Transforming Workflow Graphs

Johann Eder, Wolfgang Gruber, and Horst Pichler

University of Klagenfurt
Department of Informatics-Systems
{eder,gruber,pichler}@isys.uni-klu.ac.at

Abstract. Workflow management systems are very useful for integrating separately developed application systems by controlling flows of execution. For various purposes (e.g. distribution of activities, workflow evolution, time calculation, etc.) it is necessary to change the representation of a workflow, the structure of a workflow graph without changing it's semantics. We provide an equivalence definition of workflow graphs and introduce a set of basic transformation operations defined on workflow graphs which keep the semantics. We show how these basic operations can be combined to achieve complex transformations and briefly describe a prototypical transformation tool.

1 Introduction

Workflow management systems (WFMSs) improve business processes by automating tasks, getting the right information to the right place for a specific job function, and integrating information in the enterprise [8, 12, 10, 1]. Workflow management systems were also considered as integration tools from the very beginning, with the idea that they allow the representation of processes comprising of activities which are executed within different application systems.

Numerous workflow models have been developed, based on different modelling concepts (e.g. Petri Net variants, precedence graph models, precedence graphs with control nodes, state charts, control structure based models) and on different representation models (programming language style text based models, simple graphical flow models, structured graphs, etc.). Transformations between representations can be difficult (e.g. the graphical design tools for the control structure oriented workflow definition language WDL of the workflow system Panta Rhei had to be based on graph grammars to ensure expressiveness equality between text based and graphical notation [5]).

Another sort of transformation is to change the structure of a workflow within its representation model. Such transformations are needed in several situations. During workflow design, it allows the designer to view a workflow graph from different perspectives and change the representation to enhance readability and understandability. For distributing the activities of a workflow to separate information systems, or among different stakeholders in a virtual organization, it is useful to change the workflow graph to make this distribution explicit and easy to understand and to support the dispatcher at runtime. For workflow evolution

[7] two workflow graphs (initial and final workflow) have to be compared and a manifold of hybrid workflows have to be generated. This task is much easier, if both workflow graphs are previously prepared in a way that the comparison is facilitated by structural equivalence to the greatest possible extent, such that the comparison of the workflow only has to deal with the actual changes of the workflow which lead to different executions and not with mere representational alterations. Furthermore, such transformations are needed for time management[4], or for organizational changes [2, 16].

The main contributions of this paper are: We present a notion for equivalence of workflow models based on the idea that two workflows are equal, if they admit the same workflow instances at the level of atomic activities. We introduce a series of basic transformations which preserve this semantics of the workflows. We show how complex transformations can be constructed from the basic operations and briefly describe a workflow model management tool prototypically implementing these transformations.

2 Workflow Model

2.1 Structured Workflow Definition

A workflow is a collection of *activities, agents,* and *dependencies* between activities. Activities correspond to individual steps in a business process, agents (software systems or humans) are responsible for the enactment of activities, and dependencies determine the execution sequence of activities. We assume that workflows are *well structured*: A well-structured workflow consists of m sequential activities, $T_1 \ldots T_m$. Each activity T_i is either elementary, i.e. it cannot be decomposed any further, or it is complex. A complex activity consists of n_i parallel (and), sequential (seq), conditional (or) or alternative (alt) sub-activities ${T_i}^1, \ldots, {T_i}^{n_i}$, each of which is either elementary or complex (cf. [3]). (Complex) activities between *seq-split* and *seq-join* are always executed in sequence. An *and-split* node refers to a control element with several immediate successors, all of which are executed in parallel. An *and-join* node synchronizes the workflow as it can only be executed if all of its immediate predecessors have finished their execution. An *or-split* node refers to a control element whose immediate successor is determined by evaluating some boolean expression (conditional) and the successor node of and *alt-split*s is selected by (user-)choice. Note that our semantics of or-splits and alt-splits demands that exactly one of many successors may be executed. During *Or-join*s and *alt-joins* refer to control elements that join all the branches after or-splits and alt-splits respectively. Additionally, predicates have to be defined for each successor node of an or-split, representing a boolean expression which must yield true as precondition for execution (due to the exclusive semantics of an or-split only one of many successors may be executed, thus the predicates must be defined accordingly). Structured Workflows are often represented by *structured workflow graphs*, where nodes represent activities (rectangles) or control elements (circles) and edges correspond to dependencies between nodes (see Fig. 1). Predicates are displayed in angle brackets, attached on top of a node.

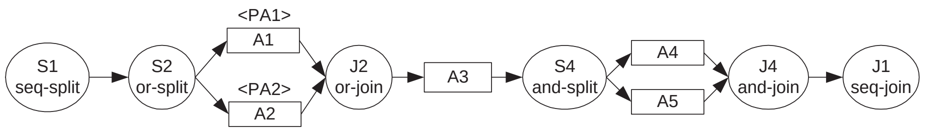


Fig. 1. Workflow graph example

According to the definition presented above a workflow graph must be *well structured*, as each split node is associated with exactly one join node and vice versa and each path in the workflow graph originating in a split node leads to its corresponding join node. For the purpose of allowing more transformations (see section 3) and the separation of workflow instance types in the workflow model we offer the notion of a *structured* workflow graph. Here a split node may be associated with several join nodes, however, a join node corresponds to exactly one split node. Each path originating in a split node has to lead to an associated join node. Such graphs are results of equivalence transformations.

2.2 Workflow Instance Types

Due to conditionals not all instances of a workflow will process the same activities. We classify workflow instances into workflow instance types according to the actual executed activities. Similar to [14], a *workflow instance type* refers to (a set of) workflow instances that contain exactly the same activities, i.e., for each or-split node in the workflow graph, the same successor node is chosen; resp. for each conditional complex activity the same child-activity is selected. Therefore, a workflow instance type is a submodel of a workflow where each or-split has exactly one successor; resp. each conditional or alternative complex activity has exactly one subactivity [3].

2.3 Equivalence of Workflows

Our goal is to support the transformation of a workflow without changing the semantics. For this purpose we need a clear definition when workflows are equivalent. Our definition is based on the consideration that workflows are equivalent if they provide the same tasks. Therefore, the equivalence of correct workflows bases on equivalent tasks and identical execution order. Workflows are equivalent, if they execute the same activities in exactly the same order. Therefore, the equivalence of structured workflows ($WF1 \equiv WF2$) is based on equivalent sets of workflow instance types [9].

Equivalence of workflows Two workflows are equivalent ($WF1 \equiv WF2$) if their sets of instance types are equivalent. Two instance type sets are equivalent if and only if for each element of one set there is an equivalent element in the other set.

Equivalence of instance types Two workflow instance types $IT1$ and $IT2$ are equivalent ($IT1 \equiv IT2$) if they consist of the same (elementary) activities with identical execution order, where the position of or-splits and or-joins in instance types is irrelevant, since an or-split has only one successor in an instance type.

3 Workflow Transformations

Workflow transformations are operations on a workflow SWF resulting in a different workflow *SWF'* (e.g. moving splits or joins) [9]. It is essential that such changes are introduced systematically and that their impact is clearly understood. Workflow model transformation is a suitable approach for this purpose [15]. The application of pre-defined transformation operations can ensure that the modified process conforms to constraints specified in the original model. In the following we provide a set of transformations, which do not change the semantics of the workflow according to the definition of equivalence given above. Complex transformations can be established on this basic set of transformations by repeated application. Transformations are feasible in both directions, i.e. from SWF to *SWF'* and vice versa from *SWF'* to *SWF*. We differ between *basic* and *complex* workflow transformations, where complex transformations are composed of consecutively applied basic transformations. Each transformation of a structured workflow graph must result in another structured workflow graph.

3.1 Basic Workflow Transformations

a) Sequence Encapsulation, Coalescing Nodes and Path Separation

• **Encapsulation in a Sequence (WFT-S)** An activity can always be encapsulated between two sequence control elements (seq-split and seq-join).
• **Or-Join Coalescing (WFT-JC1)** In a workflow SWF with a nested or-structure two succeeding or-joins and their according or-splits can be coalesced into a single or-structure. It is necessary to adjust the predicates according to the changed sequence of split-nodes $S1$ and *S2* by applying the conjunction (*, logical and). An example for this transformation is given in Fig. 2. This transformation is similar to the structurally equivalent transformation presented in [15] as far as the differences in the workflow models are concerned.
• **Alt-Join Coalescing (WFT-JC2)** This transformation is performed analogously to *WFT-JC1*, by replacing each *or*-split with an *alt*-split and each *or*-join with an *alt*-join and vice versa. Note that there are no preconditions to consider in such a scenario.

• **Separating a Conditional or Alternative Path (WFT-PS)** In a workflow *SWF* with an or-structure or alt-structure each path can be separated by means of duplicating join-node $J1$, if (and only if) $J1$ has no successor (see Fig. 2).

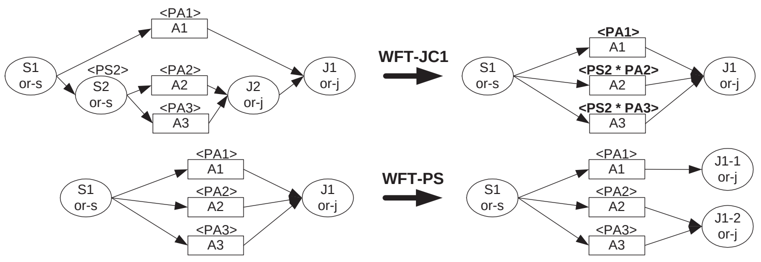


Fig. 2. Coalescing nodes and path separation

b) Moving Joins

Moving Joins means changing the topological position of a join control element (and-join, or-join, alt-join, or seq-join). This transformation separates the intrinsic instance types contained in a workflow model. Some of the following transformations require node duplication. In some cases moving a join element makes it necessary to move the corresponding split element as well.

• **Moving Join over Activity (WFT-J1)** A workflow *SWF* with an or-join or an alt-join $J1$ followed by an activity *A3* can be transformed to a workflow *SWF'* applying node duplication, so that the join $J1$ is shifted behind the duplicates *A3-1* and *A3-2* of activity *A3* as shown in Fig. 3. This transformation, and all of the following ones, can be applied to structures with any number of paths.

• **Moving Join over Seq-Join (WFT-J2)** A workflow *SWF* with an or-join or an alt-join *J1* followed by a sequence join *J2* can be transformed to workflow *SWF'* applying node duplication, so that the join $J1$ is delayed after *J2* as shown in Fig. 3. Here, *J2* will be replaced by its duplicates *J2-1* and *J2-2*, such that *J1* is the successor of *J2-1* and *J2-2*, and *A1* is the predecessor of *J2-1* and *A2* is the predecessor of *J2-2*.

• **Moving Or-Join over Or-Join (WFT-J3)** In a workflow *SWF* with a nested or-structure (i.e. within an or-structure with the split *S1* and the corresponding join *J1* there is another or-structure with the split S2 and the corresponding join *J2*), the inner join *J2* can be moved behind the outer join *J1*, which makes it necessary to move the corresponding split element *S2* and to adjust the predicates according to the changed sequence of $S1$ and *S2* by conjunction (*, logical and) or disjunction (+, logical or). This change causes the inner or-structure to be put over the outer (see Fig. 3).

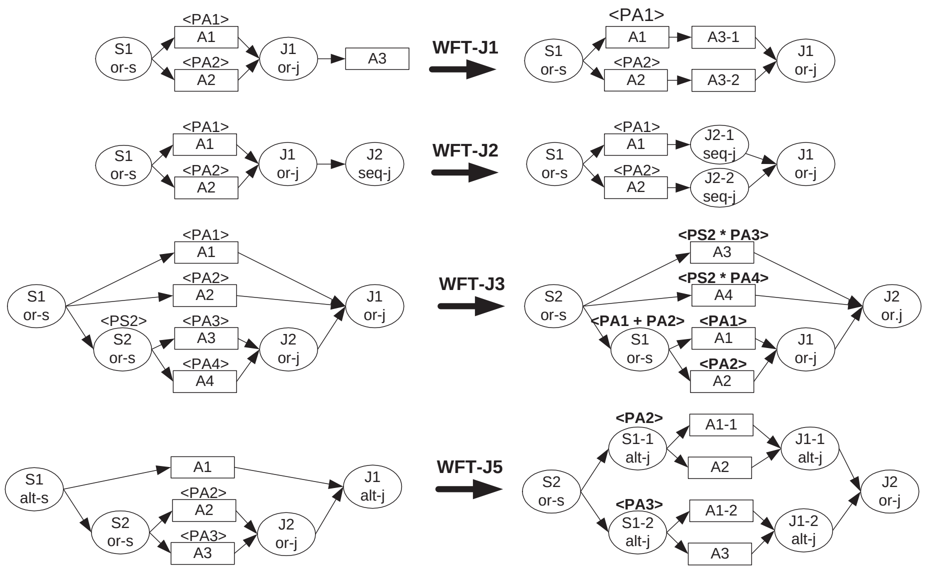


Fig. 3. Moving Joins (1)

• **Moving Alt-Join over Alt-Join (WFT-J4)** This transformation is performed analogously to *WFT-J3*, by replacing each *or*-split with an *alt*-split and each *or*-join with an *alt*-join. Note that there are no preconditions to consider in such a scenario.

• **Moving Or-Join over Alt-Join (WFT-J5)** In a workflow *SWF* with a nested alt/or-structure, i.e. within an alt-structure with the split *S1* and the join *J1* there is an or-structure with the split *S2* and the join *J2*, the inner join *J2* can be moved behind the outer join *J1*. This makes it necessary to move the corresponding split element *S2* and to duplicate control elements and activities and adjust the predicates. In fact, the inner or-structure is put over the outer structure (see Fig. 3).

• **Moving Alt-Join over Or-Join (WFT-J6)** This transformation is performed analogously to *WFT-J5*, by replacing each *or*-split with an *alt*-split and each *or*-join with an *alt*-join and vice versa.

• **Unfold: Moving Join over And-Join (WFT-J8)** The *unfold* transformation produces a *structured* graph-based structure with *multiple sequential successors*, which means that a node, with the exception of splits, can have more than one sequential successor. However, in each instance type of such a graph every node except for and-splits has again exactly one successor. An or-join or alt-join $J2$ can be moved behind its immediately succeeding and-join *J1*, requiring duplication of control elements. An example for this transformation is shown in Fig. 4. To move *J2* behind *J1* we place a copy of *J1* behind every predecessor of *J2*, so that each of these copies of *J1* has additionally the same predecessor as *J1* except for *J2*. A copy of *J2* is inserted, such that it has the copies of *J1* as predecessor and the successor of *J1* as successor. Then *J1* is deleted with all its successor and predecessor dependencies. If *J2* has no longer a successor, it will also be deleted. *Partial unfold*, as it is described in [9], is a combination of already described transformations.

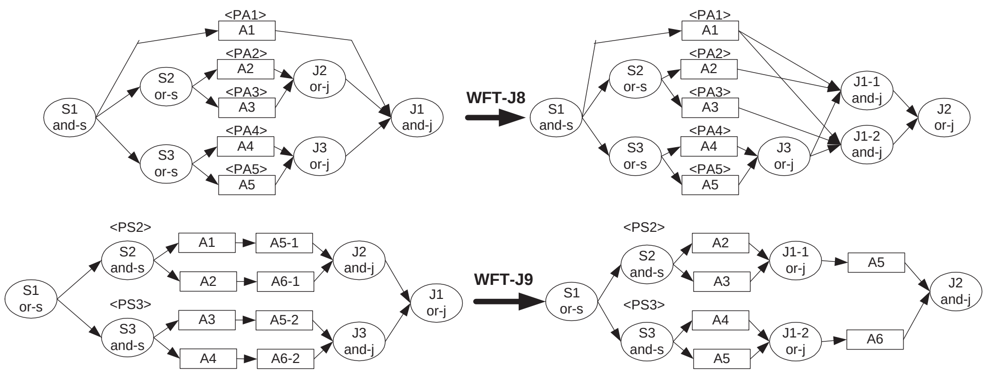


Fig. 4. Moving Joins (2)

• **Moving And-Join over Or-Join or Alt-Join (WFT-J9)** This transformation is introduced in [11]. Starting with a workflow *SWF* with an or-join $J1$, which has only and-joins $J2_1 \dots J2_m$ as predecessors, each of these and-joins $J2_i \in \{J2_1 \dots J2_m\}$ has the **identical set of predecessors** $M_1 \dots M_n$. Let the sets of the predecessors of $M_1 \dots M_n$ for every and-join be $S_1 \dots S_m$. The or-join $J1$ can be moved before the predecessors of the and-joins, which necessitates the duplicating and coalescing of control elements. The transformation is shown in Figure 4. In order to move $J1$ we place a copy of $J1$ for every predecessor of an and-joins $J2_i \in \{J2_1 \dots J2_m\}$, so that each copy of $J1$ has the same number of predecessors as $J1$ and every copy of $J1$ has as predecessor one element from every set $S_1 \dots S_m$, so that every element from $S_i \in \{S_1 \dots S_m\}$ has only one successor. Furthermore, we place a copy of an and-join $J2_i$ with its predecessors, so that every copied predecessor of the copied and-join $J2_i$ has exactly one copy of $J1$ as its predecessor. The copy of the and-join $J2$ has as successor the successor of $J1$, if existent. Now, the and-joins $J2_1 \dots J2_m$ with their predecessors and with all their successor and predecessor dependencies are deleted.

c) Moving Splits

Moving Splits changes the position of a split control element. This transformation separates (moving splits towards start) or merges (moving splits towards end) the intrinsic instance types contained in a workflow model, in analogy to join moving. Not every split can be moved: Moving alt-splits is always possible and for or-splits the predicates have to be considered. Another aspect of or-split moving to be considered is that the decision which path of an or-split is selected will be transferred towards the workflows start, so that decision uncertainties due to or-splits will be reduced.

• **Moving Split before Activity (WFT-S1)** A workflow *SWF* with an or-split or alt-split *S1* with activity *A1* as predecessor can be transformed in the workflow SWF' through node duplication, so that *S1* is located before *A1* (see Fig. 5). Here, *A1* will be replaced by its duplicates *A-1* and *A-2*, so that *S1* is the predecessor of *A-1* and *A-2*, and *A2* is the successor of *A1-1* and *A3* is the successor of *M1-2*. Predicates are adjusted.

• **Split Moving Over Seq-Join (WFT-S2)** A workflow *SWF* with an or-split or alt-split *S*2 proceeded by a sequence split *S1* can be transformed to a workflow *SWF'* through node duplication, so that the split *S*1 is delayed after *S2* as shown in Fig. 5. Here, *S1* will be replaced by its duplicates *S1-1* and *S1-2*, so that *S2* is the predecessor of *S1-1* and *S1-2*, and *A1* is the successor of *S1-1* and *A2* is the successor of *S1-2*. This transformation results in a *structured* workflow (see Fig. 5 for an example).

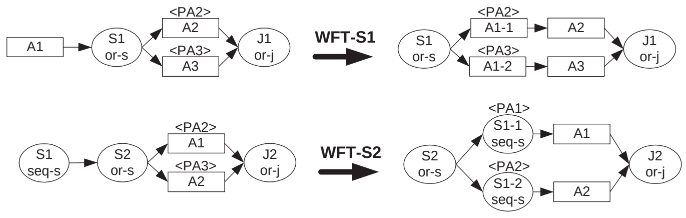


Fig. 5. Split Moving Over Seq-Split

3.2 Complex Transformations

The basic workflow transformations described above are used to define more complex transformations as compositions with facultatively repeated application of these basic transformations within their composition. These complex transformations do not change the semantics of the workflow either. Complex transformations are established to e.g. address the time constraint incorporation problems as stated in [4]. In that paper, three complex transformations are constructed: (1) the *backward unfolding procedure*, (2) the *partial backward unfolding procedure*, and (3) the *forward unfolding procedure.* Unfolding means that or-joins or alt-joins are moved topologically to the rear and or-splits or alt-splits are moved as near to the start as possible, for a simple separation of different instance types.

a) Backward Unfolding
A procedure for generating an equivalent backward unfolded workflow UW for a workflow W is described in [9]. The transformation specifies how a workflow has to be modified to become fully unfolded. An alternative approach to unfold a workflow is to apply the above listed basic transformations in a way that no or-join or alt-join element has an activity as successor. Every structured workflow can be fully unfolded, because for every constellation there is a basic transformation that can be applied in order to move the corresponding join topologically backwards. The following constellations have been identified:

– or-/alt-join before activity → use WFT-J1
– or-/alt-join before and-join → use WFT-J8
– or-/alt-join before seq-join → use WFT-J2
– or-/alt-join before or/alt-join → use WFT-J3, WFT-J4, WFT-J5, or WFT-J6
– separate path → use WFT-PS

The above procedure suffers from the potential explosion of the number of "duplicate" nodes in the unfolded workflow, since it considers each instance type separately. This is not always desirable when discriminating between instance types. To avoid this problem, we developed the *partial unfolding* technique.

b) Partial Backward Unfolding
We can unfold the workflow only where it is desired. The procedure of partially unfolding a workflow W to a partially unfolded workflow PUW begins by selecting a *hot-node*, with the side effect that all instance types going through the hot-node are factored out, or intuitively, the workflow graph reachable from the hot-node is duplicated. In principle, every node can be a hot-node. For practical reasons, we assume that a hot-node is an immediate predecessor of an or-join. Once a hot-node is identified, partial unfolding takes place as follows:

1. Mark all (transitive) successors of the hot-node;
2. Apply the transformations *WFT-J1, WFT-J2, WFT-J3, WFT-J4, WFT-J5, WFT-J6, WFT-J8, WFT-PS* on the marked workflow elements so that no or-join or alt-join element has an activity element as successor in the context of the marked workflow elements;

Note that the transformation step order is of great importance to avoid unnecessary cancellation of operations (for details see [9]).

c) Forward Unfold Procedure
A procedure for generating an equivalent forward unfolded workflow UW for a workflow W is described below. In order to unfold a workflow forward, the transformations listed above must be applied, such that no or-split and no alt-split element has an activity element as predecessor. Because of data dependencies not every structured workflow can be fully forward unfolded. For the following constellations there is a basic transformation that can be applied in order to move the corresponding split topologically forward. The following constellations have been identified:

– or-/alt-split after activity without data dependencies → use WFT-S1
– or-/alt-split after seq-split → use WFT-S2
– or-/alt-split after or/alt-join → use WFT-J3, WFT-J4, WFT-J5, or WFT-J6

For the constellations *or-/alt-split after and-join* no transformation exists.

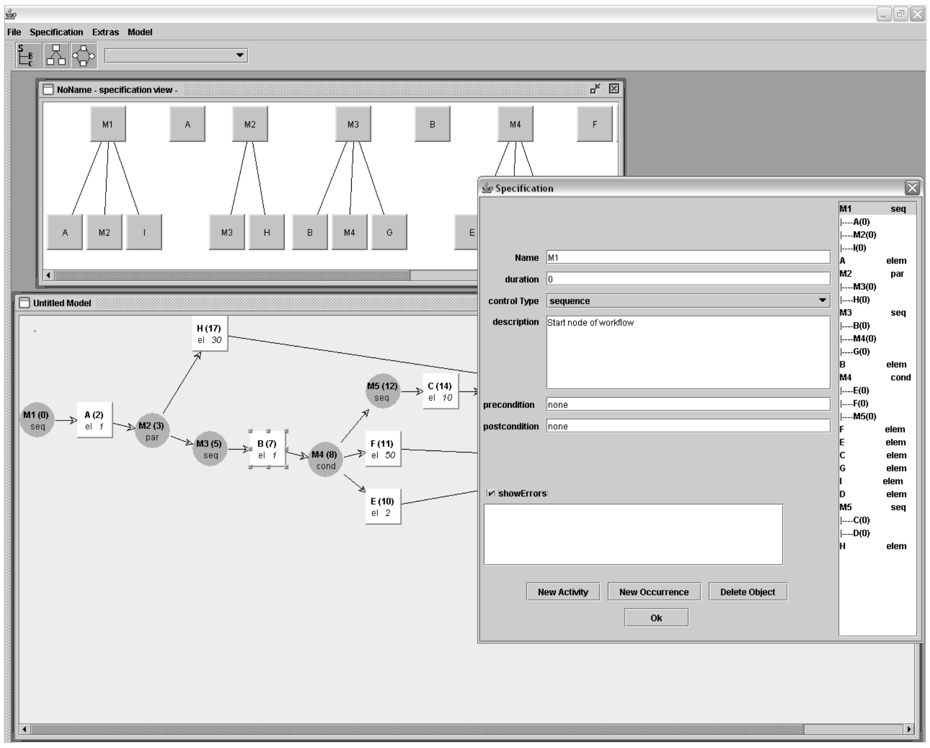
4 Graphical Workflow Designer

Our Graphical Workflow Designer (GWfD) has been developed as proof-of-concept prototype, which currently supports workflow modeling, workflow transformations, modeling of time and verification of time constraints. The requirements for the GWfD architecture were primarily platform independency, persistent data management, modularity, extensibility and simplicity. To assure platform independency the GWfD has been implemented in Java.

To achieve a modular and extensible structure we designed a three layered architecture, where each layer provides services used by the layer below. The **Data Source Connectivity Layer** represents the API to the GWfD data source. Either relational databases (accssed via JDBC) or XML files (accessed via JAXP) serve as data source, where workflows are durably stored, using the relational model from our workflow metamodel [3]. The **Application Layer** implements the application logic and functionalities, which are: create, modify, and delete workflow specifications, deduce workflow models from the specification, and perform transformations and time calculations on the workflow model. The **Presentation Layer** is responsible for the intuitive graphical visualization of workflow specifications and models. For the implementation we applied the recommended architectural design pattern *Model-View-Controller* (MVC), using the freely available Swing Component *JGraph* for the representation and modification of graph-based workflow models.

Figure 6 shows the three main views of the editor: In the **Edit View** (on the right hand side) a workflow can be created or modified. Here the basic building blocks of a workflow are specified, which are elementary activities (marked with the key word 'elem') and complex activities (marked as 'seq', 'cond', 'par' and 'alt'). The workflow structure is defined using a bottom-up approach by assigning (complex) activities to superior complex activities. If the workflow definition is complete a correctness-verification of the workflow specifications can be launched, which detects and displays modelling errors (in order to achieve a well structured workflow graph). The graphical visualization of the specification is presented in the **Specification View**, where the hierarchy of elements, as defined in the Edit View can be examined. Finally, the **Model View** reflects an initially generated graph-based *master workflow model*, which allows no transformations.

From the master model any number of *Child-Models* can be derived, each displayed in its own *Child-Model View*, where workflow transformations can be applied. Figure 7 and figure 8 show an example workflow graph before and after the unfold operation, which is applied on the conditional-join M4.



Name: M1
duration: 0
control Type: sequence
description: Start node of workflow
precondition: none
postcondition: none
[✓ showErrors]

```
M1          seq
|----A(0)
|----M2(0)
|----I(0)
A           elem
M2          par
|----M3(0)
|----H(0)
M3          seq
|----B(0)
|----M4(0)
|----G(0)
B           elem
M4          cond
|----E(0)
|----F(0)
|----M5(0)
F           elem
E           elem
C           elem
G           elem
I           elem
D           elem
M5          seq
|----C(0)
|----D(0)
H           elem
```

Fig. 6. Specification Editor, Specification View and Master Model View

In order to separate the intrinsic instance types in a workflow model, we apply the partial unfold transformation operation. Therefore, we have to specify the instance type to be separated. As illustrated in Figure 7, we select an adjacent predecessor of an *or*-join and invoke the context menu by right-clicking the mouse. When selecting the item **transform** in the context menu, a list of all possible transformation operations in this context appears, which is in this case only (partial) **unfold**. When selecting this operation the workflow model is accordingly modified and the graphical representation is updated, as we can see in Figure 8. The other transformation-operations are implemented in a similar manner, where depending on which workflow elements are selected, the GWfD proposes all possible transformation operations.

5 Conclusions

It is generally a good engineering principle to consider manipulations of design artifacts, study their properties and make applications of such manipulations instrumental in design processes. We introduce an equivalence definition on workflow models based on the notion of instance types and thus capture the semantics

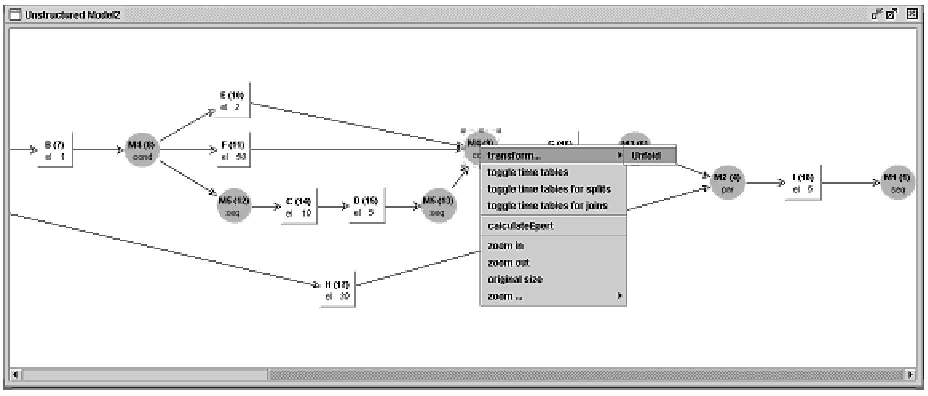


Fig. 7. Before Unfold

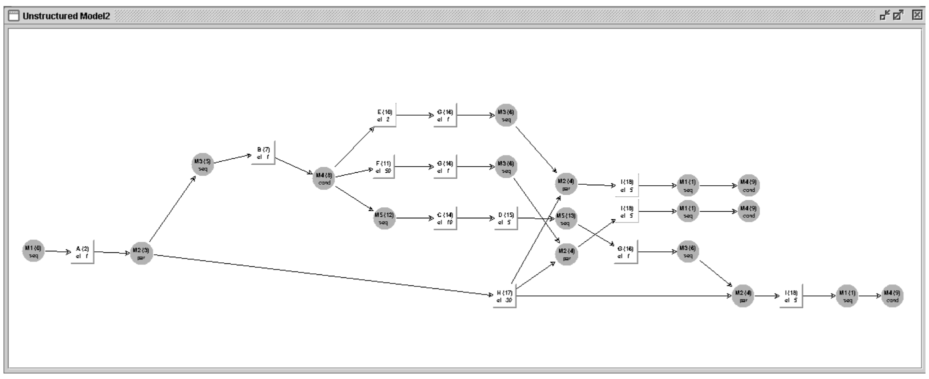


Fig. 8. After Unfold

of workflow models as the set of admitted partial orders of their basic activities. The main contribution of this work is the development of a set of basic schema transformation that maintain this semantics.

There are several applications for the presented methodology. It serves as sound basis for design tools. It enables analysts and designers to start from an initial model and improve the quality of the model step by step. We can provide automatic support to achieve certain presentation characteristics of a workflow model. A model can be transformed to inspect it from different points of view. In particular a model suitable for conceptual comprehension can be transformed to a model better suited for implementation. For workflow evolution it is possible to isolate those parts of a workflow which is actually changed. For cooperating workflows it is possible to arrange a workflow model in a way, that those parts of a workflow which are visible from outside are raised to the highest level while only internal parts of a workflow are hidden in complex activities. Workflow definitions can also be prepared to applied in different organizational settings, where activities are distributed among other application systems.

References

1. Work Group 1. Interface 1: Process definition interchange. *Workflow Management Coalition*, V 1.1 Final(WfMC-TC-1016-P), October 1999.
2. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. *Lecture Notes in Computer Science 1021*. Springer, 1995.
3. J. Eder and W. Gruber. A Meta Model for Structured Workflows Supporting Workflow Transformations. *Lecture Notes in Computer Science 2435*. Springer, 2002.
4. J. Eder, W. Gruber, and E. Panagos. Temporal modeling of workflows with conditional execution paths. *Lecture Notes in Computer Science 1873*. Springer, 2000.
5. J. Eder, H. Groiss, and W. Liebhart. The workflow management system panta rhei. In A. Dogac, et. al. (eds.), *Advances in Workflow Management Systems and Interoperability, Springer, 1997.*
6. J. Eder and W. Liebhart. The workflow activity model WAMO. In S. Laufmann, et.al, editors, *Cooperative Information Systems, 3rd Int. Conf., CoopIS, 1995.*
7. J. Eder and M. Saringer. Workflow Evolution: Generation of Hybrid Flows. in: OOIS,*Lecture Notes in Computer Science 2817*. Springer, 2000.
8. D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.
9. W. Gruber. Modeling and Transformation of Workflows with Temporal Constraints. Akademische Verlagsgesellschaft, Berlin, 2004.
10. D. Hollingsworth. The workflow reference model. *Workflow Management Coalition*, Issue 1.1(TC00-1003), January 1995.
11. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. *Lecture Notes in Computer Science 1789*. Springer, 1999.
12. P. Lawrence. *Workflow Handbook*. John Wiley and Sons, New York, 1997.
13. W. Liebhart. *Fehler- und Ausnahmebehandlung im Workflow Management*. PhD thesis, Universität Klagenfurt, 1998.
14. O. Marjanovic and M. E. Orlowska. On modeling and verification of temporal constraints in production workflows. *Knowledge and Information Systems, KAIS*, vol 1. Springer, 1999.
15. W. Sadiq and M. E. Orlowska. On business process model transformations. *Lecture Notes in Computer Science 1920*. Springer, 2000.
16. W. M. P. van der Aalst, et.al. Advanced workflow patterns. *Lecture Notes in Computer Science 1901*. Springer, 2000.