# Dynamic Workflow Schema Evolution based on Workflow Type Versioning and Workflow Migration

Revised Version, November 1998

**Markus Kradolfer, Andreas Geppert**
Department of Computer Science, University of Zurich
{kradolf,geppert}@ifi.unizh.ch

**Abstract.** An important yet open problem in workflow management is the evolution of workflow schemas, i.e., the creation, deletion and modification of workflow types in such a way that the schema remains correct. This problem is aggravated when instances of modified workflow types are active at the time of modification, because any workflow instance has to conform to the definition of its type.
This paper presents a framework for dynamic workflow schema evolution that is based on workflow type versioning and workflow migration. Workflow types can be versioned, and a new version can be derived from an existing one by applying one or more modification operations. Our approach also supports the migration of workflow instances from one workflow type version to another one. Besides a taxonomy of schema modification operations, we introduce correctness criteria that must be met by workflow schemas. We also define under which conditions workflow instances can be migrated to new workflow type versions.

## 1 Introduction and Motivation

A *workflow management system (WFMS)* [6] is a software system that provides for the specification, enactment and monitoring of workflows. Every workflow is an instance of a *workflow type* (WFT) that defines steps to be carried out by *processing entities* (humans or software systems) and further aspects such as control and data flow. The collection of the WFT defined at a certain point in time forms the *workflow schema*.

Many of the environments where WFMS are used are constantly changing, because, e.g., new customer requirements have to be met, business processes are reengineered, or new laws demand to change the way business is carried out. These changes often require the workflow schema to be modified accordingly, by defining new workflow types or deleting/modifying existing ones. Existing WFT may have to be modified due to a variety of reasons such as:

- the functionality of a WFT has to be modified due to changing requirements,
- a WFT has to be improved such that instances of it are, e.g., carried out cheaper and/or faster,
- inadequacies detected in the WFT have to be corrected.

Schema evolution is less of a problem when no workflow is active, which could be achieved by waiting until all workflows are terminated, or by aborting active ones. These are however no viable solutions, because workflows typically are of long duration and comprise considerable amounts of work. WFMS should thus support *dynamic schema evolution*, i.e. the modification of the workflow schema in the presence of active workflow instances.

When a workflow schema modification is carried out, effects on the schema as well as on existing workflows have to be considered. In particular, it has to be ensured that schema modifications do not affect the correctness of the schema and of the existing workflows. In our approach, the correctness of the schema is defined by a set of *schema invariants* which have to hold before and after schema modifications. A workflow instance is considered to be correct if it is executed as prescribed by its type.

Whenever a workflow type `wt` is modified, resulting in a new type `wt'`, it might be desirable or required to afterwards view active instances of `wt` under `wt'`. Thus, ideally all active instances can be *migrated* from `wt` to `wt'` upon schema modification. Migration is however not possible in all cases. Depending on the state an instance is in at the time of the modification of its type, migration may not be possible. For instance, if the new type requires that some additional steps have been executed before an instance can legitimately enter a specific state s, then all instances being in state s (or a successor state) cannot be migrated. The fallback solution in such cases can be to abort workflows that cannot be migrated, or to let the user decide what should happen. The first solution is not practical for workflows that have been running for a long time and accumulate large amounts of work, and the second solution is viable only for a very small number of workflows to be handled.

In this paper, we present an approach towards workflow evolution that allows to migrate workflows to modified types whenever possible. Our basic idea is to not update workflow types in place, but to *version* them [9]. Thus, workflow schema modification leads to a new version of the workflow type, and the old version remains in the schema. Those workflows that cannot be migrated can continue their execution under the old version.

The contributions of this approach, thus, are that

- it allows the modification of workflow types and the migration of workflow instances,
- it supports the efficient determination if an instance can be migrated to a certain version,
- it enables the systematic handling of instances that cannot be migrated,
- it allows to log explicitly the evolution history of workflow types.

The remainder of this paper is structured as follows. Section 2 gives an overview of the TRAMs workflow model on which the subsequent evolution concepts are based. A set of schema invariants are defined that specify the correctness of the workflow schema. The notion of workflow execution history is introduced to define *workflow execution invariants*. A taxonomy of schema modification operations is introduced in section 3. Workflow migration is investigated in section 4. *Migration conditions* are derived from the workflow execution invariants that specify when a workflow can be migrated to a certain version. Section 5 discusses relevant related work and section 6 concludes the paper.

## 2 The TRAMs Workflow Model

In this section, an overview of the basic concepts of the TRAMs workflow model [10] is given, including a set of workflow schema invariants. Furthermore, an example workflow type is described. Finally, we define how workflows are executed by introducing the notions of workflow history and workflow correctness.

## 2.1 Workflow Specification and Versioning in TRAMs

The basic concepts of the TRAMs workflow model are illustrated in Fig. 1 (in OMT-notation [15]) and explained in the following. A *workflow type (WFT)*, which is either a *complex*

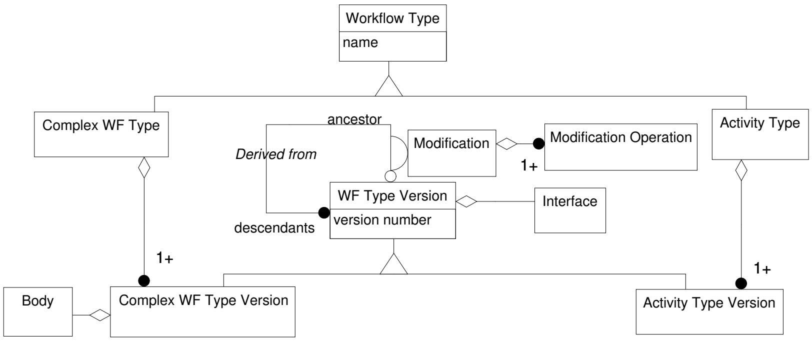



**Figure 1:** Main Concepts of the TRAMs Workflow Model

*workflow type* or an *activity type*, comprises the definition of a globally unique name and a WFT version tree. A *WFT version* defines a version number and an interface. The *interface* defines those aspects of a WFT version that are visible to other WFT versions and can be referred to by them (see Section 2.1.1 for details). A WFT version is either an *activity type version* or a *complex WFT version* that defines a body. The *body* defined by a complex WFT version is not visible to other WFT versions (see Section 2.1.2 for details). Activity type versions do not define bodies since their implementations are hidden in the processing entities. An activity type comprises one or more activity type versions and a complex WFT includes one or more complex WFT versions. A WFT version can be *derived* from another WFT version by applying one or more *modification operations* (see section 3) to it.

A WFT version y that is derived from a WFT version x is called a *descendant* of x and x is called the *ancestor* of y. The WFT versions contained in a WFT form a *version tree*, where parent-child relationships in the tree represent ancestor-descendant (derived-from) relationships. The WFT versions contained in a workflow type are distinguished by associated version numbers. Fig. 2 shows the version tree of a workflow type `wt`. `wt[1]` is derived from `wt[0]` by applying the operation `Op1` to `wt[0]`. By applying `Op2` and `Op3` to `wt[1]`, `wt[2]` is created, and so forth.

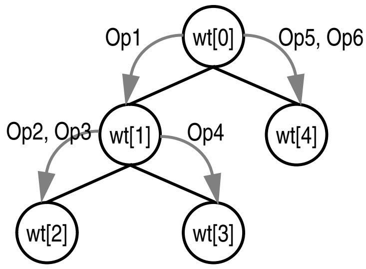



**Figure 2:** Version Tree of a Workflow Type wt

### 2.1.1 Interface

As shown in Fig. 3, the interface defined by a WFT version specifies a set of in(put) parameters and one or more end states with out(put) parameters. The in-parameters model data consumed by a workflow. Each end state represents a different outcome of a workflow, where its out-parameters model the data that is produced by the workflow.

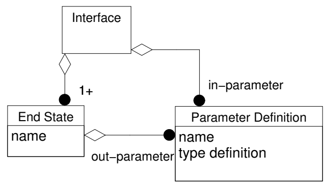



**Figure 3:** Interface of a WFT Version

### 2.1.2 Body

The structure of a complex WFT version is defined by its body (see Fig. 4). Subworkflows of the complex WFT version are specified by *subworkflow declarations*, which in turn consist of the name of the subworkflow (local to the complex WFT) and a corresponding type (i.e., a WFT version).

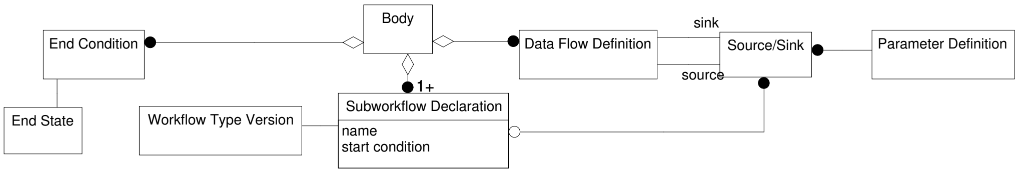



**Figure 4:** Body of a Complex WFT Version

We now introduce the relationships between workflows and WFT versions. A *workflow* is either a basic work step, called *activity*, or a *complex workflow* consisting of one or more (sub)workflows. A workflow x that is contained in another workflow y is called *subworkflow* of y and workflow y is called *superworkflow* of x. Workflows that have the same superworkflow are called *sibling workflows*. A workflow that is not part of another workflow is called *top-level workflow*. An activity is an instance of an activity type version and a complex workflow is an instance of a complex WFT version.

**Data Flow.** Data flows among workflows are specified by *data flow definitions* that associate two workflow parameters. One of the parameters is called the *source* and the other one the *sink*, meaning that parameter values “flow” from the source to the sink. Data flows can be specified by complex WFT versions.

**Control Flow.** Since we assume that processing entities are autonomous, activities which are carried out by them are regarded as black boxes whose internal structures are not visible. This means that intermediate execution states of activities are not visible for the WFMS and that the WFMS can only control the start of activities. In TRAMs, each workflow runs through the execution states `Initiated` and `Executing` and finally ends in one of its end states. The execution state of a workflow x can be referred to by the construct `state_name(x)`, which evaluates to true if x is in the state named `state_name`. Otherwise, it evaluates to false.

Control flow is specified in a declarative way by defining guards (logical expressions similar to the ones used in FlowMark [11]) on the state transitions of workflows. The following guards can be defined for workflows. For each subworkflow declaration, a *start condition* can be defined, which is a guard on the transition leading from state `Initiated` to state `Executing`. A complex WFT version can specify an *end condition* for each of its end states which are guards on the transitions leading from `Executing` to end states.

### 2.2 Workflow Type Example

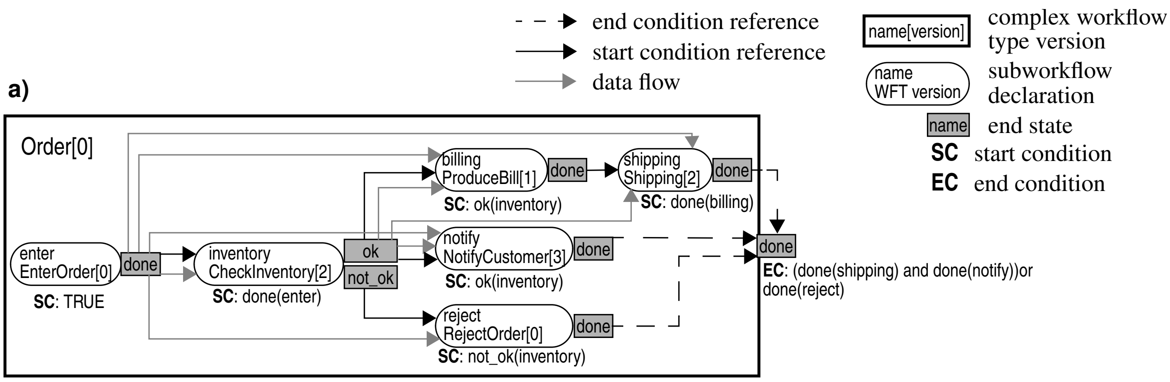
We use as a running example the complex WFT `Order`, that defines the processing of a purchase order in a company. Fig. 5a) shows `Order[0]`, a complex WFT version contained in `Order`. `Order[0]` is represented by a black rectangle that contains a special kind of graph whose oval nodes represent subworkflow declarations. The end states of the subworkflow declarations of `Order[0]` and of `Order[0]` itself are represented by gray rectangles. For each subworkflow declaration, a start condition is defined and black solid arrows indicate which end states are referenced by which start conditions. For the end state of `Order[0]`, an end condition is defined and black dashed arrows indicate which end states are referenced by the end condition. Fig. 5b) defines the interfaces of `Order[0]` and of its subworkflow declarations. Gray arrows visualize the data flows defined in Fig. 5c).

As the first step to be carried out, `Order[0]` defines a subworkflow `enter` (of WFT version `EnterOrder[0]`) that includes the creation of an electronic dossier by a clerk, containing information about the customer and the product to be ordered. After `enter` has ended, `inventory` can be started that checks if the requested product is available in the inventory. In case `inventory` ends in state `ok`, `billing` (charge the customer) and `notify` (inform the customer that the order has been accepted) can be started in parallel. If `inventory` ends in state `not_ok`, `reject` can be started that includes the writing of a rejection letter to the customer. The order processing workflow ends in state `done` if either `shipping` and `notify` both are in state `done` or `reject` is in state `done`.

### 2.3 Workflow Schema Invariants

In our approach, the correctness of the schema is defined by a set of schema invariants (this is similar to database schema evolution, e.g., for object-oriented DBMSs [2]). Invariants are conditions over the schema that have to hold before and after schema modifications. Following is a list of the schema invariants:

1. *Distinct name invariants* require that names used within the same context must be different:
   - All workflow types (activity types and complex workflow types) have distinct names.




**b)** Interface Definitions

| WFT Version | In-Parameters | End State | Out-Parameters |
|---|---|---|---|
| Order[0] | - | done | - |
| EnterOrder[0] | - | done | string product_descr, integer customer_id |
| CheckInventory[2] | string product_descr | ok | integer product_id |
| | | not_ok | - |
| ProduceBill[1] | integer customer_id, integer product_id | done | - |
| NotifyCustomer[3] | integer customer_id, integer product_id | done | - |
| Shipping[2] | integer customer_id, integer product_id | done | - |
| RejectOrder[0] | integer customer_id, string product_descr | done | - |

**c)** Data Flow Definitions

| Source | Sinks |
|---|---|
| Order[0].enter.done.product_descr | Order[0].inventory.product_descr, Order[0].reject.product_descr |
| Order[0].enter.done.customer_id | Order[0].billing.customer_id, Order[0].notify.customer_id, Order[0].shipping.customer_id, Order[0].reject.customer_id, |
| Order[0].inventory.ok.product_id | Order[0].billing.product_id, Order[0].notify.product_id, Order[0].shipping.product_id, |

**Figure 5:** WFT Version `Order[0]`

- All end states and all in-parameters defined by a WFT version have distinct names.
- All out-parameters of an end state defined by a WFT version have distinct names.
- The subworkflows specified by a complex WFT version have distinct names.

2. The *end state invariant* requires that a WFT version defines one or more end states.
3. *Data flow definition invariants* constrain the legitimate set of data flow definitions. Data flow definitions can only be specified by complex WFT versions. A data flow specified by a complex WFT version x defines the following flow of data:
   - from an in-parameter of x to
     - an in-parameter of a subworkflow defined by x or
     - an out-parameter of x or
   - from an out-parameter of a subworkflow s1 defined by x to
     - an in-parameter of a sibling workflow of s1 or
     - an out-parameter of x

   Furthermore, an out-parameter of a complex WFT version and an in-parameter of a subworkflow declaration can be the sink in at most one data flow definition.
4. *Start condition invariants* define what can be used in the start condition of subworkflow declarations. The start condition of a subworkflow s1 that is defined by a complex WFT version x may only involve references to:
   - in-parameters of x, and
   - out-parameters and execution states of sibling workflows of s1.
5. *End condition invariants* define the appropriate parts of end conditions defined by complex WFT versions. The end condition of a complex WFT version x may only involve references to:
   - in-parameters of x, and
   - execution states and out-parameters of subworkflows defined by x.
6. *Type invariants* require that types used in workflow definitions are valid:
   - A subworkflow declaration must be associated to an existing WFT version.
   - The type defined by a parameter has to be valid.
7. *Version invariants* formulate conditions that have to be fulfilled by WFT versions:
   - Every WFT version is contained in the version tree of exactly one WFT.
   - The version tree of a WFT contains at least one version.
   - The versions belonging to the same version tree must have distinct version numbers.
   - A version is either the root of a version tree or it references an existing version from which it is derived from.

### 2.4 Workflow Execution History

In this section, the notion of *workflow execution history* is introduced. The workflow execution history records all the execution state changes of workflows. Workflow execution state changes are regarded as events; thus the workflow execution history is an event history. The correctness of workflows is defined by a set of execution invariants that have to be preserved by the execution histories of workflows.

When a workflow changes its execution state, the workflow is said to generate an event. At the point in time a workflow w enters the execution state `Executing`, it generates a *start event* ($Start_w$), and upon entering an end state it generates an *end event* ($End_w$). An *event occurrence* has several attributes (see Table 1). Attributes of an event e can be referred to by <attribute_name>(e). For example, the timestamp of e (i.e. the point in time e occurred) is referred to by timestamp(e). If e is a start (end) event generated by a workflow w, parameter_value_list(e) comprises the values of the in- (out-) parameters of w at the time e occurs.

The execution of a set of workflows W is represented by the history H which is a *partial order* $H = (E, \angle)$, where E is the set of events invoked by the workflows in W and $\angle$ is an irreflexive, transitive binary relation on E such that $\forall e_1, e_2 \in E\ (\mathrm{timestamp}(e_1) < \mathrm{timestamp}(e_2)) \Leftrightarrow (e_1 \angle e_2)$. A partial order $H' = (E', \angle')$ is a restriction of H on domain E′ if $E' \subseteq E$ and $\forall e_1, e_2 \in E'\ (e_1 \angle e_2) \Rightarrow (e_1 \angle' e_2)$. H′ is a *prefix* of H, if H′ is a restriction of H and $\forall e \in E'\ \forall e_1 \in E\ (\neg(e \angle e_1) \Rightarrow (e_1 \in E'))$. $H[e_1] = (E[e_1], \angle[e_1])$ is a prefix of H with respect to $e_1$, where $E[e_1]$ contains all events of E that did not occur after the event $e_1$, i.e. $\forall e_1 \in E\ (E[e_1] = \{e \in E \mid \neg(e_1 \angle e)\})$. The history of events which are generated up to a given point in time ct is referred to as the *current history* $H_{ct} = (E_{ct}, \angle_{ct})$, where $E_{ct} = \{e \in E \mid \mathrm{timestamp}(e) \leq ct\}$. The event history of a specific workflow is a projection of H, consisting of the events invoked by the workflow and its subworkflows. Table 1 shows the history of an instance of `Order[0]` that rejects an order.

| id | kind | WFT version | subworkflow declaration | id superworkflow | parameter value list | timestamp |
|---|---|---|---|---|---|---|
| 1 | Start | Order[0] | - | - | - | 25 |
| 2 | Start | EnterOrder[0] | Order[0].enter | 1 | - | 26 |
| 2 | done | EnterOrder[0] | Order[0].enter | 1 | product_descr = “A 342 L“, customer_id = 78 | 232 |
| 3 | Start | CheckInventory[2] | Order[0].inventory | 1 | product_descr = “A 342 L“ | 233 |
| 3 | not_ok | CheckInventory[2] | Order[0].inventory | 1 | - | 349 |
| 4 | Start | RejectOrder[0] | Order[0].reject | 1 | product_descr = “A 342 L“, customer_id = 78 | 350 |
| 4 | done | RejectOrder[0] | Order[0].reject | 1 | - | 380 |
| 1 | done | Order[0] | - | - | - | 381 |

**Table 1:** History of an Instance (ID 1) of Order[0]

At the time a workflow w issues an event e, the value of the start condition defined for w at timestamp(e) is denoted by Cond_Start(e) and the value of the end condition defined for w at timestamp(e) is denoted by Cond_End(e). For the start and end conditions of workflows, the following convention applies. If no start condition (end condition) is defined for a workflow (end state) at a certain point in time, the start (end) condition is assumed to evaluate to true at this point in time.

Using the concepts introduced so far, the *correctness of a workflow* (execution) can be defined. Informally, a workflow w is considered to be correct if

- w is executed as prescribed by its WFT version and
- in case w is a subworkflow, it obeys its corresponding subworkflow declaration.

These two conditions are implications of the following *workflow execution invariants*. A workflow w is considered to be correct if the following invariants hold:

**Constraint on the number of events a workflow can invoke.** w invokes at most one start event and one end event: $(\mathrm{Start}_w \in E_{ct} \Rightarrow \neg(\mathrm{Start}'_w \angle_{ct} \mathrm{Start}_w)) \wedge (\mathrm{End}_w \in E_{ct} \Rightarrow \neg(\mathrm{End}'_w \angle_{ct} \mathrm{End}_w))$.

**Constraints on the start event.** In case w is a subworkflow of the workflow super_wf, it can only start if

- super_wf is in state `Executing`: $\mathrm{Start}_w \in E_{ct} \Rightarrow ((\mathrm{Start}_{super\_wf} \angle_{ct} \mathrm{Start}_w) \wedge (\mathrm{End}_{super\_wf} \notin E[\mathrm{Start}_w]))$,
- the start condition corresponding to w holds:
  $\mathrm{Start}_w \in E_{ct} \Rightarrow \mathrm{Cond\_Start}(\mathrm{Start}_w)$ , and
- it is defined by the WFT version of super_wf:
  $\mathrm{Start}_w \in E_{ct} \Rightarrow (\mathrm{wft\_version}(super\_wf) = \mathrm{defined\_by}(\mathrm{subwf\_decl}(\mathrm{Start}_w)))$.
  subwf_decl is a partial function that maps an event generated by a workflow w to the subworkflow declaration that corresponds to w. defined_by maps a subworkflow declaration to the WFT version the subworkflow declaration is defined by, and wft_version is a function that maps a workflow w to the WFT version w is an instance of.

Moreover, the start event of w has to comprise the values of those in-parameters (defined by the WFT version of w) that are referenced by a data flow definition.

**Constraints on the end event.** w can only invoke an end event $End_w$ if

- it has invoked a start event before: $End_w \in E_{ct} \Rightarrow (Start_w \angle_{ct} End_w)$,
- $End_w$ is defined by the WFT version of w: defined_by($End_w$, wft_version(w))
- the end condition corresponding to $End_w$ holds:
  $End_w \in E_{ct} \Rightarrow Cond\_End(End_w)$, and
- in case w is a subworkflow of the workflow super_wf, w is defined by the WFT version of super_wf:
  $End_w \in E_{ct} \Rightarrow (wft\_version(super\_wf) = defined\_by(subwf\_decl(End_w)))$.

Additionally, an end event $End_w$ that is invoked by w has to comprise the values of those out-parameters defined for $End_w$ (by the WFT version of w) that are referenced by a data flow definition.

**Constraints on the Parameter Values.** The parameter values of w have to comply to the data flows defined.

- In case w is a subworkflow of the workflow super_wf, the following has to hold for each of its in-parameters p: In case there is a data flow defined whose sink is p, p has to have the same value as the source defined by the data flow:
  $Start_w \in E_{ct} \Rightarrow (\forall p \in parameter\_value\_list(Start_w)\ \forall df \in data\_flow\_defs(super\_wf)\ (sink(df) = p) \Rightarrow (value(source(df), timestamp(Start_w)) = value(p, timestamp(Start_w))))$.
  data_flow_defs maps a workflow w (or an event generated by w) to the set of data flows defined for w. sink maps a data flow definition dfd to the sink dfd defines and source maps dfd to the source dfd defines. value maps a parameter p and a point in time t to the value p has at t.
- At the time w ends, the following has to hold for each of its out-parameters p: In case there is a data flow defined whose sink is p, p has to have the same value as the source defined by the data flow:
  $End_w \in E_{ct} \Rightarrow (\forall p \in parameter\_value\_list(End_w)\ \forall df \in data\_flow\_defs(End_w)\ (sink(df) = p) \Rightarrow (value(source(df), timestamp(End_w)) = value(p, timestamp(End_w))))$.

**Constraints on the Subworkflows.** The execution invariants have to hold for the subworkflows possibly contained in w.

# 3 Workflow Schema Modification

In this section, we define a taxonomy of workflow schema modification operations for the workflow model introduced in section 2.

## 3.1 Taxonomy of Workflow Schema Modification Operations

The schema modification operations supported are categorized into operations for the creation (1) and deletion (2) of WFT versions, and operations to create (3) and delete (4) WFT. Following, a taxonomy of the workflow schema modification operations is given.

1 Add a new WFT version
  - 1.1 Changes to the interface of a WFT version
    - 1.1.1 Changes to a parameter definition
      - 1.1.1.1 Add a new parameter definition
      - 1.1.1.2 Drop an existing parameter definition
    - 1.1.2 Changes to an end state
      - 1.1.2.1 Add a new end state
      - 1.1.2.2 Drop an existing end state
  - 1.2 Changes to the body of a complex WFT version
    - 1.2.1 Changes to a subworkflow declaration
      - 1.2.1.1 Add a new subworkflow declaration
      - 1.2.1.2 Drop an existing subworkflow declaration

1.2.1.3 Modify the start condition of a subworkflow declaration
1.2.2 Changes to a data flow definition of a complex WFT version
1.2.2.1 Add a data flow definition
1.2.2.2 Drop an existing data flow definition
1.2.3 Changes to an end condition of a complex WFT version
1.2.3.1 Modify an end condition

2 Drop an existing WFT version
3 Add a new workflow type
4 Drop an existing workflow type

A *schema modification* is the process of executing one or more of the above schema modification operations as an atomic unit. A new WFT version is created by applying one or more operations of category 1 to an existing WFT version. Operation 2 (op. 2), dropping an existing WFT version, can only be applied to leaves of version trees. The root of version trees cannot be deleted. Furthermore, a WFT version can only be deleted if no instances of it exist. Adding a new workflow type by op. 3 includes the creation of the first version of the workflow type. A workflow type including all its versions can only be dropped (op. 4) if no instances of any of its versions exist.

Note that there are no operations provided for modifying the body of activity type versions since (as already mentioned in section 2.1) the implementations of the activity type versions are hidden in the processing entities and are therefore outside the scope of the WFMS.

It has to be ensured that before and after a schema modification, the schema is correct, i.e. the schema invariants are preserved. Otherwise, the modification is rejected. As an example, consider op. 1.1.1.2, dropping a parameter definition `pd` from a WFT version `wt[x]`, resulting in a new version `wt[y]`. If `pd` is referenced in `wt[x]` by data flows, start conditions, or end conditions, the new version `wt[y]` contains invalid references. Thus, in order to prevent an invalid schema, the schema modification also has to include operations that remove the invalid references to `pd`.

Table 2 indicates for each operation which schema invariants may potentially be affected by the execution of the operation.

| Operation | Distinct Name Invariants | End State Invariant | Data Flow Definition Invariants | Start Condition Invariants | End Condition Invariants | Type Invariants | Version Invariants |
|---|---|---|---|---|---|---|---|
| 1.1.1.1 | ✓ | | | | | ✓ | ✓ |
| 1.1.1.2 | | | ✓ | ✓ | ✓ | | ✓ |
| 1.1.2.1 | ✓ | | | | | | ✓ |
| 1.1.2.2 | | ✓ | ✓ | ✓ | ✓ | | ✓ |
| 1.2.1.1 | ✓ | | | ✓ | | ✓ | ✓ |
| 1.2.1.2 | | | ✓ | ✓ | ✓ | | ✓ |
| 1.2.1.3 | | | | ✓ | | | ✓ |
| 1.2.2.1 | | | ✓ | | | | ✓ |
| 1.2.2.2 | | | | | | | ✓ |
| 1.2.3.1 | | | | | ✓ | | ✓ |
| 2 | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 4 | | ✓ | ✓ | ✓ | ✓ | ✓ | |

**Table 2:** Schema Invariants Potentially Affected by Schema Modification Operations

### 3.2 Workflow Example

After several instances of `Order[0]` have been started, the company management decides to improve the order processing in the following ways. In order to cut the shipping costs, an additional step `planning` that includes the selection of the optimal means of transportation and the coordination among different shippings has to be executed prior to shipping. Furthermore, it has turned out that sending notification letters to customers (step `notify`) can be omitted since the information contained in the letters is also contained in the bills that are sent to the customers. Finally, an additional end state `rejected` has to be introduced such that order processing workflows that reject an order end in the end state `rejected`.

In order to meet the above requirements, a new version `Order[1]` (see Fig. 6) is derived from `Order[0]` by applying the following operations to `Order[0]`:

1 add the end state `rejected` with the end condition `done(reject)` (op. 1.1.2.1),
2 add the subworkflow declaration `planning` of type `PlanShipping[1]` (op. 1.2.1.1) with the start condition `ok(inventory)`,
3 delete the subworkflow declaration `notify` (op. 1.2.1.2),
4 change the start condition of `shipping` to `done(planning) and done(billing)` (op.1.2.1.3),
5 add data flow definitions (op. 1.2.2.1) specifying flows of data from `inventory.ok.product_id` to `planning.product_id` and from `enter.done.customer_id` to `planning.customer_id`,
6 delete the data flow definitions (op. 1.2.2.2) specifying flows of data from `enter.done.customer_id` to `notify.customer_id` and from `inventory.ok.product_id` to `notify.product_id`,
7 modify the end condition of the end state `done` to `done(shipping)` (op. 1.2.3.1).

It has to be ensured that the schema invariants hold after `Order[1]` is added to the schema. For example, for the first of the above operations (adding the end state `rejected`), it has to be ensured that

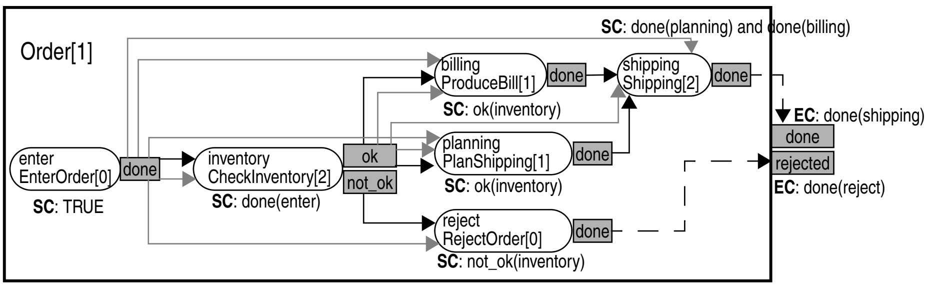
- there is not already an end state named `rejected` (distinct name invariants) and
- the end condition of `rejected` is valid (end condition invariants).

The other operations have to be checked accordingly with respect to the schema invariants that are affected.

## 4 Workflow Migration

After a new WFT version is derived from an existing one, it is often desired to view all or some workflow instances of the old version as instances of the new version. This is accomplished by *workflow migration*. The migration of a workflow `w` from a WFT version `wt[x]` to another

**a)**




**b)** Interface Definitions

| WFT Version | In-Parameters | End State | Out-Parameters |
|---|---|---|---|
| Order[1] | - | done | - |
| | | rejected | - |
| PlanShipping[1] | integer customer_id, integer product_id | done | - |

**c)** Data Flow Definitions

| Source | Sinks |
|---|---|
| Order[0].enter.done.product_descr | Order[0].inventory.product_descr, Order[0].reject.product_descr |
| Order[0].enter.done.customer_id | Order[0].billing.customer_id, Order[0].planning.customer_id, Order[0].shipping.customer_id, Order[0].reject.customer_id, |
| Order[0].inventory.ok.product_id | Order[0].billing.product_id, Order[0].planning.product_id, Order[0].shipping.product_id, |

**Figure 6:** WFT Version `Order[1]`

version `wt[y]` is the process of modifying w in such a way that it is an instance of `wt[y]` after migration. The modification of w applies to w (i.e., at the instance level) the operations through which `wt[y]` has been derived from `wt[x]`.

Whether a workflow actually can be migrated to a certain version, depends on the current state of the workflow. Consider a workflow w that is to be migrated from `wt[x]` to `wt[y]` at the point in time ct. Basically, w can be migrated to `wt[y]` if w has been executed according to `wt[y]` until ct, i.e. the history of the workflow complies to `wt[y]` at ct. A simple but inefficient approach to determine whether w can be migrated to `wt[y]` is to check every event contained in the history of w for compatibility with `wt[y]`. This approach is inefficient whenever large numbers of complex workflows have to be migrated. A more efficient approach is to consider the operations by which `wt[y]` is derived from `wt[x]`. For each schema modification operation Op, a *migration condition* is derived from the execution invariants. A migration condition of an operation Op specifies if a workflow can be migrated correctly from `wt[x]` to `wt[y]` at a certain point in time, where `wt[y]` is derived from `wt[x]` by applying Op to `wt[x]`. Thus, a workflow w can be migrated from `wt[x]` to `wt[y]` at ct if all the migration conditions of the operations by which `wt[y]` has been derived from `wt[x]` hold for w at ct.

For instance, consider a WFT version `wt[y]` that is derived from `wt[x]` by deleting a subworkflow declaration `sub_decl` from `wt[x]`. The migration condition of this operation requires that the subworkflow s corresponding to `sub_decl` of the workflow w to be migrated has not yet started. Thus, instead of analyzing every event of the history of w, the history only has to be queried if it contains the start event of s.

So far, the migration of workflows from a WFT version v to direct descendants of v has been considered. In order to illustrate the need for migration to other versions, consider the version tree in Fig. 2. Assume that the version `wt[4]` just has been created and that the current instances of `wt[3]` have to be migrated to `wt[4]`. In this case, instances of `wt[3]` can be migrated to `wt[4]` by first migrating them to `wt[1]`, then to `wt[0]` and finally to `wt[4]`. Thus, a workflow can potentially be migrated to any other version in its corresponding version tree by repeatedly migrating to the ancestor or descendant of its current WFT version. Intuitively, a workflow can be migrated from `wt[y]` to `wt[x]`, where `wt[x]` is the ancestor of `wt[y]` by applying operations that undo the effects of the operations by which `wt[y]` has been derived from `wt[x]`. This can be achieved by using inverse operations, where an inverse operation $Op^{-1}$ of an operation Op undoes the effects of Op. Clearly, for each operation of category 1 an inverse operation exists. For instance, the inverse operation of adding a parameter definition is dropping the parameter definition. Thus, returning to our example in Fig. 2, a workflow is migrated from `wt[3]` to `wt[4]` by applying the operations $Op4^{-1}$, $Op1^{-1}$, Op5, and Op6 to it.

An actual workflow migration is carried out in four steps. In the first step, the workflow to be migrated is suspended which means that the workflow and all its subworkflows are frozen in the execution state they are currently in. Thereafter, it is determined whether the workflow can actually be migrated to the desired version by verifying if the migration conditions hold. In case migration is possible, the workflow is modified by applying the according operations. Eventually, the workflow is resumed.

### 4.1 Migration Conditions

In the following, the migration conditions of the schema modification operations belonging to category 1 in the taxonomy of section 3 are defined. The migration conditions are derived from the workflow execution invariants of section 2.4. A proof of the correctness of the migration conditions can be found in appendix A.

**Adding parameter definitions (op. 1.1.1.1).** A parameter definition `pd` is added to a WFT version `wt[x]`, resulting in a new version `wt[y]`.

*Migration Condition (MC):* A workflow can be migrated at any time from wt[x] to wt[y].

**Deleting parameter definitions (op. 1.1.1.2).** A parameter definition is removed from a WFT version wt[x], resulting in a new version wt[y].

*MC:* A workflow can be migrated at any time from wt[x] to wt[y].

**Adding new end states (op. 1.1.2.1).** An end state is added to a WFT version wt[x], resulting in a new version wt[y].

*MC:* A workflow can be migrated at any time from wt[x] to wt[y].

**Deletion of end states (op. 1.1.2.2).** An end state EndZ is removed from a WFT version wt[x], resulting in a new version wt[y].

*MC:* A workflow can be migrated from wt[x] to wt[y] at time ct if it has not entered the end state EndZ until ct: $EndZ_w \notin E_{ct}$.

**Adding subworkflow declarations (op. 1.2.1.1).** A subworkflow declaration s_decl is added to a complex WFT version wt[x], resulting in a new version wt[y].

*MC*: A workflow can be migrated at any time from wt[x] to wt[y].

**Deletion of subworkflow declarations (op. 1.2.1.2).** The subworkflow declaration s_decl is deleted from a complex WFT version wt[x], resulting in a new version wt[y].

*MC*: A complex workflow can be migrated from wt[x] to wt[y] at time ct if its subworkflow s that corresponds to s_decl has not started until ct: $Start_s \notin E_{ct}$.

**Modifying start conditions (op. 1.2.1.3).** The start condition of a subworkflow declaration s_decl defined by a complex WFT version wt[x] is modified, resulting in a new version wt[y].

MC: A complex workflow c_wf can be migrated from wt[x] to wt[y] at time ct, if the subworkflow s that corresponds to s_decl has not started until ct or if s has already started and the modified start condition (NewCond_Start) holds at the start time of s:
$Start_s \in E_{ct} \Rightarrow NewCond\_Start(Start_s)$.

**Adding data flow definitions (op. 1.2.2.1).** A data flow definition is added to a complex WFT version wt[x], resulting in a new version wt[y].

*MC:* Whether a complex workflow c_wf of type wt[x], can be migrated, depends on the data sink defined by the data flow definition. If the data sink refers to an in-parameter of a subworkflow s in c_wf, c_wf can be migrated to wt[y] at time ct, if s has not yet started until ct ($Start_s \notin E_{ct}$.). Otherwise, if the data sink refers to an out-parameter of c_wf, c_wf can be migrated from wt[x] to wt[y] at time ct, if c_wf has not yet ended until ct ($End_{c\_wf} \notin E_{ct}$.).

**Deletion of existing data flow definitions (op. 1.2.2.2).** A data flow definition is dropped from a complex WFT version wt[x], resulting in a new version wt[y].

*MC:* the migration condition is identical to that of op. 1.2.2.1 (adding a data flow definition)

**Modifying end conditions (op. 1.2.3.1).** The end condition for the end state EndZ of a CWFT version wt[x] is modified, resulting in a new version wt[y].

MC: A complex workflow c_wf can be migrated from wt[x] to wt[y] at time ct if c_wf has not ended until ct in the state EndZ or if c_wf has ended in the state EndZ and the modified end condition (NewCond_End) holds at the time c_wf has entered EndZ:
$End_{c\_wf} \in E_{ct} \Rightarrow NewCond\_End(End_{c\_wf})$.

### 4.2 Example: Migrating Order Workflows

After Order[1] has been added to the schema, the question is which instances of Order[0] can be migrated to Order[1]. In order to determine which instances can be migrated, the migration conditions of the operations by which `Order[1]` has been derived from `Order[0]` (see Section 3.2) have to be considered.

1 Add the end state `rejected` with the end condition `done(reject)` (op. 1.1.2.1). The migration condition of this operation is true for all instances of `Order[0]`.
2 Add the subworkflow declaration `planning` of type `PlanShipping[0]` (op. 1.2.1.1) with the start condition `ok(inventory)`. The migration condition of this operation is true for all instances of `Order[0]`.
3 Delete the subworkflow declaration `notify` (op. 1.2.1.2). The migration condition for this operation is true for all instances of `Order[0]` whose subworkflow `notify` has not yet started.
4 Change the start condition of `shipping` to `done(planning) and done(billing)` (op.1.2.1.3).The migration condition of this operation requires that a) either `shipping` has not yet started or b) the new start condition holds at the time `shipping` has started. Since b) does not hold because `planning` is newly added and can therefore not have started already, the migration condition of this operation is a).
5 Add data flow definitions (op. 1.2.2.1) from `inventory.ok.product_id` to `planning.product_id` and from `enter.done.customer_id` to `planning.customer_id`. The migration conditions of these operations hold, if `planning` has not yet started. Thus, the migration conditions are true for all instances of `Order[0]`, since `planning` has not yet started in the instances of `Order[0]`.
6 Delete the data flow definitions (op. 1.2.2.2) specifying flows of data from `enter.done.customer_id` to `notify.customer_id` and from `inventory.ok.product_id` to `notify.product_id`. The migration conditions for these operations hold, if `notify` has not yet started.
7 Modify the end condition of the end state `done` to `done(shipping)` (op. 1.2.3.1). The migration condition of this operation requires that either a) the workflow to be migrated is not in the end state `done` or b) the new end condition holds at the time the workflow has ended in `done`. Since b) does not hold, the migration condition of this operation is a).

Thus, taken the above conditions together, those workflows can be migrated from `Order[0]` to `Order[1]` that are not in the end state `done` and whose subworkflows `shipping` and `notify` have not yet started. As an example, consider the migration at the point in time 400 of two instances of `Order[0]` whose execution histories are shown in Table 1 and Table 3. The workflow whose history is shown in Table 1 cannot be migrated to `Order[0]` since it is already in the end state `done`. The other workflow (see Table 3) can be migrated to `Order[1]` since it is not in the end state `done` and `notify` and `shipping` have not yet started.

| id | kind | WFT version | subworkflow declaration | id super-workflow | parameter value list | time-stamp |
|---|---|---|---|---|---|---|
| 7 | Start | Order[0] | - | - | - | 13 |
| 8 | Start | EnterOrder[0] | Order[0].enter | 7 | - | 14 |
| 8 | done | EnterOrder[0] | Order[0].enter | 7 | product_descr = “B 65 X“, customer_id = 54 | 300 |
| 9 | Start | CheckInventory[2] | Order[0].inventory | 7 | product_descr = “B 65 X“ | 312 |
| 9 | ok | CheckInventory[2] | Order[0].inventory | 7 | product_id = 342 | 380 |
| 10 | Start | ProduceBill[1] | Order[0].billing | 7 | customer_id = 54, product_id = 342 | 384 |
| 10 | done | ProduceBill[1] | Order[0].billing | 7 | - | 398 |

**Table 3:** History of an Instance (ID 7) of `Order[0]`

### 4.3 Further Issues

It is often desired not to migrate all instances of a certain version to another one, but only those instances fulfilling certain criteria regarding, e.g., workflow execution states or workflow parameter values. In our approach, instances of a certain WFT version can be selected by applying a *selection condition* (similar to start and end conditions defined for workflows) to a WFT version. For instance, if one would like to select all instances of the WFT version `Order[0]` whose subworkflow `enter` has ended with a `customer_id` value unequal to 40, this can be done by applying the selection condition `done(enter) and enter.done.customer_id != 40` to `Order[0]`. Instances selected in this way can then be migrated to the desired version, where only those instances are actually migrated whose migration conditions hold.

In the following, we briefly discuss different possibilities for a workflow that should be migrated to a certain version but whose migration conditions do not hold. One possibility is to abort the workflow and to restart it under the new version. Since valuable work might be lost using this approach, a better alternative is to roll back the workflow to a state, where migration is possible. This would require the support of a partial rollback mechanism.

## 5 Related Work

This section considers relevant related work in the fields of object-oriented database systems (OODBS), process-centered software engineering environments and workflow management.

Workflow schema evolution is similar to schema evolution in OODBS in that schema modifications of OODBS should be supported in the presence of instances (objects) without requiring a system shut down. In [13] and [16], class versioning is used for schema evolution in OODBS. However, the problem of migrating objects (in OODBS) and the problem of migrating workflows (in WFMS) differ significantly. For instance, when workflows are to be migrated, in contrast to objects, their execution histories have to be considered in order to find out whether migration is possible.

SPADE [1] and EPOS [8] are prominent examples of process-centered software engineering environments that support dynamic process schema evolution. In the SPADE environment, a language based on high-level Petri nets is used for process modeling. The language is reflective in that both process instances as well as process schemas are represented as tokens in the Petri nets that can be manipulated. SPADE differs from our approach in that no specific modification operations are provided. The actual modification of process instances and process schemas is only modeled by a transition in the Petri-net which represents the invocation of an editor tool by which the modifications can be carried out.

EPOS offers a reflective object-oriented process specification language that supports metaclasses, classes and instances. EPOS is built on top of a database management system that supports versioning such that process schemas can be versioned. EPOS differs from our approach in that no strict correctness requirements exist, e.g. modifications are allowed even if inconsistencies may result. Furthermore, no specific process schema modification operations are provided.

Following, related work in the area of workflow management is considered. [5] proposes a workflow evolution approach, where workflows are modeled by special kinds of Petri nets that can be modified by replacing a subnet of an existing net by another one. A correctness criteria, defining valid workflow execution sequences, specifies which modifications are correct. However, only control flow modifications are considered and no modification operations are provided in this approach.

In [14] a framework to support structural changes of individual workflows (schema evolution is not considered) is presented. Some modification operations are described that can be applied to workflows. The modification of a workflow is allowed, if certain correctness criteria hold which consider the structure as well as the state of the workflow at modification time. In contrast to our approach, workflows do not have to be in strict accordance with their type, since workflow instances can be modified without their type being modified as well.

The workflow evolution approach in [4] provides a set of modification operations that allow to change a workflow type and to migrate existing workflows to the modified type. The modification operations provided allow only to change the execution order of workflows, but not the flow of data among different workflows as in our approach. Furthermore, there is no specific concept for workflow type versioning.

In [3] a two schema architecture for WFMS is proposed. In this approach, a business process schema and a workflow schema exist. The business process schema consists of a set of business process specifications and the workflow schema contains a set of workflow specifications. A business process specification defines a set of alternative ways to achieve the objectives of a business process. A set of workflow specifications defines alternative implementations of the corresponding business process specification. The migration of workflows among different workflow specifications belonging to the same business process specification is supported, but changes of business process specifications themselves are not possible. Thus, changes to workflows are restricted to the ones foreseen in their corresponding business process specifications.

In [12] a language is proposed that allows to specify evolution policies for the instances of a workflow specification (i.e. workflow type in our terminology) that is to be modified. The evolution policies define, depending on the actual state and execution history of the instances, what should happen to the instances of the old specification, i.e. which instances have to be migrated to the new specification, which ones have to be aborted or rolled back to a certain state, and so on. In [12], concrete operations by which workflow specifications can be modified are not considered. Furthermore, only a simple versioning model supporting at most two versions of a workflow specification (the old specification and the new modified specification) is used.

The approach for dynamic workflow schema evolution presented in [8] is similar to ours in that it is based on the versioning of workflow specifications. [8] mainly concentrates on versioning issues such as managing consistent configurations of versions and does not define correctness criteria for workflow schemas and instances. The schema can be modified by either changing a (workflow specification) version itself or by deriving a new version from an existing one. The modification of a version includes the migration of the existing instances to the modified version, where several modifications of existing versions (e.g. change a parameter definition or control flow) can only be carried out if no running instances of the version to be modified exist. A workflow can only be migrated from a version x to a version y if a) y is a descendant of x and b) if x and y are equal with respect to their content, i.e. y has not been modified yet. Our approach, on the other hand, supports the migration of workflows to descendant as well as ancestor versions. Furthermore, in our approach, the version to which a workflow is to be migrated to, needs not to be equal (with respect to their content) to the version from which the workflow is migrated from. Thus, the approach presented in [8] is more limited than ours with respect to workflow migration.

Summarizing, the most significant difference between the aforementioned related work (with the exception of EPOS and [8]) and our approach is that ours is based on the explicit support for workflow type versioning which enables workflow evolution in a more systematic and flexible way. Additionally, our approach shows a way to efficiently determine whether migration is possible.

# 6 Conclusion

We have described an approach towards dynamic schema evolution that is based on workflow type versioning and workflow migration. The particular strength of this approach is that it allows a broad range of workflow schema modifications, and to keep track of the resulting history (by maintaining the corresponding WFT version trees). Upon schema modification, workflow instances can be migrated to new versions unless workflow execution invariants are violated through the migration. Those instances that cannot be migrated remain associated with the old version (which represents another advantage of an integrated treatment of versions and modification/migration).

A prototype has been developed that implements the concepts described in this paper. The prototype supports the creation of WFT versions and the execution of workflow instances. The workflow schema can be modified by using operations equivalent to those discussed in section 3, where only those operations can actually be carried out that preserve the correctness of the schema and of the existing workflows. Moreover, a subset of the instances of a certain WFT version can be selected by applying a *selection condition* to the WFT version (see Section 4.3). The selected workflows of a WFT version can be migrated to the desired version, where only those workflows are actually migrated whose migration conditions hold.

Furthermore, we plan to extend our workflow model with concepts to enable the partial rollback of workflows. Then, workflows that could not be migrated otherwise can possibly be rolled back to a state in which migration is possible.

## References


1. S.C. Bandinelli, A. Fuggetta, and C. Ghezzi. Software Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.
2. J.B. Banerjee, W. Kim, H.-J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *Proc. ACM-SIGMOD*, San Francisco, CA, May 1987.
3. P. Bichler, G. Preuner, and M. Schrefl. Workflow Transparency. *Proc. 9th Int'l Conf. on Advanced Informations Systems Engineering*, Barcelona, Spain, June 1997.
4. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data & Knowledge Engineering* 24(3), January 1998.
5. C. Ellis, K. Keddara, and G. Rozenberg. Dynamic Change Within Workflow Systems. *Proc. Conf. on Organizational Computing Systems*, Milpitas, USA, August 1995.
6. D. Georgakopoulos, M. F. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases* 3(2), April 1995.
7. M.L. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12), December 1993.
8. G. Joeris and O. Herzog. Managing Evolving Workflow Specifications. In *3rd IFCIS Conf. on Cooperative Informations Systems (CoopIS '98)*, New York, August 1998.
9. R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys* 22(4), December 1990.
10. M. Kradolfer and A. Geppert. Modeling Concepts for Workflow Specification. Technical Report 97.05, Department of Computer Science, University of Zurich, 1997.
11. F. Leymann and W. Altenhuber. Managing Business Processes as an Information Resource. *IBM Systems Journal* 33(2), 1994.
12. C. Liu, M.E. Orlowska, and H. Li. Automatic Handover in Dynamic Workflow Environments. In *10th Conference on Advanced Information Systems Engineering*, Pisa, Italy, 1998.
13. S. Monk and I. Sommerville. Schema Evolution in OODBs using Class Versioning. *SIGMOD Record, 22(3)*, September 1993.
14. M. Reichert and P. Dadam. ADEPTflex - Supporting Dynamic Changes of Workflows Without Loosing Control. *Journal of Intelligent Information Systems* 10(2), 1998.
15. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.

16. A. Skarra and S. Zdonik. The Management of Changing Types in an Object-Oriented *Database. Proc. OOPSLA 86'*. 1986.

# A Correctness of the Migration Conditions

In the following, the correctness of the migration conditions defined in Section 4.1 is proofed.

Consider one modification per point in time. wt[x] and wt[y] are WFT versions, where wt[y] has been derived from wt[y] by applying a modification operation to wt[x]. Assume wt[x] and wt[y] are correct w.r.t. the schema invariants. Assume further that the workflow instance w has been a correct execution of wt[x] until ct. We prove that w', the migrated version of w, is a valid execution of wt[y]. We prove this by contradiction: w' can only then violate its execution invariants if wt[x] or wt[y] violate schema invariants, migration conditions do not hold, or w violated execution invariants.

**Adding a parameter definition (op. 1.1.1.1).** A parameter definition pd is added to a WFT version wt[x], resulting in a new version wt[y].

In order to violate the execution invariants for this case, there must be a data flow d whose sink or source is pd. Then d has been defined before pd (to which it refers), which is a contradiction to the correctness of wt[x].

**Deleting a parameter definition (op. 1.1.1.2).** A parameter definition pd is removed from a WFT version wt[x], resulting in a new version wt[y].

In order to violate the execution invariants for this case, there must be a data flow d whose sink or source is pd. Then pd has been deleted while d is still in wt[y]. Thus, d refers to an undefined parameter, which is a violation of the schema invariants and, therefore, in contradiction to the correctness of wt[y].

**Adding new a end state (op. 1.1.2.1).** An end state e is added to a WFT version wt[x], resulting in a new version wt[y].

In order to violate the execution invariants for this case, w' must generate an end event for e. Since addition of e is the only modification operation (and end conditions have not been modified), w has also generated an end event for e.

1 Assume e has already been defined for wt[x]. Then adding e to wt[x] (which results in wt[y]) violates the unique naming invariant, i.e., wt[y] violates the schema invariants.
2 Otherwise, if e has not been defined in wt[x], then generation of an end event e in w would have violated the execution invariants for w, because e can only be generated when a corresponding end state was defined by wt[x].

**Deletion of an end state (op. 1.1.2.2).** An end state e is removed from a WFT version wt[x], resulting in a new version wt[y].

In order to violate the execution invariants for this case, w' must generate an end event.

1 Assume first that this end state is not e. At least one of the three constraints for the occurrence of end events must be false to render w' incorrect. Since the deletion of e is the only difference between wt[x] and wt[y], and w' does not generate an end event for e, it follows that w has not been correct, which is a contradiction to our assumptions.
2 Thus, assume that w' generates an end event for e. Then, w' is incorrect because it generates an end event for which no corresponding definition exists. However, generation of an end event for e is in contradiction to the migration condition for deletion of end states.

**Adding a subworkflow declaration (op. 1.2.1.1).** A subworkflow declaration wd is added to a complex WFT version wt[x], resulting in a new version wt[y].

In order for w' to violate the execution invariants, there must be a subworkflow sw of w' declared by wd, and sw must violate the execution invariants in w'. This can happen by violating control flow or data flow constraints. This in turn is however a contradiction to our assumption that wt[x] is preserving the schema invariants, because data/control flow would have been defined before the definition of its source/target (i.e., wd).

**Deletion of a subworkflow declaration (op. 1.2.1.2).** The subworkflow declaration wd is deleted from a complex WFT version wt[x], resulting in a new version wt[y].

1 Let sw be a subworkflow of w' and declared by wd. Assume that sw violates execution invariants in w' (such as data flow constraints). Each such violation requires sw to have started its execution, which is in contradiction to the migration condition.

2 Thus, sw cannot be present in w'. Assume that the deletion of sw leads to violated invariants of w'. w' can however only then become incorrect if its type refers to wd in (one or more) start conditions, end conditions, or data flow definitions. In this case, wt[y] violates schema invariants.

**Modifying a start condition (op. 1.2.1.3).** The start condition sc of a subworkflow declaration sd defined by a complex WFT version wt[x] is modified, resulting in a new version wt[y].

1 Assume workflow sw declared by sd has already been started. Then, in order for w' to violate its execution invariants, the constraints on the occurrence of sw's start event must be violated. The first and the third constraint cannot be false, because this would imply w's incorrect execution. Thus, the second constraint must be violated, i.e., sc' does not hold. Thus, from the migration condition it follows that the start event for sw is not in E, which is a contradiction to the assumption that sw has already been started.

2 Assume sw has not yet been started. Then the incorrectness of the execution of w' does not involve sw, and w itself must have been incorrect (which in turn is a contradiction).

**Adding a data flow definition (op. 1.2.2.1).** A data flow definition df (from parameter p1 to parameter p2) is added to a complex WFT version wt[x], resulting in a new version wt[y].

1 The sink of df (i.e., p2) is an in-parameter of a subworkflow sw. In order to violate the execution invariants, sw must have been started, and p1's value is not equal to p2's value. The first term is in contradiction to the migration condition, which states that in this case sw has not yet started.

2 p2 is an out-parameter of a complex workflow cw. In order to render the execution of w' incorrect, the constraints on the parameter values must be violated for cw (where the latter is in turn w' itself or a subworkflow of w'). Thus, an end event of cw must exist, and p1's and p2's values must be different. Again, the first term is a contradiction to the migration condition, which states that cw has not yet ended.

**Deletion of an existing data flow definition (op. 1.2.2.2).** Analogously to the proof for the migration condition of (op. 1.2.2.1) directly above.

**Modifying an end condition (op. 1.2.3.1).** The end condition ec for the end state es of a CWFT version wt[x] is modified, resulting in a new version wt[y].

In order for w' to become incorrect, it must violate the second constraint on the occurrence of the end event. This means, that the end event for es has occurred, and the end condition ec is false in w'. This is in contradiction to the migration condition, which states that either the event does not exist or it implies the new end condition.