Modeling Workflows with a Process-view Approach

Duen-Ren Liu Minxin Shen
Institute of Information Management
National Chiao Tung University, Taiwan
{dliu, shen}@iim.nctu.edu.tw

Abstract

Diverse requirements of participants involved in a business process bring forth the need of a flexible process model capable of providing appropriate process information for various participants. However, current activity-based approach is not adequate to provide different participants with varied process information. This work describes a novel process-view model for workflow management. A process-view is an abstracted process derived from a base process to provide abstracted process information. The underlying concept and formal model of a process-view are presented. Moreover, a novel ordering-preserved approach is proposed to derive a process-view from a base process. The proposed approach enhances the flexibility and functionality of conventional activity-based workflow management systems.

1. Introduction

As an effective process management tool, workflow management systems (WfMSs) allow a business to analyze, simulate, design, enact, control and monitor its overall business processes[5, 8, 11]. With the support of a WfMS, various participants can collaborate in effectively managing a workflow-controlled business process. In practice, these participants have different requirements and levels of authority to obtain information of business processes. To facilitate effective workflow management, a WfMS should provide various participants with adequate process information to fulfill their requirements.

Although using different notations, activity-based methodologies are extensively used process modeling techniques and adopted by many commercial products and research projects, e.g., MQ Series Workflow[7], Ultimus[13], METEOR[10], and WfMC (Workflow Management Coalition) process definition meta model[16]. Typical activity-based model is a procedure of top-down decomposition of a process. This stepwise refinement facilitates a modeler to define a process more easily and completely than one-step approaches.

However, subsequently layered process definitions do not always fit organizational hierarchy although they provide several different levels of hierarchical abstraction. Therefore, hierarchically decomposing a process may not provide each organizational level with an appropriate view of a process. Moreover, different departments may have difficulties in obtaining suitable abstractions of a process they participate in. The activity-based approach cannot adequately provide different participants with varied abstracted processes.

Enhancement of the activity-based approach has received considerable interest. Baresi et al.[2] and Gruhn[6] proposed models to integrate the modeling of activity, data, and organization. By exploiting property of SGML (Standard Generalized Markup Language) documents, Weitz[15] proposed a variant of Petri nets to combine the modeling of activity and data. Some investigations[3, 9, 14] use object-oriented technology to integrate the modeling of control flow and data flow.

van der Aalst [1] proposed a novel generic workflow model to provide the manager with an aggregated view of variants for the same workflow process. Multiple variants of the same process exist due to dynamic change. 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.

The activity-based approach should be enhanced to provide different process abstractions. Based on the notion of a view in a DBMS, this work presents a novel *virtual workflow process*, a *process-view* in a WfMS. A process-view, i.e., an *abstracted process* derived from an implemented *base process*, is used to provide abstracted information. With process views, a WfMS can provide various views of a process for different levels or departments in an organization.

Several approaches can be adopted to construct a process-view. This work describes a novel ordering-preserved approach in which a constructed process-view can preserve the original ordering of activities in a base

0-7695-0996-7/01 $10.00 © 2001 IEEE

process. A formal model is also presented to define an ordering-preserved process-view. Moreover, an algorithm is proposed to automatically generate an ordering-preserved process-view.

The rest of this paper is organized as follows: Section 2 formally defines business processes. Section 3 then describes a process-view and generally defines it. Next, Section 4 presents the proposed ordering-preserved approach to construct a process-view. Conclusions are finally made in Section 5.

2. Workflow Model: a Base Process

A process that may have multiple process-views is referred to herein as a *base process*. In general, activity-based workflow models use activities and dependencies to describe a process. Dependencies are used to describe the execution order and relationship between activities within a process. This work uses a rectangle to denote an activity and an arrow line to denote a dependency in a process graph. One dependency connects two activities. For an activity, its *succeeding* activities (*successor*) are connected by *outgoing* dependencies, and its *preceding* activities (*predecessor*) are connected by *incoming* dependencies. Each dependency is labeled with a condition. A workflow engine determines which succeeding activities will be triggered according to the condition evaluation and related ordering structure. WfMC defines six ordering structures, including *Sequence*, *AND-SPLIT*, *XOR-SPLIT*, *AND-JOIN*, *XOR-JOIN*, and *Loop* [16].

Herein, process-views are defined by introducing a formal model, revised from the standard WfMC model[16], to describe the base processes. The model focuses only on activities and dependencies to simplify the discussion.

Definition 1 (Dependency) A dependency is an ordered list (activity x, activity y, condition C), denoted by $dep(x, y, C)$. This notation indicates that after x is completed, a workflow engine can start to evaluate the condition C. The fact that x is completed and C is true is one precondition of whether y can start. This dependency is an outgoing dependency of x and an incoming dependency of y. Activity x is a predecessor of y and y is a successor of x. Activity x is called *preceding activity* and y is called *succeeding activity* in $dep(x, y, C)$.

Definition 2 (Activity) An activity is a 4-tuples $\langle AID, SPLIT\_flag, JOIN\_flag, SC\rangle$, where

1. *AID* is a unique activity identifier within a process.
2. *SPLIT_ flag* may be "NULL", "AND", or "XOR". NULL indicates this activity has only one outgoing dependency (Sequence). When multiple outgoing dependencies exist, AND indicates all succeeding branches can be followed (AND-SPLIT) while XOR indicates only one succeeding branch is followed (XOR-SPLIT).
3. *JOIN_ flag* may be "NULL", "AND", or "XOR". NULL indicates this activity has only one incoming dependency (Sequence). When multiple incoming dependencies exist, AND indicates this activity can start if all incoming dependencies have satisfied condition (AND-JOIN); XOR indicates this activity can start if one of the incoming dependencies has satisfied condition (XOR-JOIN).
4. *SC* is the *starting condition* of this activity. A workflow engine evaluates *SC* to determine whether this activity can start. If *JOIN_ flag* is NULL, *SC* equals the condition associated with its incoming dependency. If *JOIN_ flag* is XOR, *SC* equals Boolean XOR combination of all incoming dependencies' conditions. If *JOIN_flag* is AND, *SC* equals Boolean AND combination of all incoming dependencies' conditions.

SPLIT_ flag indicates how to choose outgoing dependencies to follow. *JOIN_ flag* determines how to combine incoming dependencies to trigger an activity. *SPLIT_flag*, *JOIN_flag*, and dependencies determine the control flow of a process.

Definition 3 (Process) A process P is a 2-tuples $\langle BA, BD\rangle$, where

1. BA is a nonempty set, and its members are activities within the process.
2. BD is a nonempty set, and its members are dependencies whose preceding activity and succeeding activity are contained by BA.

Definition 4 (Adjacent) Two activities are *adjacent* if connected by a dependency.

Definition 5 (Path) For a base process $P = \langle BA, BD\rangle$, $a_0, a_1, \ldots a_n \in BA$, $d_1, d_2, \ldots d_n \in BD$, where d_i represents the dependency from a_{i-1} to a_i, $i = 1, 2, \ldots n$. The list of activities and dependencies $a_0 d_1 a_1 d_2 \ldots d_n a_n$ is called the *path* from a_0 to a_n, denoted by $a_0 \rightarrow a_n$. The number of dependencies is called the *length* of a path, denoted by $length(a_0 \rightarrow a_n)$.

Definition 6 (Ordering Relation) For a base process $P = \langle BA, BD\rangle$, $\forall x, y \in BA$. Activity x is said to have a higher *order* than y if there is a path from x to y, i.e., x proceeds before y, and their ordering relation is denoted by $x > y$ or $y < x$. If $\exists x \rightarrow y$ and $y \rightarrow x$, i.e., $x > y$ and $y > x$, then every activity in the paths $x \rightarrow y$ and $y \rightarrow x$ belongs to the same loop structure. If $\nexists x \rightarrow y$ and $\nexists y \rightarrow x$ in P, i.e., x and y proceed independently, their ordering relation is denoted by $x \infty y$.

3. Virtual Process: a Process-view

In database management systems (DBMSs), a view is a virtual table generated from either physical tables or previously defined views. Based on the same notion, a process-view in WfMSs is defined herein. A process-view is generated from either physical processes (base processes) or other process-views and is considered a *virtual process*. It is used to provide abstracted information of its base process and does not modify its base process. During design time, a process modeler defines various process-views based on the participants' role. During run time, a WfMS initiates all process-view instances if their base process is initiated. Users with a specific role can obtain full information about the role's process-view instance. Process-views allow a process modeler to flexibly provide different roles (i.e., different levels or departments within an organization) with appropriate views of an implemented process.

Similar to process design, designing a process-view must identify what activities are within it and then arrange them based on dependencies and ordering structures. However, an "activity" in a process-view is not performed; it is used to express the execution states 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 dependency* for the process-view while the terms *base activity* and *base dependency* are used for the base process.

A process-view is defined as follows:

Definition 7 (Process-view) A process-view is a 2-tuples $\langle VA, VD \rangle$, where

1. VA is a nonempty set, and its members are virtual activities.
2. VD is a nonempty set, and its members are virtual dependencies.

A virtual activity is an abstraction of a set of base activities and corresponding base dependencies. A virtual dependency is used to connect two virtual activities 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 in which the original execution order in a base process is preserved. Regardless of how VA and VD are derived, paths and ordering relations can be defined in a process-view as follows.

Definition 8 (Virtual Path) For a process-view $VP = \langle VA, VD \rangle$, $va_0, va_1, \ldots, va_n \in VA$, $vd_1, vd_2, \ldots, vd_n \in VD$, where vd_i is the virtual dependency from va_{i-1} to va_i, $i = 1, 2, \ldots, n$. The list of virtual activities and virtual dependencies $va_0vd_1va_1vd_2 \ldots vd_nva_n$ is called the *virtual path* from va_0 to va_n, denoted by $va_0 \rightarrow va_n$.

Definition 9 (Ordering Relation between Virtual Activities) For a process-view $VP = \langle VA, VD \rangle$, $\forall va_i, va_j \in VA$, $i \neq j$, virtual activity va_i is said to have higher *order* than va_j, if there is a virtual path from va_i to va_j, i.e., va_i proceeds before va_j, and their ordering relation is denoted by $va_i > va_j$ or $va_j < va_i$. If there is no path from va_i to va_j or from va_j to va_i in VP, i.e., va_i and va_j both proceed independently, their ordering relation is denoted by $va_i \infty va_j$.

4. Ordering-preserved Approach

In this section, we first introduce three rules that a process-view must follow to preserve ordering property. Based on these rules, virtual activities and virtual dependencies in an ordering-preserved process-view are formally defined. *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. Also presented herein are novel algorithms that automatically generate legal virtual activities and virtual dependencies.

4.1 Basic Rules

If complying with the following three rules, a process-view preserves the ordering relations in its base process.

First, a virtual activity can be viewed as a set of activities of a base process. A virtual activity may be composed of base activities, virtual activities, or both.

Second, a virtual activity is an atomic unit of processing; it is completed if and only if each activity contained by it either has been completed or is never executed in a process instance, it starts if and only if one activity contained by it starts. In Figure 1, for example, if *SPLIT_flag* of a_1 is AND and *JOIN_flag* of a_4 is AND, a_4 cannot start until a_2 and a_3 are completed. Thus, the fact that virtual activity va_2 is completed implies that a_2 and a_3 are completed. If the *SPLIT_flag* of a_1 is XOR and *JOIN_flag* of a_4 is XOR, a_4 cannot start until a_2 or a_3 is completed. The fact that virtual activity va_2 finishes implies that one of a_2 and a_3 is completed and the other one is never executed.

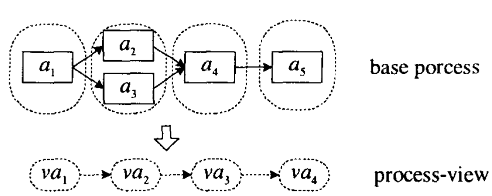

Figure 1. An illustrative example of the atomicity property

Furthermore, a situation in which an ordering relation, $\Re$ (>, < or ∞), between two virtual activities stands in a process-view implies that any ordering relation between these virtual activities' respective members is also $\Re$. The process-view in Figure 1 reveals that the ordering relation between va_1 and va_2 is $va_1 > va_2$. Because a virtual activity is an atomic unit, $va_1 > va_2$ implies that ">" is also the ordering relation between any member of va_1 and any member of va_2, i.e., $va_1 > va_2$ implies $a_1 > a_2$ and $a_1 > a_3$. Notably, the *implied ordering relations* may not conform to the ordering relations in the base process.

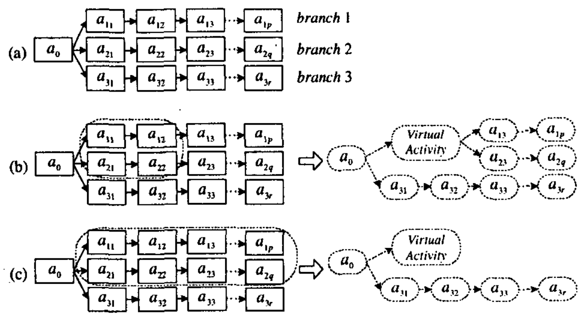

Figure 2. An illustrative example of ordering preservation in the split structure

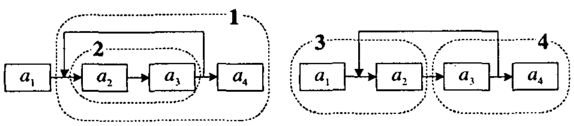

Figure 3. An illustrative example of ordering preservation in the loop structure

Third, defining a virtual activity requires that the original ordering relations between activities in a base process must be preserved. Consider a base process illustrated in Figure 2 (a), in which we want to define a virtual activity that must contain activities a_{11} and a_{22}. Figure 2 (b) and Figure 2 (c) provide two possible definitions. In the base process, three branches proceed independently and autonomously; in addition, the ordering relation between a_{13} and a_{22} is $a_{13} \infty a_{22}$. However, if we define a virtual activity as shown in Figure 2 (b), 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 *Virtual Activity* > a_{13} infers an implied ordering relation: $a_{22} > a_{13}$, i.e., a_{13} must wait for a_{22} completed to start. This implied ordering relation does not conform to the ordering relation in the base process. To preserve original ordering relations, the virtual activity must contain all activities in *branch* 1 and 2 as shown in Figure 2 (c).

For loop structure, the repetitive execution order must be kept when defining a virtual activity. According to Figure 3, each numbered dotted round rectangle is a possible definition. Although alternatives 1 and 2 are valid, 3 and 4 change the original ordering relations. Alternative 3 creates an implied ordering relation: $a_3 > a_1$, i.e., a_1, a_2, and a_3 may be repetitively executed (a_1 and a_2 are viewed as an atomic unit). Alternative 4 also creates an implied ordering relation: $a_4 > a_2$, i.e., a_4 may be executed without waiting for the repetitive execution condition of a_2 and a_3 is satisfied.

In sum, a *legal* virtual activity in an ordering-preserved process-view must follow three rules:

Rule 1 Membership: A virtual activity's member may be a base activity or a previously defined virtual activity. The membership among base activities and virtual activities is defined transitively. If x is a member of y and y is a member of z, then x is also a member of z.

Rule 2 Atomicity: A virtual activity, an atomic unit of processing, is completed if and only if each activity contained by it either has been completed or is never executed. A virtual activity starts if and only if one activity contained by it starts. In addition, if an ordering relation, $\Re$ (>, < or ∞), between two virtual activities stands in a process-view, then an implied ordering relation $\Re$ stands between these virtual activities' respective members.

Rule 3 Ordering preservation: The implied ordering relations between two virtual activities' respective members must conform to the ordering relations in the base process.

Moreover, based on Rule2 and Rule3, the following lemma can be derived:

Lemma 1. For an ordering-preserved process-view $VP = \langle VA, VD\rangle$ as derived from a base process $BP = \langle BA, BD\rangle$, $\forall va_i, va_j \in VA, i \neq j, \forall a_x, a_y \in BA, x \neq y$, a_x is a member of va_i and a_y is a member of va_j, if the ordering relation between va_i and va_j, $va_i \Re va_j$, stands in *VP*, then $a_x \Re a_y$ stands in *BP*.

Proof: $va_i \Re va_j$ implies $a_x \Re a_y$ (Rule 2). According to Rule3, the implied ordering relation $a_x \Re a_y$ must conform to the ordering relation in *BP*. Therefore, $va_i \Re va_j$ stands in $VP \Rightarrow a_x \Re a_y$ stands in *BP*. □

The approach is called ordering-preserved because implied ordering relations, as derived from a process-view, conform to the ordering relations in the base process.

4.2 Formal Model

The rules that a process-view should comply with have been introduced above. Next, virtual activities and virtual dependencies in an ordering-preserved process-view are formally defined.

Definition 10 (Virtual Activity) For a base process $BP = \langle BA, BD\rangle$, a virtual activity *va* is a 6-tuples $\langle VAID, A, D, SPLIT\_flag, JOIN\_flag, SC\rangle$, where

1. *VAID* is a unique virtual activity identifier within a process-view.
2. *A* is a nonempty set, and its members follow three rules:
   a. Its members may be base activities that are members of *BA* or other previously defined virtual activities that are derived from *BP*.
   b. The fact that *va* finishes implies that each member of *A* is either completed or never executed during run time; *va* starts implies that one member of *A* starts.
   c. $\forall x \in BA$, $x \notin A$, the ordering relations between *x* and all members (base activities) of *A* are identical in *BP*, i.e., $\forall y, z \in BA$, $y, z \in A$, if $x \Re y$ exists in *BP*, then $x \Re z$ also exists in *BP*.
3. *D* is a nonempty set, and its members are dependencies whose succeeding activity and preceding activity are contained by *A*.
4. *SPLIT_ flag* may be "NULL" or "MIX". NULL suggests that *va* has only one outgoing virtual dependency (Sequence) while MIX indicates that *va* has more than one outgoing virtual dependencies.
5. *JOIN_ flag* may be "NULL" or "MIX". NULL suggests that *va* has only one incoming virtual dependency (Sequence) while MIX indicates that *va* has more than one incoming virtual dependencies.
6. *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 associate with different ordering structure. Therefore, MIX is used to abstract the complicated ordering structures. A WfMS evaluates *SC* to determine whether *va* can start. Section 4.4 further discusses *JOIN_flag*, *SPLIT_flag*, and how to derive *SC*. Members of *A* are called *va*'s *member activities* and members of *D* are called *va*'s *member dependencies*. The abbreviated notation $va = \langle A, D\rangle$ is used to represent a virtual activity to save space in subsequent discussions.

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

Section 4.4 further discusses the relationship between *VC* and *C*. In the following discussion, condition fields of base dependencies and virtual dependencies are omitted for brevity. Next, we will demonstrate in Theorem 1 that the process-view defined according to Definition 10 and Definition 11 is ordering-preserved.

Theorem 1. For a process-view $VP = \langle VA, VD\rangle$, derived from a base process $BP = \langle BA, BD\rangle$, if members of *VA* follow Definition 10 and members of *VD* follow Definition 11, then *VP* is ordering-preserved.

Proof:

$\forall va_i, va_j \in VA$, $i \neq j$, $va_i = \langle A_i, D_i\rangle$, $va_j = \langle A_j, D_j\rangle$, $\forall a_x \in A_i$, $\forall a_y \in A_j$, if $va_i \Re va_j$, then the implied ordering relation $\Re$ stands between a_x and a_y (according to Rule2 Atomicity).

Case 1: $\Re$ is ">", i.e., $va_i > va_j$. It can be shown by induction that if $va_i > va_j$, then $\exists a_p \in A_i$, $a_q \in A_j$, such that $a_p > a_q$ stands in *BP*. Since $a_p > a_q$ and $a_q, a_y \in A_j$, by Definition 10.2.c, $a_p > a_y$. Since $a_p > a_y$ and $a_x, a_p \in A_i$, it further derives $a_x > a_y$ (by Definition 10.2.c). Therefore $a_x > a_y$ stands in *BP*. The proof for the case of "<" is similar and is omitted.

Case 2: $\Re$ is "∞", i.e., $va_i \infty va_j$. It can be shown by induction that if $\exists a_p \in A_i$, $a_q \in A_j$, $a_p > a_q$, then $va_i > va_j$. Thus, if $va_i \infty va_j$, then $a_p > a_q$ does not exist. Similarly, $a_p < a_q$ does not exist. Thus, $\forall a_x \in A_i$, $\forall a_y \in A_j$, $a_x \infty a_y$.

We have shown that if $va_i \Re va_j$ stands in *VP*, the implied ordering relation between a_x and a_y conforms to the ordering relation between a_x and a_y in *BP*. $\therefore$ *VP* is ordering-preserved.□

The virtual activity that follows Definition 10 maintains original ordering relations when abstracting base activities. However, this approach only ensures that syntax, i.e., execution order, is correct. Notably, the semantics of virtual activities is not of concern in this work. A process modeler must specify the implication of each virtual activity.

4.3 Essential Activity

From a process modeler's perspective, however, he/she merely wants to conceal sensitive activities or aggregate detailed activities. In addition to these activities, what activities must be included to form a legal virtual activity is not his/her primary concern and should be supported by a process-view definition tool. These sensitive or detailed activities are called essential activities.

Definition 12 (Essential Activity) Before defining a virtual activity, a modeler must select some activities that are essential to this virtual activity. These chosen activities are called *essential activities*, which 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 a "minimum virtual activity" can be found among these activities.

Definition 13 (Cover) For an essential activity set *EAS*, we say that $va_i = \langle A_i, D_i \rangle$ cover $va_j = \langle A_j, D_j \rangle$, if and only if $A_j \supseteq EAS$, $A_i \supseteq A_j$ and $D_i \supseteq D_j$.

Definition 14 (Minimum Virtual Activity) For an essential activity set *EAS*, a virtual activity $\langle A, D \rangle$ is called a *minimum virtual activity*, denoted by *min_va*(*EAS*), if it does not cover other virtual activities and $A \supseteq EAS$.

Given an *EAS*, a modeler must identify *min_va*(*EAS*). Besides essential activities, *A* only contains those activities needed to preserve original ordering relations in the base process, i.e., the *min_va*(*EAS*) only contains essential and adequate information to abstract *EAS*. Adding more activities, which are neither a modeler selected nor ordering preservation needed, into *A* merely adds unnecessary information into *min_va*(*EAS*).

The procedure of defining a process-view can be summarized as follows: A process modeler must initially select essential activities. The process-view definition tool then automatically generates a legal minimum virtual activity that covers (encapsulates) these essential activities. Above two steps are repeated until the modeler finishes all virtual activities that he/she needs. Next, the definition tool automatically generates all virtual dependencies between these virtual activities and ordering fields (JOIN, SPLIT) and starting condition of each virtual activity (control flow).

4.4 Algorithm

In this section, we introduce algorithms to derive an ordering-preserved process-view. Algorithm 1 derives the member activities and dependencies of a minimum virtual activity based on the modeler specified essential activities. Then we discuss how to derives virtual dependencies and the *JOIN_flag*, *SPLIT_flag*, and *SC* field of each virtual activity in the process-view.

Algorithm 1: Minimum Virtual Activity Generator

For a given essential activities set *EAS*, Figure 4 shows the algorithm capable of obtaining an *min_va*(*EAS*) = $\langle A, D \rangle$. Because *D* can be derived from *A* and *EAS* is known, members of *A* must be identified. As mentioned in Section 4.3, if $\forall x \in BA$, $x \in A$, and $x \notin EAS$, then *x* exists in *A* for ordering preservation. Obviously, *EAS* is a starting point to identify *x*.

Initially, the algorithm creates an activity set *TAS* that equals *EAS*. According to the definition of a virtual activity (Definition 10.2.c), $\forall x \in BA$, $x \in A$, the ordering relations between *x* and all members of *A* are identical in the base process *BP*. Therefore, $\forall x \in BA$, $x \notin EAS$, if the ordering relations between *x* and all members of *TAS* are not identical in *BP*, then *TAS* is not a legal (i.e., ordering-preserved) virtual activity.

To decide which of the activities should be added into *TAS* to form a virtual activity that is legal and minimum, the algorithm checks the ordering relations from the activities that are adjacent to a member of *TAS*. If the ordering relations between an adjacent activity and members of *TAS* are not identical, the adjacent activity should be added into *TAS* to follow Definition 10.2.c. If

procedure VAGenerator (Base process $BP = \langle BA, BD \rangle$, Essential activities set *EAS*)
begin
  Temp Activities Set *TAS* = *EAS*
  repeat
    Activity Set TAS_1 = *TAS*
    Adjacent activity set $AAS = \{ x \mid \forall x, y \in BA, x \notin TAS, y \in TAS \text{ and } \exists dep(x, y, C) \}$
    while *AAS* is not empty ***do***
      Select an activity *x* from *AAS*
      Remove *x* from *AAS*
      if $\exists y, z \in BA$, $y, z \in TAS$, such that $x \Re y$ exists in *BP* and $x \Re z$ does not exist in *BP*
      /* The ordering relations between *x* and all base activities of *TAS* are not identical in *BP* */
      then Add *x* to *TAS*
      end if
    end while
  until $TAS = TAS_1$
  An Activity Set *A* = *TAS*
  A Dependency Set $D = \{ dep(x, y, C) \mid x, y \in A \}$
  min_va(*EAS*) = $\langle A, D \rangle$
end

Figure 4. The algorithm of a minimum virtual activity generator

TAS has changed, the *repeat-until* loop repeats again to check the ordering relations. The *repeat-until* loop (Line 4~15) repeatedly executes until the activity set *TAS* satisfies the ordering-preserved condition (Definition 10.2.c). Finally, $\forall x, y \in BA, x \notin TAS, y \in TAS$, x and y are adjacent, if the ordering relations between x and all members of *TAS* are identical in *BP*, then the *repeat-until* loop stops.

After A has been determined, members of D are those dependencies whose succeeding activity and preceding activity both are members of A (Definition 10.3). The minimum virtual activity of EAS, *min_va*(*EAS*), equals $\langle A,D \rangle$.

Owing to that this virtual activity conforms to Definition 10, it is a legal virtual activity. Moreover, the algorithm checks ordering relations from adjacent activities, resulting in a minimum virtual activity. The proof is illustrated in [12].

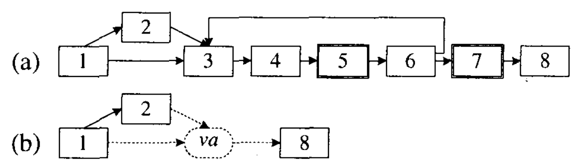

Figure 5. A process example

Example 1. This example illustrates how to derive the *min_va*(*EAS*) if part of a base process is shown in Figure 5 (a) and *EAS* = {5, 7}. At the beginning, *AAS* = {4, 6, 8} and *TAS* = {5, 7}. Activity 6 is added into *TAS* since 5 > 6 but 7 < 6; activity 8 is not added to *TAS* since 7 > 8 and 6 > 8. Notably, 4 is added into *TAS* since 4 > 7, 4 > 5, and 4 < 5 ($\because 5 \rightarrow 4$). Therefore, *TAS* changes to {4, 5, 6, 7} and *repeat-until* loop repeats again. In the second execution, *AAS* = {3, 8}. Activity 8 is not added into *TAS* since 4 > 8, 5 > 8, 6 > 8, and 7 > 8. Activity 3 is added into *TAS* since 3 < 6 ($\because 6 \rightarrow 3$) but 3 > 7. Therefore, *TAS* is updated to be {3, 4, 5, 6, 7} and the *repeat-until* loop repeats again. In the third execution, *AAS* = {1, 2, 8}. *TAS* does not change in the *while* loop since the ordering relations between each adjacent activity and members of *AAS* are identical in *BP*. Therefore, the *repeat-until* loop stops and *A* = {3, 4, 5, 6, 7}, *D* = { *dep*(3, 4), *dep*(4, 5), *dep*(5, 6), *dep*(6, 7), *dep*(6, 3) }. The result is shown in Figure 5 (b).

Virtual Dependency

After all virtual activities have been generated, the process-view definition tool derives virtual dependencies by Definition 11. First, members of a virtual dependency must be specified. *VC* field of each virtual dependency must then be specified.

For a process-view *VP*, its virtual activity set *VA* is known. First, whether a virtual dependency is associated with two virtual activities must be determined. $\forall va_i, va_j \in VA, i \neq j, va_i = \langle A_i, D_i \rangle, va_j = \langle A_j, D_j \rangle$, if $\exists dep(a_x, a_y, C_{xy})$, $a_x \in A_i, a_y \in A_j$, then $\exists vdep(va_i, va_j, VC_{ij})$ and the $dep(a_x, a_y, C_{xy})$ is a member of $vdep(va_i, va_j, VC_{ij})$. After checking each base dependency, all virtual dependencies and their members can be derived.

For a base activity, its *JOIN_flag* determines how to combine the conditions of incoming dependencies. Therefore, for the members of a virtual dependency, conditions of the base dependencies that have the same succeeding base activity are combined by the succeeding base activity's *JOIN_flag*. According to atomicity rule, a virtual activity starts if one member activity starts. Therefore, these conditions that derived from different succeeding base activities are then combined by Boolean OR. Given two virtual activities $va_i = \langle A_i, D_i \rangle$ and $va_j = \langle A_j, D_j \rangle$, where $A_j = \{a_{y1}, a_{y2}, \ldots, a_{yn}\}$. Let C_{yk} be the joined condition of all dependencies from A_i to a_{yk}, k=1..n, $C_{yk} = f$ ($all\ C_{x,yk}$), where f = *JOIN_flag* of a_y, $\forall a_x \in A_i$ and $\exists dep(a_x, a_{yk}, C_{x,yk})$. For the virtual dependency $vdep(va_i, va_j, VC_{ij})$, $VC_{ij} = (C_{y1}$ OR C_{y2} ...OR $C_{yn})$.

The fact that *VC* evaluates to true is one precondition of the execution of a succeeding virtual activity. Whether a succeeding virtual activity can start depends on its starting condition (*SC*).

Due to the space limit of this paper, illustrative examples are not presented and can be found in [12]

Ordering Structure and Starting Condition

After all virtual dependencies have been generated, ordering fields (JOIN, SPLIT) and starting condition of each virtual activity can be derived. If a virtual activity has only one outgoing virtual dependency, its *SPLIT_flag* is NULL. Otherwise, having two or more outgoing virtual dependencies, *SPLIT_flag* of the virtual activity is MIX. We cannot simply determine that the *SPLIT_flag* of the virtual activity is AND or XOR since a virtual activity abstracts a set of base activities that may associate with AND-SPLIT and XOR-SPLIT concurrently.

Similarly, if a virtual activity has only one incoming virtual dependency, its *JOIN_flag* is NULL. Otherwise, having two or more incoming virtual dependencies, *JOIN_flag* of the virtual activity is MIX. For a base activity, *JOIN_flag* determines the relationship between its starting condition (*SC*) and the conditions (*C*) of its incoming base dependencies. For a virtual activity, MIX-JOIN abstracts the existence of different join structure in its member base activities. Therefore, the starting condition (*SC*) of a virtual activity cannot simply use *JOIN_flag* to combine incoming virtual dependencies' *VC*. MIX-SPLIT/JOIN is used to represent multiple paths structure, whether a virtual activity can start depends on the *SC* field that is derived as following.

For a virtual activity *va*, it starts if one of its member activities starts (Atomicity Rule). Therefore, the starting condition of *va*'s each member activity must be determined, and then the starting condition (*SC*) of *va* equals the Boolean OR combination of each member activity's starting condition. If $va = \langle A, D \rangle$, $A = \{a_1, a_2, a_3, \ldots, a_n\}$, $\forall a_x \in A$, the starting condition of a_x is $SC(a_x)$, then the starting condition of *va*, $SC(va) = (SC(a_1)$ OR $SC(a_2)$ ... OR $SC(a_n))$. That means *SC* is true if one member activity's starting condition is satisfied.

As mentioned above, *SC* of a virtual activity is determined by atomicity rule. In this manner, a process-view can express progress information of a base process. Moreover, evaluating *VC* of each virtual dependency can indicate ordering behavior in a process-view.

5. Conclusion

This work proposes a novel concept of process abstraction: process-view. Process-view enhances the conventional activity-based model to satisfy diverse requirements of abstraction. A process modeler can easily use a process-view definition tool to provide numerous views of a business process for different levels and divisions. Process-view achieves information abstraction and progress monitoring. Each role can obtain adequate information on a business process via the role-related process-view, thereby facilitating the coordination within an enterprise. Moreover, in light of the importance of execution order in a business process, this work also proposes an ordering-preserved approach to construct a process-view that ensures the original ordering of activities in the base process is preserved. The algorithm, which automatically derives a minimum virtual activity, assists vendors in implementing process-view definition tools in their commercial systems. The proposed approach increases the flexibility and functionality of current WfMSs.

Acknowledgments

The work was supported in part by National Science Council of the Republic of China under the grant NSC 88-2416-H-009-023.

References

[1] W.M.P. van der Aalst, "Generic workflow models: how to handle dynamic change and capture management information?", *Proceedings of 4th IFCIS International Conference on Cooperative Information Systems (CoopIS'99)*, IEEE Computer Society, Edinburgh, Scotland, pp. 115-126, Sept. 1999.

[2] L. Baresi, F. Casati, S. Castano, M.G. Fugini, I. Mirbel, and B. Pernici, "WIDE workflow development methodology", *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC'99)*, ACM Press, San Francisco, CA USA, pp. 19-28, Feb.1999.

[3] K.A. Butler, A. Bahrami, C. Esposito, and R. Hebron, "Conceptual models for coordinating the design of user work with the design of information systems", *Data & Knowledge Engineering*, vol.33, no. 2, pp.191-198, 2000.

[4] S. Castano, V. De Antonellis, and M. Melchiori, "A methodology and tool environment for process analysis and reengineering", *Data & Knowledge Engineering*, vol. 31, no 3, pp. 253-278, 1999.

[5] D. Georgakopoulos, M. Hornick, and A. Sheth, "An overview of workflow management - from process modeling to workflow automation infrastructure", *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119-153, 1995.

[6] V. Gruhn, "Business process modeling and workflow management", *International Journal of Cooperative Information Systems*, vol. 4, no. 2&3, pp. 145-164, 1995.

[7] IBM, "MQ Series Workflow", http://www-4.ibm.com/software/ts/mqseries/workflow

[8] S. Jajodia, "Interview: Amit Sheth on workflow technology", *IEEE Concurrency*, vol. 6, no. 2, pp. 21-23, 1998.

[9] G. Kappel, P. Lang, and S. Rausch-Schott, "Workflow management based on objects, rules, and roles", *IEEE Bulletin of the Technical Committee on Data Engineering*, vol. 18, no. 1, pp. 11-18, 1995.

[10] N. Krishnakumar and A. Sheth, "Managing heterogeneous multi-system tasks to support enterprise-wide operations", *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 155-186, 1995.

[11] F. Leymann and W. Altenhuber, "Managing business processes as an information resource", *IBM System Journal*, vol. 33, no. 2, pp. 326-348, 1994.

[12] D.-R. Liu, M. Shen, "Workflow modeling: a process-view approach", Technical report, Institute of Information Management, National Chiao-Tung University, 2000.

[13] Peace Systems Integrations, "Ultimus Workflow Suite", http://www.ultimus1.com

[14] G. Vossen and M. Weske, "The WASA2 object-oriented workflow management system", *Proceedings of the international conference on Management of data (SIGMOD)*, ACM Press, Philadelphia, PA USA, pp. 587–589, May 1999.

[15] W. Weitz, "Combining structured documents with high-level petri-nets for workflow modeling in internet-based commerce", *International Journal of Cooperative Information Systems*, vol. 7, no. 4, pp. 275-296, 1998.

[16] Workflow Management Coalition, "Interface 1: process definition interchange process model", Technical report, WfMC TC-1016-P, Version 1.1, 1999.