \in A_{i} \) and \( \exists a_{t} \in A_{j} \) such that \( \text{dep}(a_{s}, a_{t}, R, C_{xy}) \) exists in \( BP \), then \( \text{vdep}(va_{i}, va_{j}) \).
\(va_i, R, VC_{ij}\) exists in VP and \(dep(a_x, a_y, R, C_{xy})\) is a member of \(vdep(va_i, va_j, R, VC_{ij})\). After checking each base dependency, all virtual dependencies and their members can be derived.

The following illustrates the derivation of VC field of a virtual dependency. The \(JOIN\_flag\) of a base activity determines how the conditions of incoming dependencies are combined to derive starting condition of the base activity. Therefore, for the members of a virtual dependency, the conditions of base dependencies that share the same succeeding base activity are combined using the \(JOIN\_flag\) of that succeeding base activity. According to the atomicity rule, a virtual activity is started if one member activity is started. Therefore, the conditions derived from different succeeding base activities are then combined using Boolean OR. Restated, given two virtual activities succeeding base activities are then combined using \(JOIN\_flag\) of \(vdep\). Let \(C_{m, yk}\) represent the condition of a dependency from a member of \(A_i\) to \(a_{yk}\), for \(m = 1, 2, \ldots, n\): Let (1) \(C_{m, yk}\) represent the condition of a dependency from a member of \(A_i\) to \(a_{yk}\), for \(m = 1, 2, \ldots, n\); (2) \(f_k\) be the \(JOIN\_flag\) of \(a_{yk}\); (3) \(C_{y,k}\) represent the joined condition of all dependencies from members of \(A_i\) to \(a_{yk}\), i.e., \(C_{y,k} = (C_{1,y,k}f_kC_{2,y,k} \ldots f_kC_{h,y,k})\). For the virtual dependency \(vdep(va_i, va_j, R, VC_{ij})\), \(VC_{ij} = (C_{y,1} OR C_{y,2} \ldots OR C_{y,n})\).

In Fig. 17(a), for example, \(VC_{12} = (C_1 OR C_2)\), i.e., if \(C_1\) or \(C_2\) is true, then \(VC_{12}\) is true. However, in Fig. 17(b), \(C_1\) and \(C_2\) are combined using the \(JOIN\_flag\) of \(a_{2}\), i.e., \(VC_{12} = (C_1 AND/XOR C_2)\). For the complex combination shown in Fig. 17(c), \(VC_{12} = ((C_1 AND/XOR C_2) OR (C_2 AND/XOR C_4))\). In Fig. 17(d), \(VC_1 = (C_1 AND/XOR C_2), VC_2 = (C_2 AND/XOR C_4), VC_3 = C_5\).

Deriving virtual loop dependencies is similar to the derivation of virtual dependency. Given the virtual activity set \(VA\) of a process-view VP derived from \(BP\), for any two distinct virtual activities \(va_i = \langle A_i, D_i, L, VC_{ij}\rangle\) and \(va_j = \langle A_j, D_j\rangle\): if \(\exists a_x \in A_i\) and \(\exists a_y \in A_j\) such that \(dep(a_x, a_y, L, LC_{xy})\) exists in \(BP\), then \(vdep(va_i, va_j, L, VLC_{ij})\) exists in VP. According to Definition 12, \(VLC_{ij} = LC_{xy}\).

4.5.3. Ordering structure and starting condition

Following the generation of all virtual dependencies, the ordering fields (\(JOIN\_SPLIT\_flag\)) and starting conditions of each virtual activity can be derived. If a virtual activity has a single outgoing virtual dependency, its \(SPLIT\_flag\) is NULL. Otherwise, when multiple outgoing virtual dependencies exist, the \(SPLIT\_flag\) of the virtual activity is MIX. The \(SPLIT\_flag\) of the virtual activity cannot simply be AND or XOR, since a virtual activity abstracts a set of base activities that may be concurrently associated with AND-SPLIT and XOR-SPLIT.

Similarly, if a virtual activity has a single incoming virtual dependency, its \(JOIN\_flag\) is NULL, while if it has multiple incoming virtual dependencies, its \(JOIN\_flag\) is MIX. For a base activity, the \(JOIN\_flag\) determines the relationship between its starting condition (SC) and the conditions (C) of its incoming base dependencies.

Fig. 17. Four examples of virtual dependencies.
Nevertheless, for a virtual activity, MIX-JOIN abstracts the existence of different join structures in its member base activities. Therefore, the starting condition (SC) of a virtual activity cannot simply use the JOIN_flag to combine the conditions (VC) of incoming virtual dependencies. MIX-SPLIT/JOIN is used to represent multiple path structures, and the starting of a virtual activity depends on the SC field, which is derived as follows.

A virtual activity va is started if one of its member activities is started (atomicity rule). Therefore, the starting condition of each member activity of va must be determined, after which the SC of va equals the Boolean OR combination of the starting condition of each member activity. If va = ⟨A, D⟩, A = {a₁, a₂, ..., aₙ}, and the starting condition of ai is SC(ai) for i = 1…n, then the starting condition of va is SC(va) = (SC(a₁) OR SC(a₂)…OR SC(aₙ)). Consequently, SC(va) is true if the starting condition of one member activity is satisfied.

In Fig. 18(a), for example, VC₁ = C₁ and VC₂ = C₂, SC(a₁) = C₁ and SC(a₃) = C₂, SC(va₂) = SC(a₂) and SC(va₃) = SC(a₃). Moreover, in Fig. 18(b), VC₁ = (C₁ AND/XOR C₃) and VC₂ = (C₂ AND/XOR C₄). Whether a₂ can be started depends on its JOIN_flag (AND/XOR), i.e., SC(a₂) = (C₁ AND/XOR C₃). SC(va₂) = SC(a₂), SC(a₃) = (C₂ AND/XOR C₄). SC(va₃) = SC(a₃). In Fig. 19, if the JOIN_flag of a₄ is AND and the JOIN_flag of a₄ is XOR, then VC₁ = (C₁ OR C₂) and VC₂ = (C₃ OR C₄). SC(a₄) = (C₁ AND C₃) and SC(a₄) = (C₂ XOR C₄). SC(va₄) = SC(a₄) OR SC(a₄) = ((C₁ AND C₃) OR (C₂ XOR C₄)). Under such circumstances, the JOIN_flag of va₄ cannot merely be AND or XOR. MIX is used to abstract such combinations and a WfMS evaluates SC to trigger va₄.

4.5.4. Prototype system

A prototype system is currently being implemented based on the above discussion. As displayed in Fig. 12, the tool can automatically generate a legal and minimal virtual activity based on the essential activities chosen by a process modeler. Once all base activities have been contained by virtual activities, the tool automatically generates an order-preserving process-view based on the defined virtual activities. Currently, a modeler must use the prototype system to define a base process. In the future, the prototype will be enhanced to interpret base processes that are defined by using commercial products.

5. Related work

Workflow models, in which business processes are formally described to generate process definitions, are generally classified as communication-based and activity-based [1]. The former attempts to identify process objectives, while the latter focuses on identifying process structures. A survey [1] of commercial WfMSs revealed that most supported workflow models are activity-based.
The activity-based approach is easily understandable when modeling business tasks and their ordering relations. Owing to the top-down decomposition of a process, activity-based methods yield different hierarchical abstractions. However, as mentioned in Section 1, these hierarchical abstractions cannot provide each organizational level and unit with an appropriate view of a process. This work contributes to introducing the notion of view into WFMSs, and proposing a systematic procedure to derive adequate process abstractions from a base process for different participants.

This work enhances the capability of process abstraction in conventional activity-based process models, while the enhancement of the activity-based approach has also attracted considerable interest. Since process modeling can be considered from various aspects, such as functional and information aspects [16,17], numerous investigations have enhanced the activity-based approach by combining it with other aspects. These investigations have focused on specifying the interrelationships among these aspects. For example, Gruhn [18] proposed a model to integrate the modeling of activity, data, and organization. ARIS [19] describes business processes from the aspects of function, organization, data, output, and control. Moreover, some studies have exploited object-oriented technology or Petri nets to combine the modeling of control and data flow [20–23]. Such enhancements concern the integration of multiple aspects in a workflow model. However, this work focuses on deriving abstracted processes for different organizational roles. Notably, the term “view” used in some of above works represents an aspect of process modeling, while herein it represents a process abstraction.

Several formal process modeling techniques include the notion of process abstraction, such as process algebras and Petri-Nets [8–11]. These models can define some activities as silent activities (also called τ activities) that are not observable. By renaming activities to silent activities, the desired abstraction can be obtained. In contrast, process-views are derived through the bottom-up aggregation of activities to provide various levels of abstraction of a process. Conventional abstraction may be considered as partial abstraction since it provides partial observability of a process. Relative to partial abstraction, the proposed approach is considered to be aggregate abstraction, since it provides adaptable granularity of a process via bottom-up aggregation, i.e., a virtual activity may represent an aggregation of a set of base activities.

Partial abstraction does not reveal the progress status of the silent activities of a base process. For example, if a view “a₁ → a₃ → a₅ → a₈” is partially abstracted from the process shown in Fig. 16(b), then the view does not expose the progress of silent activities (i.e., a₂, a₄, a₆, and a₇). However, if the proposed approach defines a process-view as “{a₁, a₂} → {a₃} → {a₄, a₅, a₆} → {a₇, a₈}”, the progress of the virtual activity which contains a₁ and a₂ expresses the aggregated progress of these two base activities. In addition to concealing sensitive details that partial abstraction can provide, process-views also provide high-level managers with aggregated information on a desired process. Furthermore, a process-view can be systematically derived from a base process. Nevertheless, partial abstraction may assist a process modeler in identifying essential activities needed to define a process-view.

Aggregate abstraction generally includes partial abstraction. For example, given a virtual activity va₁ which is an aggregation of base activities a₁, a₂, and a₃, if the weight of a₁ and a₃ are zero and the weight of a₂ is one, then va₁ can be viewed as a partial abstraction of a₁, a₂, and a₃. The binary weight implies the notion of visible/invisible or important/unimportant that is the core of partial abstraction. Thus, partial abstraction can be derived from aggregate abstraction by using the concept of weight.

Various activity-based process models have been suggested for workflow management, such as the WFMC process model [7] and the Petri-Net variants [24,25]. Although not formally specifying the operational semantics of process definitions as Petri-Net-based workflow models [25], the WFMC process model has been extensively used to design and implement WFMSs. Besides, much workflow-related research is based on the WFMC process model. Accordingly, this work revises the WFMC process model to design the process-view model. Consequently, the process-view model can be
further extended to be incorporated into commercial products, since WfMC standards are accepted by major workflow vendors. This work focuses on enhancing the WfMC process model to derive process-views, i.e., aggregate process abstractions. Deriving the aggregate process abstractions for other activity-based models such as the Petri-Net variants is worth exploring. However, such a study is beyond the scope of this work and is proposed as a topic of future work.

Van der Aalst [26] proposed a novel generic workflow model to provide a manager with an aggregated view of variants for the same workflow process. Dynamic change has created multiple variants of the same process. A representative process, in which each activity represents the aggregation of all identical activities of these process variants, is used as the aggregated view. The generic process model focuses on providing aggregated information of dynamically changing process variants, while the process-view proposed herein aggregates different activities within a process to create various abstracted views.

Effective management of collaborative processes in virtual enterprises is important [27,28]. Related interorganizational workflow models, e.g., [29,30], achieved information concealment (autonomy) but were unable to monitor the progress of other cooperating organizations. Georgakopoulos et al. [3] used a service activity to abstract an entire process of a service provider. A service activity implements a service interface that defines several application-specific states and operations. Application-specific activity states extend the generic activity states defined by WfMC [31] to express possible process states. A service consumer is only aware of the state transition diagram of a service activity (information concealment), and can use the operations and states defined by the service interface to monitor the progress of a service provider’s process (progress monitoring). In the proposed approach, a modeler can define various process-views to achieve different levels of information concealment. Furthermore, the progress of base processes can be monitored through virtual activities and virtual processes. Notably, this work focuses on illustrating the process-view model and the novel approach to derive a process-view. This work on process-view will be further extended in the future to support information concealment and progress monitoring of collaborative processes in virtual enterprises.

6. Conclusion and future work

This work proposes a novel concept of process abstraction: process-view. Process-view enhances the conventional activity-based model to satisfy the diverse needs for obtaining abstracted process information. A process modeler can easily use a process-view definition tool to provide numerous views of a business process for different levels, divisions, and enterprises. The process-view achieves information abstraction and progress monitoring. Each role can obtain adequate information on a business process by setting up a role-related process-view, thereby facilitating hierarchical coordination within an organizational unit and horizontal coordination across multiple organizations (internal or external). The proposed approach increases the flexibility and functionality of current WfMSs.

Moreover, given the importance of execution order in business processes, this work also proposes an order-preserving approach to construct a process-view that preserves the original execution order of its base process. The proposed algorithm automatically derives minimal virtual activities and related virtual dependencies to generate an order-preserving process-view. The algorithm assists vendors in implementing process-view definition tools in their commercial systems. A real world example is used to demonstrate the feasibility of applying process-views.

According to the atomicity rule, the execution state of a virtual activity is either started or completed. However, a base activity may have more semantic states such as suspended or aborted (i.e., abnormal completion) state. Determining the state of a virtual activity is difficult, for example, if the current states of its three member activities are suspended, started, and aborted. This problem becomes more complicated when a process modeler is allowed to define more application-specific
activity states, such as CMI approach [3]. To express the progress information of process-views semantically, the issue of state abstraction requires further investigations.

This work currently defines the produced/consumed data of a virtual activity as a set of the produced/consumed data of member activities. However, according to the demands of different roles, a process modeler may wish to define process-view relevant data as an aggregation of base process relevant data (e.g., summation or average), or a selected portion of base process relevant data. Furthermore, process relevant data may have various semantic meanings and definitions. For example, a sales order or an insurance contract have various semantics in different steps of their processing workflow. Abstracting process relevant data from these aspects is a difficult problem, and future enhancement should provide more advanced abstraction of process-view data.

Acknowledgements

This research was supported by the National Science Council of the Republic of China under contract No. NSC 88-2416-H-009-023-N9 and NSC-89-2416-H-009-041. The authors would like to thank Dr. Churn-Jung Liau (Institute of Information Science, Academic Sinica, Taiwan) for his valuable suggestions on the notations and proofs of this paper. In addition, the authors would like to thank the anonymous referees for their valuable comments.

Appendix A. Proof of lemma

Lemma 1. Consider a process-view \( VP = \langle VA, VD \rangle \), where members of \( VA \) follow Definition 10 and members of \( VD \) follow Definitions 11 and 12, derived from a base process \( BP = \langle BA, BD \rangle \). For any two distinct virtual activities \( va_i \) and \( va_j \) in \( VP \), \( va_i = \langle A_i, D_i \rangle \) and \( va_j = \langle A_j, D_j \rangle \): if there exist a base activity \( a_i \in A_i \) and a base activity \( a_j \in A_j \) such that \( a_i > a_j \) holds in \( BP \), then \( va_i > va_j \) holds in \( VP \).

Proof. Let \( va_i \) and \( va_j \) be two distinct virtual activities in \( VP \), \( va_i = \langle A_i, D_i \rangle \) and \( va_j = \langle A_j, D_j \rangle \), where \( va_i \) contains a base activity \( a_i \) and \( va_j \) contains a base activity \( a_j \) (i.e., \( a_i \in A_i \), \( a_j \in A_j \)), and \( a_i > a_j \) holds in \( BP \). A path \( p \) from \( a_i \) to \( a_j \) must exist, since \( a_i > a_j \) holds in \( BP \). The following proves this lemma by induction on the length of path \( p \) from \( a_i \) to \( a_j \).

When the length of the path \( p \) from \( a_i \) to \( a_j \) is one, \( \text{dep}(a_i, a_j, \ldots) \) exists in \( BP \). According to Definitions 11 and 12, \( \text{vdep}(va_i, va_j, \ldots) \) exists in \( VP \), and thus \( va_i > va_j \) holds in \( VP \).

The induction hypothesis assumes that for any two distinct virtual activities \( va_r \) and \( va_l \) in \( VP \), \( va_r = \langle A_r, D_r \rangle \) and \( va_l = \langle A_l, D_l \rangle : va_r > va_l \) holds in \( VP \) if there exist a base activity \( a_r \in A_r \), a base activity \( a_l \in A_l \), and a path from \( a_i \) to \( a_j \) with length \( \leq k \) such that \( a_i > a_j \) holds in \( BP \).

The induction step must show that \( va_i > va_j \) holds in \( VP \), when the length of the path \( p \) from \( a_i \) to \( a_j \) is \( \leq k+1 \). Let \( a_l \) be a base activity on the path \( p \) from \( a_i \) to \( a_j \), where \( a_l \neq a_i \) and \( a_j \). The length of the subpath of \( p \) from \( a_i \) to \( a_l \) is \( \leq k \) since the length of \( p \) is \( \leq k+1 \). Besides, the length of the subpath of \( p \) from \( a_l \) to \( a_j \) is \( \leq k \). Moreover, \( a_i > a_l \) holds in \( BP \) and \( a_l > a_j \) holds in \( BP \). If \( a_i \in A_l \) or \( a_j \in A_l \), then \( va_i > va_j \) holds in \( VP \). On the other hand, if \( a_l \) is a member of \( va_l \), where \( va_l \in VA \), \( va_l \neq va_i \) and \( va_j \), then \( va_i > va_l \) holds in \( VP \) according to the induction hypothesis, since \( a_i > a_l \) holds in \( BP \) and the length of the path from \( a_i \) to \( a_l \) is \( \leq k \). Similarly, \( va_l > va_j \) holds in \( VP \). Consequently, \( va_i > va_j \) holds in \( VP \), since both \( va_i > va_l \) and \( va_l > va_j \) hold in \( VP \).

Lemma 2. Consider a process-view \( VP = \langle VA, VD \rangle \), where members of \( VA \) follow Definition 10 and members of \( VD \) follow Definitions 11 and 12, derived from a base process \( BP = \langle BA, BD \rangle \). Let \( LP \) be a loop-derived sub-process of \( BP \); let \( VLP \) be a loop-derived sub-process-view of \( VP \); \( VLP \) corresponds to \( LP \). For any two distinct virtual activities \( va_i \) and \( va_j \) in \( VLP \), \( va_i = \langle A_i, D_i \rangle \) and \( va_j = \langle A_j, D_j \rangle \): if there exist a base activity \( a_i \in A_i \) and a base activity \( a_j \in A_j \) such that \( a_i > a_j \) holds in \( LP \), then \( va_i > va_j \) holds in \( VLP \).
Proof. The proof is similar to the proof of Lemma 1 and is omitted. □

Lemma 3. Consider a process-view \( VP = \langle VA, VD \rangle \), where members of \( VA \) follow Definition 10 and members of \( VD \) follow Definitions 11 and 12, derived from a base process \( BP = \langle BA, BD \rangle \). For any two distinct virtual activities \( a_i \) and \( a_j \) in \( VP \), \( a_i = \langle A_i, D_i \rangle \) and \( a_j = \langle A_j, D_j \rangle \), where \( a_i > a_j \) holds in \( VP \). A path \( vp \) from \( a_i \) to \( a_j \) must exist, since \( a_i > a_j \) holds in \( VP \). The following proves this lemma by induction on the length of the path \( vp \) from \( a_i \) to \( a_j \).

When the length of the path \( vp \) is one, \( vdep(v_{a_i}, v_{a_j}) \) exists in \( VP \). By Definitions 11 and 12, \( vdep(a_i, a_j) \) exists in \( BP \), where base activities \( a_i \) and \( a_j \) are members of \( VA \) and \( VA \), respectively, (i.e., \( a_i \in A_i \) and \( a_j \in A_j \)). Thus, \( a_i > a_j \) holds in \( BP \).

The induction hypothesis assumes that for any two distinct virtual activities \( a_r \) and \( a_t \) in \( VP \), \( a_r = \langle A_r, D_r \rangle \) and \( a_t = \langle A_t, D_t \rangle \); if \( a_r > a_t \), holds in \( VP \) and there is a path from \( a_r \) to \( a_t \) with length \( \leq k \), then there exist a base activity \( a_s \in A_s \) and a base activity \( a_t \in A_t \) such that \( a_s > a_t \) holds in \( BP \).

The induction step must show that there exist a base activity \( a_s \in A_s \) and a base activity \( a_t \in A_t \) such that \( a_s > a_t \) holds in \( VP \), when the length of the path \( vp \) from \( a_i \) to \( a_j \) is \( \leq k + 1 \). Let \( a_k \) be a virtual activity on the path \( vp \) from \( a_i \) to \( a_j \), where \( a_k = \langle A_h, D_h \rangle \), \( a_k \neq a_i \) and \( a_j \). The length of the subpath of \( vp \) from \( a_i \) to \( a_k \) is \( \leq k \). Moreover, \( a_i > a_k \) holds in \( VP \) and \( a_k > a_j \) holds in \( VP \). Now, there exist a base activity \( a_s \in A_s \) and a base activity \( a_h \in A_h \), such that \( a_s > a_h \) holds in \( BP \), according to the induction hypothesis, since \( a_i > a_k \) holds in \( VP \) and there is a path from \( a_i \) to \( a_k \) with length \( \leq k \). Similarly, there exist a base activity \( a_t \in A_t \) and a base activity \( a_h \in A_h \) such that \( a_t > a_h \) holds in \( BP \). If \( a_h = a_r \), then \( a_t > a_h \) holds in \( BP \) since both \( a_i > a_h \) and \( a_r > a_j \) hold in \( BP \). On the other hand, if \( a_h \neq a_r \), then \( a_i > a_r \) holds in \( BP \), since \( a_i \) and \( a_h \) are members of \( VA \), \( a_t \) is not a member of \( VA \), and the ordering relations between \( a_t \) and all members of \( VA \) are identical in \( BP \), according to Definition 10(1c). Consequently, \( a_i > a_j \) holds in \( BP \) since both \( a_i > a_r \) and \( a_t > a_j \) hold in \( BP \). □

Lemma 4. Consider a process-view \( VP = \langle VA, VD \rangle \), where members of \( VA \) follow Definition 10 and members of \( VD \) follow Definitions 11 and 12, derived from a base process \( BP = \langle BA, BD \rangle \). Let \( LP \) be a loop-derived sub-process of \( BP \); let \( VLP \) be a loop-derived sub-process-view of \( VP \); \( VLP \) corresponds to \( LP \). For any two distinct virtual activities \( a_i \) and \( a_j \) in \( VLP \), \( a_i = \langle A_i, D_i \rangle \) and \( a_j = \langle A_j, D_j \rangle \); if \( a_i > a_j \) holds in \( VLP \), then there exist a base activity \( a_s \in A_s \) and a base activity \( a_j \in A_j \) such that \( a_s > a_j \) holds in \( LP \).

Proof. The proof is similar to the proof of Lemma 3 and is omitted. □

References


