

# Dynamic Evolution in Workflow Management Systems

Paulo Dias  
INESC-ID

Technical University of Lisbon  
Paulo.Dias@inesc-id.pt

Pedro Vieira  
INESC-ID

Technical University of Lisbon  
Pedro.Vieira@inesc-id.pt

António Rito-Silva  
INESC-ID

Technical University of Lisbon  
Rito.Silva@inesc-id.pt

## Abstract

*The environments where workflow management systems are typically used in are constantly changing. It is nearly impossible to foresee, at business process design stage, all the combinations of tasks needed to achieve the process' goals. Workflow management systems capable of supporting dynamic changes on executing instances in a flexible manner are thereby demanded. Former approaches have proposed a solution to this issue based on workflow type versioning and workflow instance migration. However, the operations that handle the modification of workflow types as well as the migration algorithm for workflow instances are tied to a particular representation of workflow types. This means that whenever the representation changes, the migration algorithms and the modification operations have to change accordingly. This is a current problem due to the uprising of different specifications for workflow definition languages, which are far away from stabilizing. In this paper, we address this problem by clearly decoupling the modification operations of workflow types from the internal structures and algorithms responsible for the instances' migration. In this manner we enable the use of different workflow definition languages while keeping the same internal structures and migration algorithms.*

## 1. Introduction

A critical aspect for enterprises to succeed is the way they use technology to support business processes. Workflow management systems as software systems designed to coordinate and automate business processes have been object of interest during the latest years. Typically a workflow system involves two distinct activities: the creation of formal business process descriptions and their execution. The descriptions are usually known as *workflow types* (WFTs) whereas the execution units are known as *workflow instances* (WFIs)

or simply *workflows*. The set of WFTs is called *workflow model*, and it can be manipulated through the elements provided by a *workflow meta-model*. The workflow meta-model also establishes the rules for the creation of WFTs. In short, the meta-model defines how to represent a WFT.

Since business processes are the active parts of the business, it is nearly impossible, at design stage, to provide the WFT that best suits each situation. Therefore, workflow management systems should be able to accommodate evolution scenarios, i.e. it should be possible to modify the workflow model, by changing existing WFTs, even if there are WFIs already running according to the WFT changed. Whenever the workflow model is modified the correctness of both the model and the executing instances must be preserved. The model can become invalid when, e.g. a WFT that is referenced by another is removed. On the other hand, a WFI can become invalid if, e.g. the respective WFT is changed in such a way that it is not in accordance with the execution instance at that moment. The evolution of a WFT is less of problem if it happens while there are no instances executing according to that type. Unfortunately, these situations are rare because workflows are normally of long duration and involve non negligible amounts of work. Thus, dynamic evolution of WFTs, i.e. modifications on executing instances, must be supported.

Former approaches [1, 2] have provided solutions for the evolution of WFTs based on WFT versioning and migration of WFIs. A version [5] represents a state of an evolving entity which can be submitted to a version control process. In [1], versioning has been applied to WFTs. Instead of modifying directly the WFT, the idea is to create new versions of that type. These new versions are derived from former ones by the application of a sequence of well defined operations. Regarding migration, it is the process whereby a WFI, which was executing according to a WFT, continues its execution according to a new WFT. Although these approaches overcome the problem of dynamic evolution, the modification operations for WFTs as well as the migration algorithms considered are tied to a particular

representation of WFTs (a particular meta-model). This means that if the representation changes it will be necessary to re-implement not only the modification operations but also the migration algorithms.

In the latest years a few WFTs meta-models have been proposed by different companies (e.g. WSFL, BPEL, etc.) and it seems that an agreement on a common specification is far from happening. So, we believe that any workflow based software should be able to cope with this multiplicity of languages.

In this paper we provide a solution to the problem of dynamic evolution of WFTs but focusing mainly on the adaptability to different kinds of representations for WFTs. In this way we have developed a component which adds evolution support to a particular workflow management system and which is easily adaptable and reusable regardless of the meta-model used to represent WFTs. The main contributions of this work are the following:

- Definition and implementation of modification operations for WFTs
- Versioning support for WFTs
- Implementation of a migration algorithm completely independent from the representation used to define WFTs.
- Support for different evolution policies if the migration of a WFI is not possible at a given moment.

The remainder of the paper is organized as follows: Section 2 describes briefly the workflow management system which has been the target for our Evolution Component. Section 3 discusses the topics related to evolution of WFTs. Section 4 presents the aspects related to migration of WFIs. Section 5 describes the architecture of the Evolution Component developed. Section 6 visits the most relevant related works, and finally, Section 7 presents the conclusions and future work.

## 2. The micro-workflow architecture

In this section, we introduce an extension to the *Micro-Workflow* architecture proposed by [3] since it is the target architecture for our *Evolution Component*. Whereas the *Micro-Workflow* provides the basic means to define and execute workflows, the *Evolution Component* can be viewed as a plug-in which provides additional functionalities. Fig. 1 shows the main components of the architecture.

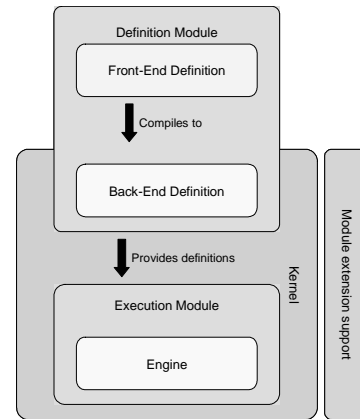


Fig. 1. The *extended micro-workflow architecture*

The core of the *Micro-Workflow* is composed of two modules. These are the *Definition Module* and the *Execution Module*. The former is responsible for managing workflow definitions while the latter is responsible for managing the execution of instances of the stored definitions. The *Module Extension Support* is responsible for linking the different advanced components (e.g. Evolution Component) with the kernel.

### 2.1. Execution module

This module contains the life cycle management operations for the WFIs. It is responsible for reading the shared workflow definitions from the definition module and for executing them according to an execution algorithm.

The execution module defines a set of lifecycle operations that enable the control over the WFIs. These operations are:

- **Execute** – creates an instance of a workflow definition and starts it. As an immediate result of the start operation, a unique instance identifier called `workflowId` is generated and returned to the caller of the operation.
- **Suspend** – temporarily blocks the execution of a given workflow.
- **Resume** – continues the execution of a previously suspended workflow.
- **Query state** – queries the execution state of a WFI given its unique identifier.
- **Terminate** – stops the execution of a workflow. A terminated workflow cannot resume its execution. However, information related to the instance is maintained, so that a client can access it.

## 2.2. Definition module

The *Definition Module* is responsible for managing the workflow definitions. In order to support future evolutions of the framework, the definition module is split in two layers (or sub-modules): the front-end definition and the back-end definition. This reduces the coupling, enabling the support of different client representations of a workflow definition while keeping the same (different from the client's) data structures at the execution engine level. Any workflow model can be used as a client front-end model as long as it meets the following two conditions:

- **CONDITION 1** - there is a mapping from the concepts represented in the front-end model to those provided by the back-end model;
- **CONDITION 2** - the front-end model provides the compilation rules;

### 2.2.1. Back-end definition

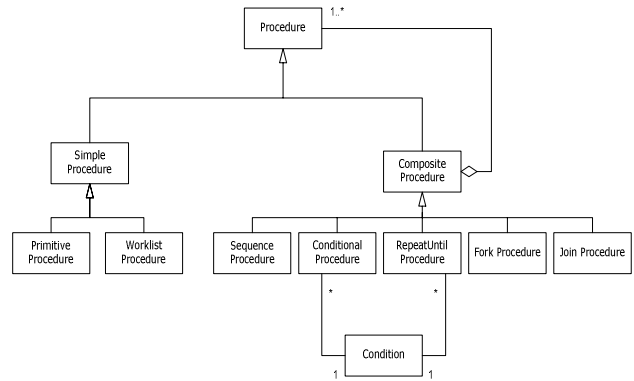
The basic structure for the back-end workflow definition is a directed graph. In the graph, the nodes represent activity steps, and the transitions between nodes represent the flow. The flow can be either control or data. Control flow establishes the order in which the graph should be navigated to execute its nodes. Data flow defines what data is passed from one activity to another. Every graph should have only one *start* node and one *end* node.

### 2.2.2. Front-end definition

The key abstraction of the front-end definition used in this work is the *Procedure*. A tree of hierarchically composed procedures defines a workflow. The various types of procedures presented in this model are based on [3] where the author proposes control structures for procedures, which are similar to those available in structured programming languages. The two main subtypes of *Procedure* are:

- **Simple Procedure** – procedures of this type represent leaf nodes in the tree of procedures and are associated to the work tier, i.e. they provide an abstraction of domain-specific work to be performed outside the framework. A *Simple Procedure* can be either a *Primitive Procedure*, which represents a software service, or a *Worklist Procedure*, which represents a piece of work to be performed by a human worker.
- **Composite Procedure** – the composite procedures are used to represent the structures that manage the flow of control (sequences, conditions, etc).

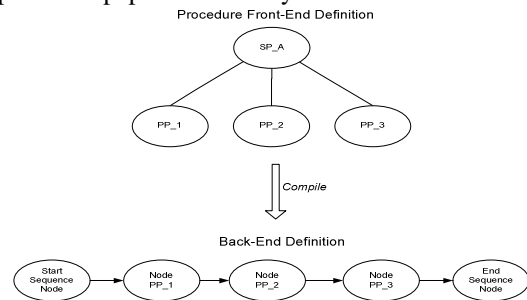
Fig. 2 illustrates the meta-model for the procedure based workflow front-end definition.



**Fig. 2. Meta-model of the procedure based front-end definition**

### 2.2.3. Mapping the procedure based front-end to the back-end

The compilation is performed in a bottom-up fashion starting from the transformation of the leaf procedures to their graph representation, navigating the tree upwards and adding the graphs at each composite procedure until the root is reached. The resulting graph is the graph for the back-end workflow definition, which the execution engine knows how to execute. When defining the compilation algorithm it is important to consider all the existing rules: control flows, data flows and what message types to set to each graph node. Fig. 3 illustrates roughly the mapping of a sequence procedure (front-end) to the internal back-end graph representation. The detailed description of the compilation rules is beyond the scope of this paper and thereby it will not be discussed.



**Fig. 3. Procedure front-end definition / Back-end definition**

## 3. Modification of Workflow Types

A flexible workflow management system should enable the modification of WFTs, even while there are instances executing according to the type that needs to be changed. This section describes how we handle this issue.

First we introduce the concept of WFT version and present an extension to the meta-model of the Fig. 2 with WFT versioning support. Afterwards we introduce the concept of modification operation.

### 3.1. Workflow type versioning

The main idea behind WFT versioning is to create a new version of a WFT rather than to update directly the WFT each time it needs to be changed. In fact, WFTs are just containers for versions. The behavioral workflow information (front-end workflow definition) for a given type is kept within each version of that type. Fig. 4 illustrates the extension to the meta-model presented in Fig. 2, with the support for WFT versioning.

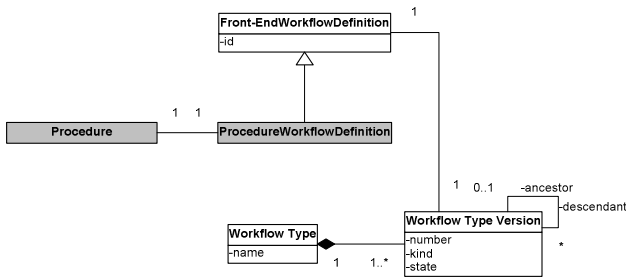


Fig. 4. Meta-model with support for versioning of workflow types

A WFT is composed of one or more WFT versions and a version belongs exclusively to a WFT. Each version can have several descendents and can have but one ancestor. A `Front-EndWorkflowDefinition` is the abstraction used to represent workflows at the front-end level. The front-end definition used in this work is based on procedures (gray shaded classes) and so, a WFT version knows a `ProcedureWorkflowDefinition` which refers to a `Procedure` element. Note that several other front-end definitions might be used to describe workflows without having to modify the versioning structure (white shaded classes). This is accomplished by extending the `Front-EndWorkflowDefinition` class just like we have done for the procedures.

A version is uniquely identified by a *number* and according to the purpose of its creation it is classified either as *revision* or *variant*. A *revision* substitutes an existing version, i.e. it is an update to an outdated version, whereas a *variant* may coexist with other versions in order to respond to a very specific situation.

When a new WFT is added to the model a *root version* is created for that type. Any WFT modification implies, first, the creation of a *descendent* version from a former one and then the application of modification operations to the new created version. Typically a version is created

through an incremental process. The concept of *version's state* [6, 7] reflects the different stages of development associated to a version. A version can lie in three states: *transient*, *published* and *obsolete* (see Fig. 5).

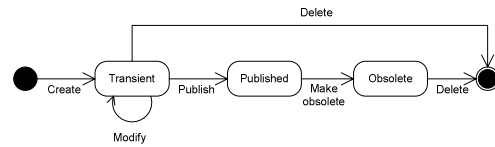


Fig. 5. Workflow type version state diagram

The creation of a new version sets its state to *transient*. In this state it is not possible to create instances for that version neither to derive *descendant* versions. On the other hand a version in this state can be modified or removed. When the modification process of a version in state *transient* is finished, the state is changed to *published*. A version in this state can neither be removed nor modified but, new instances can be created as well as descendant versions. Finally, when a version in the state *published* becomes useless its state is changed to *obsolete*.

### 3.2. Modification operations

To enable the manipulation of the WFT model of Fig. 4 we need a well defined set of operations. By well defined we mean basically two things: completeness and correction [10]. Completeness is the ability to create and remove all the elements that are part of the WFT model. Correction is the ability to ensure that after the application of a sequence of operations, the resulting model and instances are not incorrect. In order to achieve these two requirements, we assign preconditions to the operations. An operation cannot be executed if its precondition is not satisfied. The modification operations are divided in two subclasses:

- CLASS 1 - Operations to create and remove WFT as well as to manipulate versions. These are completely independent from the workflow front-end definition used.
- CLASS 2 - Operations to change the contents of a WFT version. This set of operations needs to be aware of the front-end constructs and thereby it is dependent on the particular front-end used. Consequently, these operations must be redefined whenever the front-end changes. We enumerate two examples of operations that enable the modification of a WFT based on procedure definitions:
  - OP1 $\Leftrightarrow$ addStepAfterSequenceProcedure(`wftID`,`versionNumber`,`sequenceId`,`newStep`)
  - OP2 $\Leftrightarrow$ addBranchInForkProcedure(`wftID`,`versionNumber`,`forkId`,`newBranch`)

The application of OP1 with `sequenceID=SP_A` and `newStep=newPP` to the sequence procedure of Fig. 3 would result in the sequence procedure of Fig. 6. The resulting procedure is a new version of SP\_A, and so, we call it SP\_A[v1].

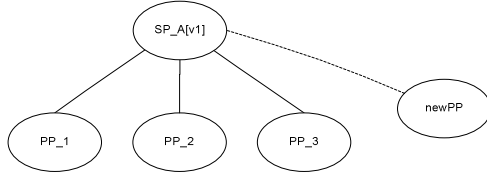


Fig. 6. New version of SP\_A after OP1

## 4. Migration of workflow instances

The migration of a WFI is the process whereby a WFI is associated to a new version different from the current one. When a WFI  $w$ , which is an instance of the version  $wt[x]$ , is migrated to a version  $wt[y]$ , it starts executing according to  $wt[y]$ . It must be ensured that the migration process does not produce invalid WFIs. In particular, the migration of a WFI  $w$  to a version  $wt[y]$  should be allowed only if  $w$  is a valid instance of  $wt[y]$  after the migration.

### 4.1. Migration conditions

A simple way to determine whether a WFI  $w$  can be migrated to a WFT  $wt[y]$  at the moment  $t$  is to analyze the events contained in the history of  $w$  at  $t$  and check the compatibility with  $wt[y]$ . This means that for each event in the history of  $w$  it must be verified if the event can lead to an invalid instance after the migration of  $w$  to  $wt[y]$ . Although this might be a valid algorithm to check whether the migration is possible, it is certainly not the most efficient. If  $wt[x]$  and  $wt[y]$  are not that different the effort of determining whether the migration is possible can be extremely reduced. Therefore, our approach takes into consideration the modification operations that originate the new version (CLASS 2 operations, see Section 3.2). So as to determine whether the migration is possible, migration conditions for each modification operation are considered. A migration condition for an operation  $Op$  specifies whether the workflow can be migrated correctly from  $wt[x]$  to  $wt[y]$  at a given moment  $t$ , where  $wt[y]$  has been derived from  $wt[x]$  by the application of  $Op$ . Consequently, a workflow  $w$  can be migrated from  $wt[x]$  to  $wt[y]$  at a given moment  $t$  if all the migration conditions whereby  $wt[y]$  has been derived from  $wt[x]$  are satisfied by  $w$  at  $t$ .

Although the migration conditions are directly inferred from the modification operations, they operate at a different level of the Evolution Component. Whereas the modification operations of version contents operate at the

front-end level of the component, the migration conditions are based on the back-end level. In fact, this is the main reason for the independence between the migration algorithm and the front-end definition used.

Consider the SP\_A procedure example, and the operation that adds a new step to the end of the sequence (OP1) generating the new version SP\_A[v1]. The migration condition associated to this operation states that, in order to migrate an instance which is executing according to SP\_A to an instance that will execute according to SP\_A[v1], the execution point of the instance should not have reached the EndSequence node (gray shaded in Fig. 7). In other words, the migration is possible if the WFI is executing in one of the white nodes of the graph.

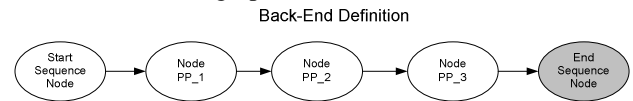


Fig. 7 Migration conditions

The interesting aspect about this approach is that although OP1 depends on the front-end (procedure level), the associated migration condition is set in terms of the back-definition (graph level). In this manner, the migration algorithm, which is responsible for checking whether the migration is possible and migrating the instance, manipulates only the nodes of the back-end graph. This makes the algorithm completely independent from the front-end definition used. First it checks whether the migration is possible by comparing the already executed nodes to the nodes set by the migration conditions, and then, in case of all the conditions are satisfied, the migration can be performed.

Hence, if we use another front-end definition (BPEL, WSFL, etc) we need obviously to re-implement the modification operations that change the version's contents but, the migration conditions are still set in terms of the back-end definition, which means the migration algorithm remains intact.

### 4.2. Evolution policies

As was stated in the previous section, the migration of a WFI depends on the evaluation of a set of migration conditions. Sometimes, the migration of a WFI may not be possible. Consider the example of the previous section. If the node EndSequence has already started its execution, the migration to SP\_A[v1] is not possible. So as to provide a proper response to such situations we have defined three kinds of actions to be taken:

- Abort – aborts the execution of a given WFI.
- Complete – terminates the workflow execution according to the current WFT definition.
- Rollback – rolls back a given WFI to a point where the migration is already possible.

The first two solutions are very simple to handle since they consist of direct actions on WFIs. However, both have disadvantages. While the former leads to the loss of large amounts of work, the later may not be acceptable. For some reason, it may be even impossible to continue executing according to the old definition.

The introduction of the rollback policy tries to overcome the problems arisen from the first two approaches. This solution consists of a step by step action supported by the execution history of the WFI. The rollback action consists in analyzing the execution history of a given instance and executing the respective undo operation for each activity. During the process, the execution history is updated, i.e. the state change events are successively removed. This process terminates when all the migration conditions are satisfied.

As it happens with the migration algorithm, the rollback algorithm also operates on the workflow back-end definition, and thus it may be reused regardless of the front-end used.

## 5. Architecture of the evolution component

In this section we describe the architecture of the evolution component that has been developed to supply the Micro-Workflow management system with evolution capability. We stress the importance of some options taken while designing and implementing this component, namely the ones concerning the support for different front-end representations of WFTs. We show the impacts of changing the front-end definition in the component's architecture.

### 5.1. Conceptual architecture

The Evolution Component is split in three modules. The types and versions manager is responsible for managing the structure of types and versions. CLASS 1 operations described in Section 3.2 are implemented within this module. The migration manager encapsulates the migration algorithm and the contents manager handles the modification operations applied to the contents of a version. CLASS 2 operations are implemented within this module. The types and versions manager interacts with the migration manager when a migration of a WFI is requested. Fig. 8 shows the existing interactions among internal modules and between the modules and external entities. By encapsulating different functionalities in different modules we are promoting the adaptability and reuse of the component. For instance, a modification in the front-end representation used to describe WFTs, e.g. from procedures to BPEL, would have impact only in the containers manager. The other modules would remain intact.

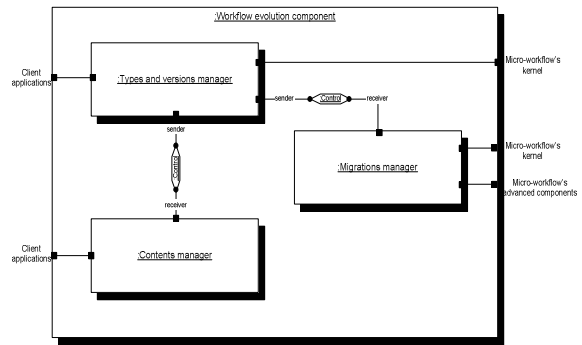


Fig. 8. Conceptual view of the evolution component

### 5.2. Implementation architecture

The Evolution Component has been developed in Java. Fig. 9 depicts the UML class diagram.

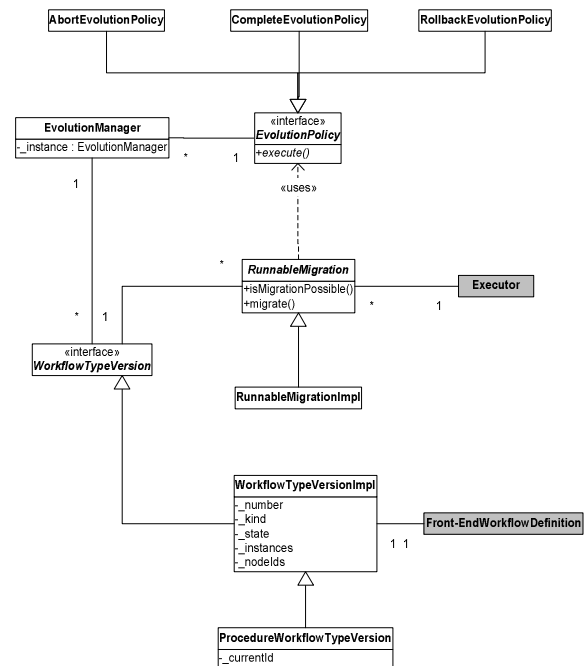


Fig. 9. Evolution component's UML class diagram

The EvolutionManager class implements the operations responsible for managing the structure of WFTs and versions. This class implements the methods for creating, adding and removing WFTs and versions (CLASS 1 operations, see Section 3.2). Moreover, this class interacts with the workflow kernel in order to update the workflow definitions in case of WFT modification. The WorkflowTypeVersion interface is the abstraction for the concept of WFT version.

`WorkflowTypeVersionImpl` implements the `WorkflowTypeVersion` interface. This class should be extended so that different front-end definitions may be used. `ProcedureWorkflowTypeVersion` is the extension provided to deal with procedure definitions. This class implements the operations in CLASS 2, which depend directly on the front-end used.

The `RunnableMigration` abstract class defines the migration algorithm for a given instance of WFT version. The `RunnableMigrationImpl` class extends the behavior of `RunnableMigration` implementing the `isMigrationPossible()` and `migrate()` methods. The former method checks whether the migration is possible given the instance execution history. This is done through the analysis of the migration conditions (see Section 4.1) against the current execution state which is retrieved from the history component of the Micro-Workflow. The later implements the migration, by updating the definition of the instance within the kernel.

The `EvolutionPolicy` interface is the abstraction used to represent the strategy to be applied if the migration is not possible. The `AbortEvolutionPolicy`, `CompleteEvolutionPolicy` and `RollbackEvolutionPolicy` classes implement the `EvolutionPolicy` interface defining the method `execute()` according to the action to be carried out.

Table 1 shows the relationships between the modules described in Section 5.1 and the classes presented in Fig. 9. In order to support different front-end definitions we simply need to change the contents manager, which corresponds to create a particular new class including the modification operations for the elements belonging to the new front-end. The other modules may be reused *as-is*.

**Table 1. Modules and classes relationships**

Modules	Class(es)
Workflow types and versions manager	<code>EvolutionManager</code> <code>WorkflowTypeVersion</code> , <code>WorkflowTypeVersionImpl</code>
Contents manager	<code>ProcedureWorkflowTypeVersion</code>
Migration manager	<code>RunnableMigration</code> , <code>RunnableMigrationImpl</code> , <code>EvolutionPolicy</code> , <code>AbortEvolutionPolicy</code> , <code>CompleteEvolutionPolicy</code> , <code>RollbackEvolutionPolicy</code>

## 6. Related work

The major contributions, regarding evolution of workflow types, come mainly from [1] and [2]. On the

versioning topic, the approach presented in this paper is conceptually very similar to [1]. The major advantage of our approach is that we support versioning of WFTs regardless of the workflow definition language used.

On migration of WFIs, the approach presented in [2] differs greatly from ours. First, a version modification includes the migration of the existing instances of the modified version. Second, for some modifications it is not possible to migrate if there are instances executing according to the modified version. Finally, a WFI can only be migrated from a version  $x$  to a version  $y$  if: (a)  $y$  is a descendent of  $x$  and (b) if  $x$  and  $y$  are equal in terms of content, i.e. if  $y$  has not yet been modified. In our approach, the target version does not need to be equal to the source version in order to perform a migration.

Still on this topic, our work is similar to [1] in that we use the same algorithm to migrate WFIs, i.e. we associate migration conditions to the modification operations, and then, based on these conditions, the migration is performed. However, in [1], the migration conditions are set in terms of execution invariants. These invariants depend on the particular meta-model used to represent WFTs. The result is a migration algorithm dependent on the WFT representation. In our approach modification operations are decoupled from the migration conditions because they are specified at different levels of the component. The modification operations operate at the front-end level and the migration conditions are defined at the back-end level. This enables the support of different front-ends while keeping the same migration algorithm. The changes to perform are restricted to the modules that depend on the meta-model used.

## 7. Conclusions and future work

This paper covers the issue of WFTs evolution based on WFT versioning and WFI migration. Although there are a few works on this area, they do not address the problem of reusability and adaptability. The constant changes on the specifications of workflow languages have driven our thoughts towards the development of a component, which besides adding evolution support to a workflow management system can be easily adapted to different representations of WFTs.

Future work will comprise two different issues. On the one hand, the Evolution Component will be extended to support a few additional features, namely complex WFT versioning, i.e. the ability to work with sub-workflow versions rather than versioning a workflow definition as a whole. This work will cover the topic of version's proliferation. On the other hand, we will apply the Evolution Component to a very specific domain, which is the Support for Distributed Software Development following an XP approach [9]. We believe that a flexible workflow management system could be very useful in the

coordination of XP development teams. As XP methodologies are mainly based on Agile Processes [8] the workflow system used should support adaptive behaviour. In this manner, we feel that combining the Evolution Component features with the extended Micro-Workflow (see Section 2) can result in a powerful tool to aid XP project coordination.

## References

- [1] M. Kradolfer and A. Geppert. Dynamic Workflow Schema Evolution Based on Workflow Type Versioning and Workflow Migration. *Fourth IECIS International Conference on Cooperative Information Systems*, September 1999, Edinburgh.
- [2] G. Joeris and O. Herzog. Managing Evolving Workflow Specifications. *Third International Conference of Cooperative Information Systems*, August 1998, New York.
- [3] D. Manolescu. Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development. *Phd Thesis*, 2002.
- [4] W.M.P van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, January 2003.
- [5] R. Conradi and B. Westfechtel. Version Models for Software Configuration. *ACM Computing Surveys*, June 1998.
- [6] H. T. Chou and W. Kim. A Unifying Framework for Versions in a CAD Environment. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB '86)*, Kyoto, Japan, August 1986.
- [7] M.E.S. Loomis. Object Versioning. *Journal of Object-Oriented Programming*, January 1992.
- [8] Agile Processes, <http://www.agilealliance.com>
- [9] Xprogramming, <http://www.xprogramming.com>
- [10] J. B. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM SIGMOD*, San Francisco, USA, May 1987.