This is an earlier version of the paper published in
3rd IFCIS International Conference on Cooperative Information Systems (CoopIS'98), New York, Aug. 1998, pp. 310-319.

Managing Evolving Workflow Specifications*

Gregor Joeris and Otthein Herzog
Intelligent Systems Department, TZI - Center for Computing Technologies
University of Bremen, PO Box 330 440, D-28334 Bremen
joeris|herzog@informatik.uni-bremen.de

Abstract

Dynamic evolution of workflow models due to process (re)engineering activities and dynamic changing situations of the real process is one of the most important challenges in workflow management. In this paper, we present an approach for the management of evolving workflow specifications which copes with the evolution of a workflow schema and the dynamic modification of workflow instances. The approach is based on the integrated modeling of workflow schema and instance elements, the separated definition of 'what to do' and 'how to do' in the workflow schema, late binding of workflows at run-time, and the versioning of the workflow schema. On this basis, we support lazy, eager, and selective propagation as well as local customization of instances and their upward propagation. Furthermore, we address the problem of managing consistent configurations of the versioned entities of a workflow schema. In our workflow-specific versioning approach, the consistency of the workflow configuration is guaranteed and hence the version mechanism is transparent to the user.

1. Introduction

Due to the heterogeneity of the involved process steps and today's demand on continuous process improvement, flexibility and adaptability is one of the most challenging requirements for a workflow management system (WFMS) and its underlying workflow modeling language [21]. Flexibility of a WFMS encompasses two fundamental aspects:

(1) The specification of a *flexible execution behavior* to express in advance an accurate and less restrictive behavior, e.g., to support cooperative activities within a process-centered environment (cf. [8, 16, 13]).

(2) The *evolution of workflow models* in order to flexibly modify workflow specifications on schema and instance level due to process (re)engineering activities and dynamic changing situations of the real process (cf. [6, 2, 9, 5]).

In this paper, we focus on the evolution of workflow models. There are several reasons why workflow specifications evolve over time (cf. [20]). One of the most important ones is the demand for continuous process *improvement* which leads to a continuous adaptation of the corresponding workflow specifications. Another reason is caused by the predictability of the modeled process: in order to support semi-structured processes a workflow model may be refined or *adjusted ad hoc* to the real world situation. Further reasons are *corrections* of an erroneous workflow model and the *customization* of a workflow to the needs of a specific case.

Workflow evolution causes two main problems: first, the management of different workflow schema versions and the support of different change propagation strategies to running workflow cases; second, ensuring the dynamical correctness of workflow changes w.r.t. running instances. The aim of this paper is to present an approach which supports the *management* of evolving workflow specifications. It is not our goal to present a minimal and correct set of change operations since several changes are needed to obtain a new semantically correct workflow specification. Therefore, we are quite sure that we need to pay more attention to the management aspect of workflow evolution. In particular, mechanisms which support performing complex workflow changes such as partial suspension of running instances, analyzing the impact of workflow changes w.r.t. to the current execution states, and support for different propagation policies are needed.

Our approach for managing evolving workflow specifications is based on the separated modeling of 'what to do' and 'how to do', the versioning of the workflow schema, schema releases, late binding, and facilities for suspending and resuming of running workflow instances. We firmly believe that particularly version management of a workflow schema, which can be treated as complex design objects, is the key to support workflow evolution. Finally, our workflow meta model tightly integrates schema and instance elements to allow for analyzing the impact of schema changes on their instances.

* This work was partially supported by the German Ministry for Research and Technology (BMBF) , project MOKASSIN under grant number 01 IS 601 D.

In section 2, we outline the main requirements of workflow evolution management. Section 3 gives a brief overview of our workflow modeling language. Section 4 introduces our approach to workflow schema versioning and its application for the management of evolving workflow specifications. Section 5 discusses related work, and section 6 gives a short conclusion.

2. Requirements of Workflow Evolution

From a technical point of view, we have identified the following main requirements for a WFMS to support the evolution of workflow specifications:

A. Support for the management of evolving workflow schemata: Different workflow schema versions have to be managed and different propagation strategies of workflow schema changes to their workflow instances (cf. [6, 5, 4]) have to be provided by a WFMS in order to flexibly support the migration from one business process to an improved one, to support alternative workflows for process variants, and to support ad hoc changes of a workflow. We distinguish

- *lazy propagation*, i.e., a workflow schema is changed without any impact on currently enacting instances. The new workflow schema version becomes only relevant for all new workflow instances which are created on the basis of the modified schema. This policy is useful for the introduction of a new (improved) 'short-living' business process.
- *eager propagation*, i.e., workflow schema changes are propagated immediately to all workflow instances of the changed workflow definition. Thus, this strategy leads to on-the-fly changes (see below). This policy is useful for error corrections or process improvements of long-lived processes.
- *selective propagation*, i.e., workflow schema changes are propagated immediately to a selected set of workflow instances of the changed workflow definition. The *selection* of the workflow instances may be done manually by the process designer or automatically by the WFMS according to the instance execution states. The propagation has possibly to be delayed until the instance has reached the required state (cf. [4]). Furthermore, local adjustments to some workflow instances may additionally take place.
- *local modifications and upward propagation:* A special case of selective propagation is the change of the workflow definition for exactly one workflow instance in order to locally customize the workflow structure for a specific case (before the execution starts) or to locally adjust it (e.g., due to exceptional situations). This strategy is also useful in the case of processes which cannot be planned completely in advance (e.g., software processes). In this case, planning and enactment of processes have to be interleaved. When locally modified workflows turn out to be of general interest, the changes have to be propagated to the general workflow definition (which we denote as *upward propagation*) and/or to other workflow instances.
- *merging*: When changes have to be applied to different workflow variants – this is the case when a workflow schema of a locally modified workflow instance is changed – we need mechanisms which support merging of different workflow specifications.

B. Consistency of workflow instance changes: Change propagation to or local modification of running workflow instances lead to the problem of controlling and handling the impact of the changes to running instances in order to avoid inconsistent execution states. In general, unrestricted changes should be disallowed and regulations/compensations may be needed to re-establish the consistency of the execution state.

The consistency of a workflow instance depends on the dynamical correctness of the workflow schema and the current execution state. The dynamical correctness of a workflow schema ensures that every execution state within reach from the *initial* state is safe. Usually, analysis and simulation tools are used by the workflow modeler to check this property. Correspondingly, a workflow instance is in a consistent execution state if its *current* execution state is safe with respect to the new workflow schema. Among this correctness criteria the property of compliance defined by [5] is also important. It requires that the whole execution trace of a workflow instance is legal w.r.t. the new workflow schema. When this property holds, a workflow instance can be migrated to a new workflow schema without any regulations. Otherwise either some activities are rolled back to make the instance compliant, or a variant of the workflow has to be introduced to which the instance is compliant and which avoids rollbacking of activities.

Examples: The extension of the workflow schema by a new workflow definition will not affect any workflow instances, whereas the deletion of a control flow dependency of a workflow may result in the re-evaluation of the execution states. The deletion of a workflow definition which is populated by instances should be prohibited anyway. Finally, the removal of a process step may require compensation activities for those instances which already have performed this step.

Let us conclude this section with two final remarks: First, and this is an essential point when we want to increase the flexibility of a WFMS by supporting dynamic

workflow modifications, we are convinced that we need a workflow modeling language on a high level of abstraction in order to allow for a participatory design and change of workflows. We have designed our workflow modeling language with this requirement in mind.

Second, we need flexible mechanisms which restrict possible workflow changes so that the workflows are (semantically) valid according to general business rules, and which restrict the actors who are authorized to perform different types of changes based on the integration of an organization model (cf. [4, 22]).

3. The Workflow Modeling Approach

Our process modeling language is based on object-oriented modeling techniques, i.e., all relevant entities are modeled as attributed, encapsulated, and interacting objects (which does not imply an OO process modeling method). In particular, workflow schema and workflow instance elements are modeled as first level objects and their relationships are explicitly maintained. Following the principle of separation of concerns, we divide the overall model into sub-models for tasks and workflow, documents and their versions, resources, and organizational units (see [14] for a detailed overview). In the following, we focus only on the modeling of tasks and their flow structure (as illustrated in Fig. 2) and concentrate in the next section on the integration of the versioning concept rather than on details of the modeling constructs.

3.1 Task and Workflow Definition

Functional and structural aspects: First of all, a *task definition* (or task type) is separated into the definition of the *task interface* which specifies 'what is to do', and potentially several *workflow definitions* which specify how the task may be accomplished (how to do) (see Fig. 1). The task interface is defined by a set of attribute definitions which may hold so-called workflow relevant data (omitted in Fig. 2), a set of parameter definitions which define the type and kind of inputs and outputs of a task, a behavior definition which defines the external behavior of a task (e.g., transactional or non-transactional; omitted in Fig. 2), and a set of business rules which constraints the valid workflow definitions (omitted in Fig. 2).

```
taskdefinition <task_def_name> [ is_a <task_def_name>]
  [attribute_definition_list]                  // ⎡
  [parameter_definition_list]                  // | task
  [behavior_definition]                        // | interface
  [business_rule_definition_list]              // ⎣
  [workflow_definition_list]                   // [ task body
```

Figure 2: Structure of a task definition

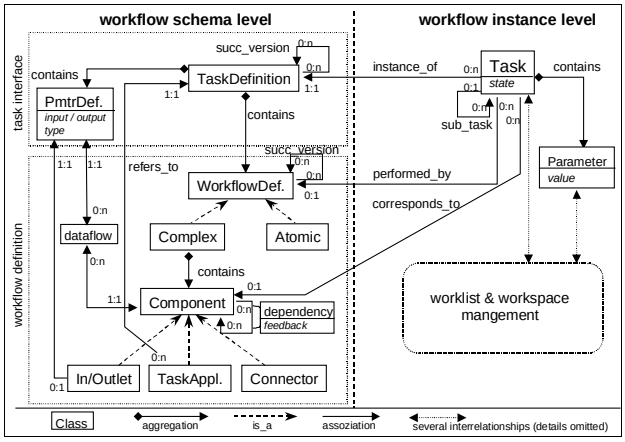

Figure 1: Workflow part of the meta model (simplified)

A workflow definition[1] may be atomic consisting only of a process or program description, or complex (see Figure 2). A complex workflow is defined in a process-oriented manner by a *task graph* which consists of task components, connector types (and/or/xor-splits/joins), and data inlets and outlets. A task component is an applied occurrence of a task definition representing the invocation hierarchy. The semantics of the connector types are total, conditional, and exclusive conditional branching for and-, or- and xor-splits, respectively. The corresponding join-connectors synchronize the activated branches. A data inlet (or outlet) is a data source (or sink) in order to realize a vertical dataflow between the parameter of the task definition and their use within the workflow. Task components and connectors are linked by control flow dependencies. Iterations within this task graph are modeled by a special feedback relationship. Furthermore, task components can be linked by dataflow dependencies according to the input and output parameters of their task definitions. Finally, task definitions can be locally declared to another task definition restricting their visibility to this task type. Thus, the declaration and invocation hierarchy of task types are separated (as known from programming languages). Both, the separation of 'what' and 'how' to do as well as the separation of the declaration and invocation hierarchy, which are violated by several workflow modeling approaches, are fundamental when introducing our concept of workflow schema versioning and integrating it with the workflow modeling concepts.

[1] Note, that we use workflow definition in a more restricted sense defining only how a task has to be done. To avoid misunderstandings, from now on we use workflow specification as a general term (independent from our approach) in the usual more broader sense. Further, we use workflow schema to denote the declaration of all workflow specifications: w.r.t. our approach this means the declaration of all task definitions which may contain several workflow definitions.

The decision which workflow definition is used to perform a task is done at run-time (late binding). Every workflow definition has a condition, which acts as a guard and restricts the allowed workflows according to the current case. In a non-deterministic case, the choice is left open to the user.

Behavioral aspects: The dynamic behavior of tasks and of the task graph is defined by a statechart variant and event passing among related tasks of the task graph. Every task type inherits from a predefined statechart which defines the fundamental states and transitions of a task (illustrated in Fig. 3, omitting the conditions and events which are associated with every operation). A task can be treated as a *reactive component* which encapsulates its internal behavior and interacts with other components by message/event passing. Thus, we understand the dependencies between tasks as communication channels between objects. The connector types are treated as special tasks with specific pre-conditions and event handling rules. Thus, the semantics of the control flow dependencies are defined by the signals passed along them. This leads to a combined approach which integrates the flexibility of rule-based techniques with the high-level constructs of task graphs. Furthermore, the inter-object communication is a natural basis for a distributed enactment of workflows.

Example: When a task is finished the event 'predecessor_done' is sent to the successor tasks which will evaluate their activation condition and may perform the enable transition (see also Fig. 6 d) and e)). When the task can be executed automatically (this is particularly the case for connectors), the enable event which was raised by the enable transition will trigger the start transition. Another example is the suspension of a task which is signaled to all subtasks that will trigger the disable, suspend, or abort transition of the subtasks depending on their behavior definition and their current execution state.

3.2 Task Instances

For instantiation, neither a copy of a task definition is created (e.g., as in the petri-net based SPADE approach [1]) and enriched by execution-relevant information (e.g., assignment of start tokens) nor separated representation formalisms for schema and instance level nets are used (e.g., as in EPOS [12] or DYNAMITE [10]). Rather, only the actual task instances with their execution state, their dynamic invocation hierarchy, and their actual dataflow are covered at instance level. Because of our focus on collaborative and human-centered workflows, we have integrated workspace management facilities to support different types of data interchange (which have been omitted in Fig. 2 and which are not addressed here; see [13] for details).

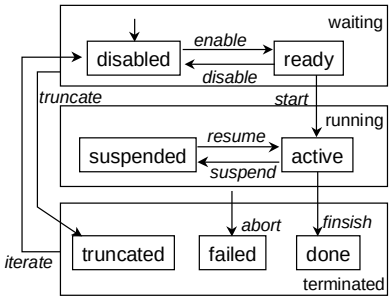

Figure 3: Predefined statechart

A task instance is associated with exactly one task definition (version), and this instance relationship is explicitly maintained in the process database (as illustrated in Fig. 2). Further, when a workflow was chosen for execution at run-time a 'performed_by' relationship is inserted, the subtasks are created according to the chosen workflow definition, and for every subtask the corresponding component within the workflow definition is identified. Thus, the dynamic task hierarchy is created step-by-step.

Thus, workflow schema and instance elements are tightly integrated. All execution-relevant structural information of a workflow schema can be accessed by the instances, and, vice versa, changes of the workflow schema can be analyzed according to their behavioral consistency w.r.t. to the corresponding instances. Furthermore, workflow schema changes immediately affect all instances since the workflow engine will schedule the task according to the changed schema. To support lazy and selective propagation as well as local modifications of a workflow instance, we use the schema versioning concept which is introduced in the next section, where we also explain how the execution state is kept consistent when dynamic changes are performed.

The tight coupling of schema and instances is also reflected by our architecture. We follow an integrated approach which does not separate between build- and run-time environments and which is the basis to support dynamic workflow changes. Due to space limitations we must omit further details of our system architecture, which is designed as a distributed object system.

4. Workflow Evolution Management Using Schema Versioning

This section describes our approach to the management of workflow evolution where we emphasize the management of different workflow specification versions rather than on the correctness of a certain change operation. We introduce our workflow schema versioning concept and its integration with the separated modeling of task and workflow definitions and describe how versioning can be used to support different propagation policies.

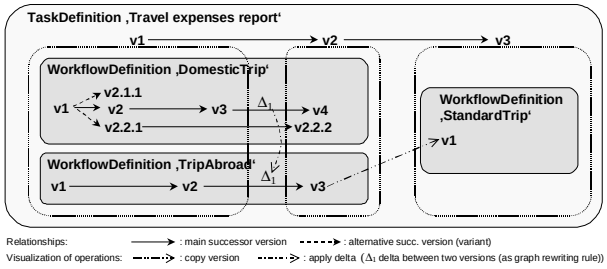

Figure 4: Task and workflow definition versions

4.1 Comparison to Schema Evolution and Schema Versioning in OODB

Although the general problem of the evolution of a schema in a OODBMS ([3, 23]) and of a workflow schema is similar to some extent, some differences are important: First, workflow schemata are more specific and are treated as first level objects: Several workflow specifications exist within a workflow dictionary in order to provide workflow support for different processes. The design, analysis, and management of these workflow specifications is a main and frequent task within WFM. Thus, according to this functionality workflow schemata are first level objects.

Second, the problem of schema evolution in OODBMS is based on the long-liveness of the instance data and the need for a new structuring or a new structural view on this data. The main problem is the conversion of the database to the new schema. On the other hand, workflow schema evolution leads to the problem of managing different change propagation strategies and the handling of on-the-fly changes as described in Section 2. Furthermore, the lazy propagation policy, which is useful in many cases, shows that workflow instances are often short-living rather than long-living entities.

Due to these differences, approaches of schema versioning in OODBMS (cf. [15, 19, 17]) differ from our approach to workflow schema versioning. In OODBMS, several versions of a schema can be seen as different (structural) views on the instance data which may be also versioned in order to fit into the different schema versions (as proposed by [17]). Different applications may use different schema versions when accessing potentially the same data. In a WFMS, a workflow instance is associated with exactly one version of a workflow specification.

4.2 The Workflow Schema Versioning Approach

A workflow schema may be treated as a complex design object which can be versioned as an ordinary object. Therefore, workflow schema versioning is very similar to object versioning ([11, 7]). When applying versioning to a workflow schema we have to define how versions of an (schema) object are represented and on what level of granularity versioning is applied.

Further, we have to cope with the well-known problem of managing consistent configurations of complex versioned objects which applies to the decomposition of workflows. To the best of our knowledge, this fundamental problem of the management of evolving workflow specifications has not been addressed so far.

Granularity of versioning: Versioning is usually applied to a coarse-grained entity (e.g., document or class) which consists of a fine-grained internal structure (e.g., document content or attributes/methods). According to schema versioning in OODBs, we distinguish between the versioning of the whole workflow schema (schema level) and the versioning of workflow specifications (class level).

First, the versioning of the whole workflow schema is not a reasonable solution for managing workflow evolution. Except that this solution avoids the problem of managing configurations, it has several limitations. Even in the case of minor changes, e.g., of the flow structure of a workflow, all workflow specifications have to be derived although they are not affected.

Second, versioning on the level of workflow specifications (in our approach task definitions) is more appropriate for managing evolving workflows. Since workflow specifications are part of or may be applied within other workflow specifications, they are not independent from each other. Therefore, the problem of ensuring consistent configurations of workflow versions arises. The derivation of a new version of a workflow specification WF will lead - in combination with supporting different propagation policies - to the (transitive) derivation of new versions of all workflow specifications where WF occurs. Thus, in order to obtain a consistent workflow specification, always top-level workflow specifications (with their tree of sub-workflows) have to be derived.

Due to these problems we introduce a third and more fine-grained level of versioning which uses the separation of a task definition into the interface definition and the workflow definition. Besides the versioning of complete task definitions we also support versioning of a workflow definition as part of a task definition version. This solution has several advantages for the management for evolving workflows as explained in the next sub-section. Note, that our schema versioning approach depends on the separated modeling of task and workflow definitions, but is in particular independent from the modeling elements which define how the flow structure is specified. This implies that our task graphs could be replaced by, e.g., a petri-net based approach.

Example: In Fig. 4 three versions with a linear history of the task definition 'travel expense report' are shown. Initially, this task definition consists of two alternative workflow definitions for domestic trips and trips abroad, respectively. Process improvement has led to new versions of both workflow definitions whereas for the migration of the version v1 of the DomesticTrip workflow two variants were needed to correctly migrate running instances. In version v3 of travel expense report the different workflows for domestic trips and trips abroad was given up and only one workflow definition was designed based on the current 'TripAbroad' workflow definition versions.

Representation of versions: Since we use versioning for the recording of well-defined states of evolving workflow specifications and for managing their corresponding workflow instances, we use an extensional (state-based) versioning approach for the representation of task and workflow definition versions. Every task/workflow definition version is explicitly recorded. A derivation relationship between versions covers the version history and expresses a successor relationship between versions. This derivation relationship forms a DAG. Thus, alternative versions (variants) are represented as branches which may be joined later on. Further, the DAG is structured so that within every (sub)branch one main branch is identified. Formally, a version DAG is defined by the triple (V, f, R) where V is a set of Versions, $f : V \rightarrow V$ a partially defined function which defines the (main) successor revision representing the main branch, and $R \subseteq V \times V$ a relation which defines the variants derived from a version (see Fig. 4).

The decision when to derive a new version depends on both the propagation strategy which has to be applied and the decision of the workflow designer about what kind of changes lead to a new version. Derivation of a new version is orthogonal to changing it.

Version operations and configuration management: So far, we have not addressed the problem of managing consistent configurations of the versioned entities of the workflow schema. Since our versioning approach aims at supporting workflow evolution management, the WFMS should automatically guarantee the consistency of the task and workflow definition configurations and hence configuration should be transparent to the user. Furthermore, as explained above, transitive derivations of versions w.r.t. to their composition relationship should be avoided. Our solution is based on providing specific derivation operations for task and workflow definition versions.

A) Derivation of task definition versions: When a new task definition version is derived (i.e., a copy is taken and a derivation relationship is inserted) the workflow definition versions of the old task definition version are not part of the new task definition version. Rather, the most current workflow definition version (i.e., the version of the main branch without any successor version) is also derived and assigned to the new task definition version (step (1) and (2) in Fig. 5). In any case, a workflow definition version is part of exactly one task definition version and hence has a unique task interface.

Next, we have to handle task components which are part of other workflow definition versions and refer to the old task definition version (cf. Fig. 5). Every workflow definition version WFD_{old} which has no successor versions and which contains a task component that refers to the old task definition version TD_{old} is also derived (3) and in the resulting workflow definition version WFD_{new} the reference of the task component is updated to the new task definition version TD_{new} (4).

B) Derivation of workflow definition versions: A workflow definition version is always derived in the context of a task definition version TD and the new workflow definition version WFD_{new} becomes part of TD. The source workflow definition versions WFD_{old} from which the new one is derived must be either also part of TD or part of a direct predecessor of TD. The latter case occurs when a new task definition version has been derived and one wishes to adopt the most current workflow definition version within a branch from the old task definition version (as explained above, this is automatically done for the main branch when a task definition version is derived). E.g., this was done in Fig. 4 for version v2.2.1 of the workflow definition 'DomesticTrip'. Since the task interface is not affected by the derivation of a new

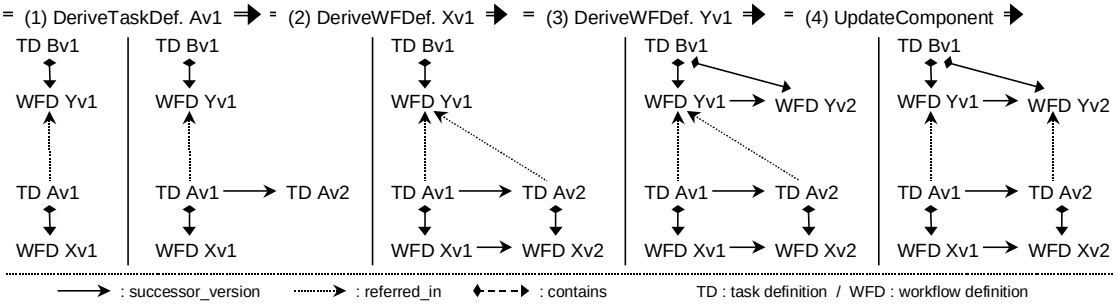

Figure 5: Step-by-step derivation of a new task definition version

workflow definition version, the impact of this operation remains local avoiding any configuration problems.

Among deriving a new version, a new task or workflow definition (version) may also be created by taking a copy of an existing version to avoid modeling similar processes from scratch (as done when joining the 'DomesticTrip' and 'TripAbroad' workflows together into one standard reporting workflow in Fig. 4). This copy operation has no further semantics in our version model.

Another useful operation is the application of the differences between two directly succeeding versions to another variant in order to propagate changes to different variants. This is particularly useful to propagate general workflow changes to locally modified instances, which are represented in our approach in a uniform way as variants as described below. Currently, such a delta is only defined for task graphs. It is represented as a graph rewriting rule which can be applied only if the variant matches the left-hand side of the rule.

In order to use the introduced workflow schema versioning concept for realizing different propagation strategies, further mechanisms are needed (re-bind of workflow instances and controlling of the version state) which are introduced in the next sub-sections.

4.3 Supporting Different Propagation Strategies

Due to the tight coupling of schema and instance elements, a change of a task or workflow definition version immediately affects all its instances (*eager propagation*). On the other hand, a *lazy propagation* strategy can be realized by deriving a new version which has no instances after derivation.

Further, in order to support *selective propagation*, we allow to re-bind workflow instances to a new version which may be selected according to their execution state. An instance may be re-bound to a new version if the old and new version are equal (w.r.t. their content) and the new version succeeds (transitively) the old version w.r.t. the version derivation DAG. Thus, the re-bind operation has to be performed before changes are made and therefore has no consequences to the workflow instances and needs no conversion. Furthermore, when changes are performed on the new version the impact of the modification on the execution states can be analyzed in order to decide whether these change are safe.

Note, that the workflow modeler usually select the workflow instances which should be migrated on the basis of their execution state. However, when he/she makes changes that lead to inconsistent execution states of some instances (in particular, when the compliant property has to be guaranteed), a variant can be created and the relevant instances are re-bound to this variant which is used to define an alternative, compliant workflow for these instances. Thus, selective propagation may be iterated resulting in the definition of different variant and the refinement of the selected instances.

Finally, *local modifications* of a task instance are also realized by the uniform concept of versioning. In this case, a variant of the current task or workflow definition version (depending on the kind of change) is derived and the task instance is (re)bound to this variant. Further, this variant is locked for the use by other instances so that changes of this variant are local to the task instance. Thus, no additional mechanisms are needed to support local modifications and their upward propagation. The latter is done by releasing the variant for general use.

A re-bind may be performed for the workflow definition version only, or for a task definition version. In the latter case, according to the configuration problem described above and the late binding of a task instance to the workflow definition, re-binding of task instances is done consistently to both task and workflow definition version.

4.4 Performing Workflow Schema Changes

After introducing how different propagation strategies are realized by schema versioning, we present a taxonomy of our change primitives and sketch how on-the-fly changes and complex change transactions are supported.

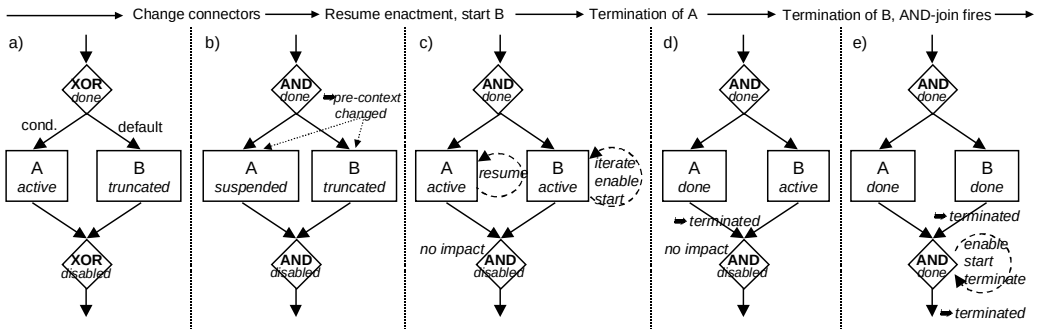

Figure 6: Handling of dynamic changes

Handling of on-the-fly changes: Our approach to dynamic changes of enacting workflow instances is based on encapsulating any basic change operation by a pre-condition which restricts its application and by raising a corresponding event which is handled by the affected instances in order to ensure the behavioral consistency of the execution states. Thus, the applicability of a change operation to a task or workflow definition depends on the execution state of all associated instances and the impact of the operation is handled by event-trigger mechanisms. This approach has been proposed in our previous work on software process management ([10]).

Example: When a new task dependency is added, a corresponding event 'dependency_added' is sent to the relevant task instances which react on this event as defined in their behavior specification. In general, they will reevaluate their activation condition and probably will trigger the enable, disable, suspend, resume or abort transition depending on their current state.

Note that the pre-condition as well as the event handling rules can be redefined for certain task definitions and hence their external behavior can be adapted to its facilities of how reacting on dynamic changes. E.g., humans which perform a manual task may react very flexibly to changes of the context of task (e.g., adding a new preceding task or change of the input parameter), whereas a system-performed task has to be aborted and restarted.

The following taxonomy lists the basic change primitives with their pre-defined enabling condition. The taxonomy is classified by a) the kind of change (add, change, or remove) which is specified only when its distinction is relevant for handling the modification, and b) the model element that is changed.

1) *add task definition:* always allowed
2) *delete task definition*: prohibited if the task definition has yet instances or if an applied occurrence within a workflow definition exists
3) *changes of the interface of a task definition:*
   a) *parameter definitions:* allowed if no instances in state running exists except of optional parameters which may be added at any time and can be removed if the actual parameter doesn't yet contain data.
   b) *behavior definition* ((sub)states and state transitions with their ECA rule): allowed only if the task definition has not yet instances
   c) *business rules:* changes of business rules do not affect directly instances but the workflow definition which have to comply to the rules
4) *changes of the workflow definitions (task body) of a task definition:*
   a) *add workflow definition*: always allowed
   b) *delete workflow definition:* prohibited if the workflow definition was chosen for execution by an instance
   c) *change atomic workflow definition:* allowed if no instances in state active exists
   d) change complex workflow definition
      i) *components:* add always allowed; change & remove if no instance is in state running
      ii) *dependencies/dataflows:* dependent on the behavior definition of the target component; always allowed as a matter of principle

Note, that - in the case of the application of a lazy or selective propagation strategy - in particular changes of the flow structure require only the derivation of a new workflow definition version rather than of the whole task definition. Thus, the configuration problem does not arise for this most frequent type of change. Only, for the change of the task interface (item 3) a whole new task definition version has to be derived.

Performing complex changes: Whereas the structural and behavioral consistency of a workflow change can be ensured by an (atomic) change operation, the semantical correctness cannot be guaranteed because the re-design of a workflow encompasses several change operations. To support guaranteeing the semantical correctness of workflow changes, we introduce a life-cycle diagram for versions, which consists of the three states changeable, released, and expired (see Fig. 7).

The creation and execution of a workflow instance is permitted only, if the corresponding task and workflow definition versions are released. Thus, in the case of lazy propagation, the creation of new instances can be deferred until the new version is released. In the case of eager propagation, when the release of a task or workflow definition version is revoked, the execution of all instances is suspended (i.e., the workflow engine stops scheduling the tasks and hence atomic tasks may still be active; on demand, atomic tasks may be additionally suspended or aborted). After probably analyzing and simulating the new workflow, the version is re-released and hence the execution of all instances is resumed. The engine will update the execution state according to the raised events and will schedule the tasks according to the new schema because of the tight integration of schema and instance. Note, that it is not generally forbidden to perform workflow changes on a released task or work-

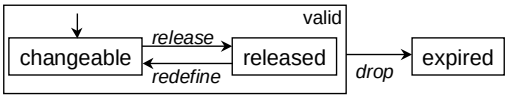

Figure 7: Version life-cycle diagram

flow definition version as described above. Rather, such a change will directly affect all running instances (if any).

Example: In Fig. 6, a conditional branch is changed into a total branch which requires the change of two connectors. To perform this change, the execution of the affected components has been suspended. On resuming the execution, the engine will handle the raised events. Due to the change of the preceding component, the transition conditions of A and B are evaluated. In this case, A is resumed and B, which was previously not selected is activated by triggering first the iterate and subsequently the enable and start transition.

When a task or workflow definition version is no longer valid it enters the state *expired*. In this state, running instances of this version are allowed to complete their execution but it is prohibited to create new instances of this version. This state transition may be done manually when a new version is released or automatically when a specified expiration date is reached.

5. Related Work

Related work on workflow evolution predominantly concentrates on correctness issues of on-the-fly changes (cf. [9, 5, 22]), whereas little attention has been paid to the management of different workflow schema versions.

Our approach is based on experience in designing the DYNAMITE approach ([10]) to software process management and therefore some basic concepts are similar to the approach proposed in this paper. But there are several extensions and differences to the DYNAMITE approach: in DYNAMITE, the declaration and invocation hierarchy is not separated, conditional branching as well as conditional selection of the workflow of a task is not supported, the schema elements are not modeled as first level objects, and schema evolution and versioning is not supported. So far, DYNAMITE focuses on interleaved planning and enacting of dynamically evolving software processes (instances).

The work of Casati et al. [5] focuses on workflow schema changes and is mainly concerned with correctness issues. Casati et al. introduce the compliance property and describe how different migration strategies may be supported. On the other hand, it remains unclear how different workflow schema versions are managed. Further, the contribution neither discusses the decomposition of workflows, nor the separated modeling of task and workflow definitions, nor resulting configuration problems of the management of evolving workflow specifications. W.r.t. to the tight integration of workflow schemata and their instance which has been presented by Casati et al., we follow a similar approach which has been extended by late binding of the enactable workflows.

Bichler et al. [4] propose an approach, which supports different migration strategies and transitional regulations to migrated workflow instances. The correctness of workflow evolution is based on a two-layered schema approach, where a business process is separated from the workflow that implements it. Bichler et al. neither discuss the management of different schema versions nor the decomposition of workflows. Further, only changes of the flow structure are considered. Finally, business process models can only be specified by object life-cycle diagrams which are restricted in their application.

In the EPOS approach [12] to software process management, a task instance net is built up by an AI planner on the basis of a task and product schema and the structure of the product instances. Schema changes are mostly handled by re-planning of the task instances nets. Versioning is applied in EPOS to the whole process model including the schema and its instances. Fine-grained schema versioning and different propagation strategies are not supported. Furthermore, a product-structure independent definition of a process as needed for workflow management is not supported by EPOS.

In SPADE [1], which is based on a high-level petri-net variant, populated copies of a net template are enacted by possibly distributed process engines. Further, late instantiation is used for an activity which acts as a complex transition and is defined by a petri-net. Changing a net template obviously does not affect any net instances and vice versa, so that lazy propagation and local modifications can be supported. On the other hand, eager propagation is supported only by suspending and manually updating an instance. Template changes do not take the execution state into account, and the versioning of the process specifications is not supported in SPADE.

CRISTAL [18] focuses on workflow support for product development processes and applies versioning mechanisms to the so-called meta-objects of its integrated product and workflow model in order to enhance the reuse of such models. The use of workflow versioning for handling dynamic changes is not discussed in the paper.

6. Conclusion

In this paper, we have presented a workflow schema versioning approach, which supports the management of evolving workflow specifications on the schema and instance level. Although the versioning is applied on a class level, no configuration problem arises for the user of the system. The adoption of versioning concepts for workflow schema evolution, its adaptation to the domain of process modeling, and the integration of the versioning mechanisms with process-specific modeling facilities like the separation of ‘what to do’ and ‘how to do’ or like late

binding leads to a powerful concept for the management of evolving workflow specifications.

Future work will be directed to the problem of merging of workflow specification versions, to a more general approach of propagating changes between variants, and to mechanisms which allow for a domain-specific definition of rules to which workflow changes have to comply.

The introduced concepts have been prototypically implemented in the project MOKASSIN. A prototype has been presented at CeBIT'98. The architecture of our system, the problems of distribution in dynamic workflow management and the experience we made will be discussed in a subsequent paper.

References

[1] Bandinelli, S.; Fuggetta, A.; Ghezzi, C.: "Software Process Model Evolution in the SPADE Environment", in *IEEE Transactions on Software Engineering*, Vol. 19, No. 12, 1993; pp. 1128-1144.

[2] Bandinelli, S.; Di Nitto, E.; Fuggetta, A.: "Policies and Mechanisms to Support Process Evolution in PSEEs", in *Proc. of the 3rd Int. Conf. on the Software Process*, 1994; pp 9-20.

[3] Banerjee, J.; Chou, H.-T.; Garza, J.F., Kim, W.; Woelk, D.; Ballou, N.; Kim, H.-J.: "Data Model Issues for Object-Oriented Applications", *in ACM Transactions on Office Information Systems*, 5(1), Jan. 1987; pp. 3-26.

[4] Bichler, P.; Preuner, G.; Schrefl, M.: "Workflow Transparency", *in Proc. of 9th Int. Conf. on Advanced Information Systems Engineering (CAiSE'97)*, Barcelona, Spain, 1997.

[5] Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G.: "Workflow Evolution", in *Proc. of 15th Int. Conf. on Conceptual Modeling (ER'96)*, Cottbus, Germany, 1996; pp. 438-455.

[6] Conradi, R.; Fernström, C.; Fugetta, A.: "A Conceptual Framework for Evolving Software Processes", *ACM SIGSOFT Software Engineering Notes*, Vol. 18, No. 4, 1993; pp. 26-34.

[7] Conradi, R.; Westfechtel, B.: "Version Models for Software Configuration Management", to appear in *ACM Computing Surveys*, 1998

[8] Ellis, C.A.; Nutt, G.J.: "Workflow: The Process Spectrum", in *NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, 1996.

[9] Ellis, C.A.; Keddara, K.; Rozenberg, G.: "Dynamic Change Within Workflow Systems", in *Proc. of the Int. Conf. on Organizational Computing Systems COOCS'95*, Milpitas, CA, 1995; pp. 10-21.

[10] Heimann, P.; Joeris, G.; Krapp, C.-A.; Westfechtel, B.: "DYNAMITE: Dynamic Task Nets for Software Process Management", in *Proc. of the 18th Int. Conf. on Software Engineering*, Berlin, Germany, 1996; pp. 331-341.

[11] Katz, R.H.: "Toward a Unified Framework for Version Modeling in Engineering Databases". *ACM Computing Surveys*, 22(4), Dec. 1990; pp. 375-408.

[12] Jaccheri, M.L.; Conradi, R.: "Techniques for Process Model Evolution in EPOS", *IEEE Transactions on Software Engineering*, Vol. 19, No. 12, 1993, pp. 1145-1156.

[13] Joeris, G.: "Cooperative and Integrated Workflow and Document Management for Engineering Applications", in *Proc. of the 8th Int. Workshop on Database and Expert System Applications, Workshop on Workflow Management in Scientific and Engineering Applications*, Toulouse, France, 1997; pp. 68-73.

[14] Joeris, G.: "Change Management Needs Integrated Process and Configuration Management", in Jazayeri, M; Schauer, H (eds.), *Software Engineering - ESEC/FSE'97*, Proceedings, LNCS 1301, Springer, 1997; pp. 125-141.

[15] Kim, W.; Chou, H.-T.: "Versions of Schema for Object-Oriented Databases, in *Proc. of the 14th Int. Conf. on Very Large Databases (VLDB)*, Los Angeles, USA, Morgan Kaufmann, 1988; pp. 148-159.

[16] Kamath, M.; Ramamrithan, K.: "Bridging the gap between Transaction Management and Workflow Management", in *NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, 1996.

[17] Lautemann, S.-E.: "SchemaVersions in Object-Oriented Database Systems", in *Proc. of the 5th Int. Conf. on Database Systems for Advanced Applications*, Melbourne, Australia, 1997; pp. 323-332.

[18] McClatchey, R. et al.: "Version Management in a Distributed Workflow Application", in *Proc. of the 8th Int. Workshop on Database and Expert System Applications, Workshop on Workflow Management in Scientific and Engineering Applications*, Toulouse, France, 1997; pp. 10-15.

[19] Monk, S.; Sommerville, I.: "A Model for Versioning of Classes in Object-Oriented Databses", in *Proc. of the 10th Britisch National Conf. on Databases (BNCOD)*, Aberdeen, Scotland, LNCS 618, Springer, 1992; pp. 42-58.

[20] Nguyen, M.N.; Conradi, R.: "Towards a Rigorous Approach for Managing Process Evolution", in Montangero, C. (Eds.), *Software Process Technology – 5th European Workshop EWSPT'96*, LNCS 1149, Springer, 1996; pp. 18-35.

[21] Nutt, G.J.: "The evolution toward flexible workflow systems", *Distributed Systems Engineering*, 3(4), Dec. 1996; pp. 276-294.

[22] Reichert, M; Dadam, P.: "ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control", *Journal of Intelligent Information Systems*, 10(2), 1998; pp. 93-129.

[23] Zicari, R.: "A Framework for Schema Updates in an Object-Oriented Database System", in Bancilhon, F.; Delobel, F.; Kanellakis, P. (eds.), *Building an Object Oriented Database System - The Story of O_2*, Morgan Kaufmann, San Mateo, CA, 1992.