Informatik Forsch. Entw. (2004) 18: 132–151
Digital Object Identifier (DOI) 10.1007/s00450-004-0157-5

Informatik
Forschung und Entwicklung

Dealing with forward and backward jumps in workflow management systems

Manfred Reichert[1], **Peter Dadam**[1], **Thomas Bauer**[2]

[1] University of Ulm, Dept. Databases and Information Systems, 89069 Ulm, Germany;
E-mail: {reichert,dadam}@informatik.uni-ulm.de
[2] DaimlerChrysler Research and Technology Ulm, Dept. RIC/ED, Postfach 2360, 89013 Ulm, Germany;
E-mail: thomas.tb.bauer@daimlerchrysler.com

Received: 6 October 2002 / Accepted: 8 January 2003

Abstract. Workflow management systems (WfMS) offer a promising technology for the realization of process-centered application systems. A deficiency of existing WfMS is their inadequate support for dealing with exceptional deviations from the standard procedure. In the ADEPT project, therefore, we have developed advanced concepts for workflow modeling and execution, which aim at the increase of flexibility in WfMS. On the one hand we allow workflow designers to model exceptional execution paths already at buildtime provided that these deviations are known in advance. On the other hand authorized users may dynamically deviate from the pre-modeled workflow at runtime as well in order to deal with unforeseen events. In this paper, we focus on forward and backward jumps needed in this context. We describe sophisticated modeling concepts for capturing deviations in workflow models already at buildtime, and we show how forward and backward jumps (of different semantics) can be correctly applied in an ad-hoc manner during runtime as well. We work out basic requirements, facilities, and limitations arising in this context. Our experiences with applications from different domains have shown that the developed concepts will form a key part of process flexibility in process-centered information systems.

Key words: Workflow management – Adaptive workflow – Exception handling – Forward/backward jump

1 Introduction

E-Business has significantly increased the competitive pressure companies must face [4]. To meet this challenge enterprises are developing a growing interest in supporting their business processes more effectively and in streamlining their application systems such that they behave "process-oriented"; i.e., to offer the right tasks at the right point in time to the right persons along with the information and application functions needed. Workflow management systems (WfMS) like MQSeries Workflow, Staffware, or INCOME Workflow offer a promising technology for this [33,58]. Designed for a distributed environment they increase the number of work processes (workflow; abbr. WF) that can pass through an electronic workplace. For this purpose, the business process logic is extracted from application code. So, instead of a large, monolithic program package we obtain a set of WF activities which represent the application functions. The process logic between them (i.e, control and data flow) is specified in a separate WF schema. Usually, for WF modeling graphical formalisms like Petri Nets [1,38,58], Statecharts [32,60], UML Activity Diagrams [16], or block-structured description languages [13,36,41] are used. They allow the WF designer to quickly define and modify WF schemes at a high semantic level, and enable the buildtime components of the WfMS to detect behavioral inconsistencies and errors in a very early implementation stage [46,47,52,58].

Long regarded as technology for the automation of well-structured, repetitive processes, showing only little variations in their possible execution sequences, WF management is in the throes of transformation as more and more non-traditional applications require comprehensive process support as well. In many domains, like hospitals, engineering environments, or E-Commerce, however, process-oriented information systems will not be accepted if rigidity comes with them [4,8,14,18,26]. Instead users must be able to flexibly deviate from the standard process (e.g., by skipping WF activities or by working on a WF activity ahead of the normal schedule), in particular to handle exceptional situations [45,50]. (In this paper exceptions constitute events which may occur during WF execution and which require deviations from the standard business process.) In doing so, it is very important that the use of the WfMS is not more cumbersome and time-consuming than simply handling the exception by a telephone call to the right person. As reported by several groups, insufficient flexibility and adaptability have been primary reasons why many WfMS failed in process automation projects in the past [17,19,41].

Generally, we have to differ between deviations that can be pre-planned and deviations for which this is not possible.

This paper is a revised and extended version of [40]. The described work was partially performed in the research project "Scalability in Adaptive Workflow Management Systems" funded by the Deutsche Forschungsgemeinschaft (DFG).

Concerning pre-planned deviations, their context as well as the actions necessary to handle them are known beforehand. They, therefore, can be already considered at buildtime in order to achieve a flexible WF execution behavior. As opposed to this, deviations that cannot be pre-planned may become necessary to deal with unforeseen events and must be dynamically handled during WF execution. In practice, both kinds of deviations frequently occur and must therefore be adequately supported by WfMS.

The present work is embedded in the ADEPT project which aims at the flexible support of enterprise-wide business processes [5,14,41]. We have developed and implemented advanced concepts for the modeling, execution, and monitoring of workflows as well as for the dynamic change of in-progress WF instances. Our work is based on first-hand knowledge with clinical as well as engineering workflows [8,14]. We have observed that many exceptions are known in advance and can therefore be considered already at buildtime, which decreases the necessity of "expensive" ad-hoc interventions during runtime. To enable users to cope with unforeseen exceptions as well, additionally, we offer advanced concepts for dynamic WF changes. They are based on the ADEPTflex calculus which enables authorized users to dynamically change the structure, the state, and the attributes of in-progress WF instances in a consistent manner and at a high semantic level [41].

In this paper we develop advanced concepts for both, the increase of flexibility at buildtime and its enhancement during runtime. Thereby, we focus on the support of forward and backward jumps, which are indispensable to flexibly deal with exceptions in WfMS [14]. While the former enable deviations in forward direction (e.g., to skip unnecessary activities or to work on a particular activity ahead of the normal schedule), backward jumps make it possible to partially roll back the flow to a previous execution state and to re-continue work in this state (e.g., when activity execution fails). We present concepts for both, the pre-modeling of jumps at buildtime and their dynamic application during runtime. To better understand related issues and problems, we consider the viewpoint of the WF designer as well as of the end user. In some respects forward and backward jumps bear resemblance to GOTO statements in programming languages. However, deviations from standard procedures concern exception handling at a higher semantic level, which is indispensable for WfMS to cover a broad spectrum of processes. (Note that the need for supporting jump operations has been approved by several other research groups as well [1,27,36,42,50].) Nevertheless, jumps must not be complicated for users or lead to an undefined execution behavior. For this reason, ADEPT imposes several restrictions for their use, which either have to be checked at buildtime (pre-planned jumps) or must be ensured when applying the jump during runtime (ad-hoc jumps). Backward jumps, for example, must always result in a former state of the WF instance in order to guarantee a consistent execution behavior. Forward jumps, in turn, must not lead to activity program invocations with missing input data or to skipping of imperative activities. Finally, jump operations must be properly integrated with respect to authorization and documentation.

Although very important for realizing and adaptive workflows, forward and backward jumps do not cover all exception handling procedures needed in practice. As we have reported in earlier papers [14,41], ADEPT provides other facilities as well. Examples include the ad-hoc insertion or deletion of WF activities, the late modeling of sub-workflows, and the dynamic change of WF attributes (e.g., activity work assignments). In addition, several research groups have used the ADEPT WfMS for implementing sophisticated exception handling procedures on top of it, like the automatic adaptation of workflows or the dynamic creation of WF instances as response to occurring exceptions [4,53]. In this context, ECA rules (Event – Condition – Action) can be used to describe the conditions leading to an exception and the actions necessary to handle them [11,36]. Other exception handling approaches are discussed in Sect. 5.

The outline of this paper is as follows: Sect. 2 furnishes basic information about WF modeling and execution in ADEPT – background information which is necessary for a further understanding of this paper. Section Sect. **??**forward describes how pre-planned as well as ad-hoc forward jumps can be flexibly realized in WfMS. In Sect. 4 we set out how backward jumps have to be handled. We discuss related work in Sect. 5 and conclude with a summary in Sect. 6.

2 Background information

For each business process type to be supported, a corresponding WF schema has to be defined and stored in the WfMS. An example is depicted in Fig. 1. Among other things, the diagrammed WF schema defines WF activities as well as the control and data flow between them. The work presented in this paper uses the ADEPT formalism [39,41] for WF modeling and execution. On the one hand this WF meta model is expressive enough to adequately model real-world processes [14], on the other hand, the resulting WF models are easy to understand for WF designers as well as for end users. ADEPT allows to model aspects like control and data flow, work assignments, or time constraints. Furthermore it has proven itself by enabling correctness saving rules (e.g., no deadlocks, no data losses, no temporal inconsistencies, no undefined work assignments), ad-hoc changes of in-progress WF instances [41], and distributed WF execution [5,6]. By implementing these concepts in a powerful prototype [29], we have demonstrated that they work in conjunction with each other as well. In the meantime, the ADEPT prototype is used by research groups from different application domains for the implementation of flexible, process-centered information systems [4,53].

2.1 Control flow modeling and execution

The control flow schema of a WF is represented by an attributed WF graph with distinguishable node and edge types. As shown in [41] this enables efficient correctness checks (see below) and eases the handling of loop backs. Formally, a control flow schema S corresponds to a tuple $S = (N, E, ...)$ with node set N and edge set E. To each control flow edge $e \in E$ an edge type $ET(e)$ from the set $EdgeTypes = \{\texttt{CONTROL\_E}, \texttt{SYNC\_E}, \texttt{LOOP\_E}\}$ is assigned: CONTROL_E denotes "normal" order relations between activities, SYNC_E "wait-for" relations between activities of parallel branches, and LOOP_E loop backs. Similarly, each node $n \in N$ has a node type $NT(n) \in NodeTypes$ (with

Table 1. Predecessor and successor functions defined on WF graphs

`c_succ(n)/c_pred(n)`	set of all *direct* successors/predecessors of activity `n` considering control edges with type `CONTROL_E`
`c_succ*(n)/c_pred*(n)`	set of all *direct* or *indirect* successors / predecessors of activity `n` considering control edges with type `CONTROL_E` (transitive closure)
`succ(n)/pred(n)`	set of all *direct* successors/predecessors of activity `n` referring to control edges with type `CONTROL_E` or `SYNC_E`
`succ*(n)/pred*(n)`	set of all *direct* and *indirect* successors/predecessors of activity `n` referring to control edges with type `CONTROL_E` or `SYNC_E`

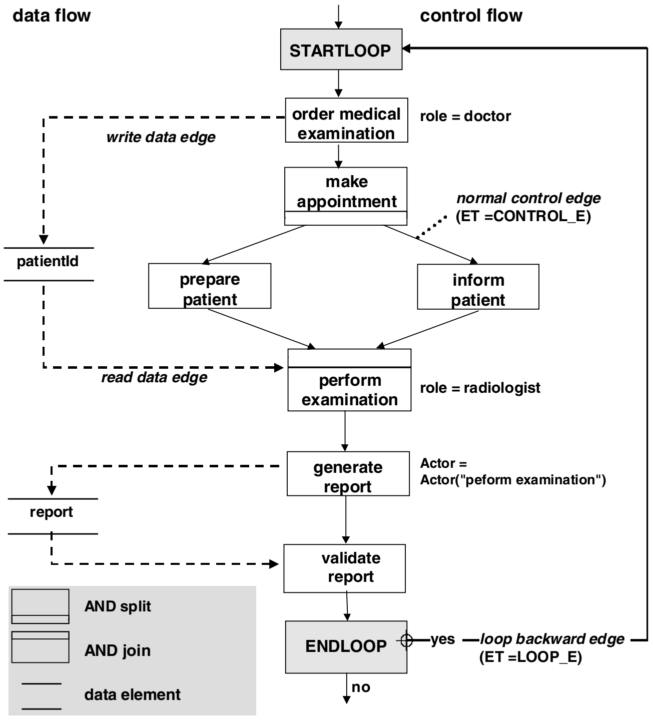


Fig. 1. Workflow modeling in ADEPT

$NodeTypes$:= {STARTFLOW, ENDFLOW, ACTIVITY, STARTLOOP, ENDLOOP, AND Split, XOR Split, AND Join, XOR Join}). Based on these ingredients, sequences, parallel branchings (AND split, AND join), conditional branchings (AND/XOR split, XOR join), and loops (STARTLOOP, ENDLOOP) can be easily modeled. For this we have adopted concepts from block-structured process description languages [15] and enriched them by additional control structures. Branchings as well as loops have exactly one entry and one exit node. Control blocks may be nested but are not allowed to overlap. As this limits expressive power, in addition, the already mentioned synchronization edges are offered to WF designers, which allows them to describe more complex control structures if required. We have selected this block structure because it is rather quickly understood by users, it allows the provision of user-friendly, syntax-driven model editors, and it makes it possible to implement efficient algorithms for control and data flow analyses. – Table 1 informally summarizes predecessor and successor functions on WF graphs which are needed for the following considerations.

Based on a given WF schema S new WF instances can be created and started. To determine which activities are to be executed next, for each WF instance we maintain information about its current state by assigning markings $NS(n)$ and $ES(e)$ to each activity node n and to each control edge e. Corresponding to this, a WF graph with associated markings is denoted as a *WF instance graph*.[1] An example is depicted in Fig. 2. It shows two WF instances created from the WF schema from Fig. 1. Similar to Petri Nets [58], markings are determined by well defined firing rules [41]. In doing so, markings of already passed regions are maintained (except loop backs). Furthermore nodes and edges belonging to non-selected branches of a conditional branching will be explicitly marked as SKIPPED and FALSE_SIGNALED, respectively. ADEPT ensures well-defined dynamic properties, including the absence of deadlocks, the proper termination of the flow, and the reachability of markings which enable activity execution (for details see [39,41]). The described block structuring as well as the used node and edge types help us to accomplish this in an efficient manner. Deadlocks, for example, can be excluded if the WF graph does not contain cycles over control and synchronization edges (see [39] for details).

State transitions of a single activity instance are depicted in Fig. 3. Initially, the activity status is set to NOT_ACTIVATED. It is changed to ACTIVATED when all pre-conditions for executing this activity are met. In this case corresponding work items are inserted into the worklists of authorized users (determined by role-based work assignments). If one of them selects the respective item from his worklist, activity status changes to RUNNING and respective work items are removed from other worklists. Furthermore, an application component associated with the activity is started. At successful termination, activity status passes to COMPLETED.

2.2 Data flow modeling and data context management

Data exchange between activities is realized by the use of global process variables (called *data elements* in the following). Data elements are connected with input and output parameters of WF activities. Each activity input parameter is mapped to exactly one data element by a read data edge and each activity output parameter is connected to a data element

[1] Note that this must not mean that for each WF instance a separate WF graph is maintained. A WF instance graph represents a logical view on a WF instance, and does not give any hint concerning its physical representation.

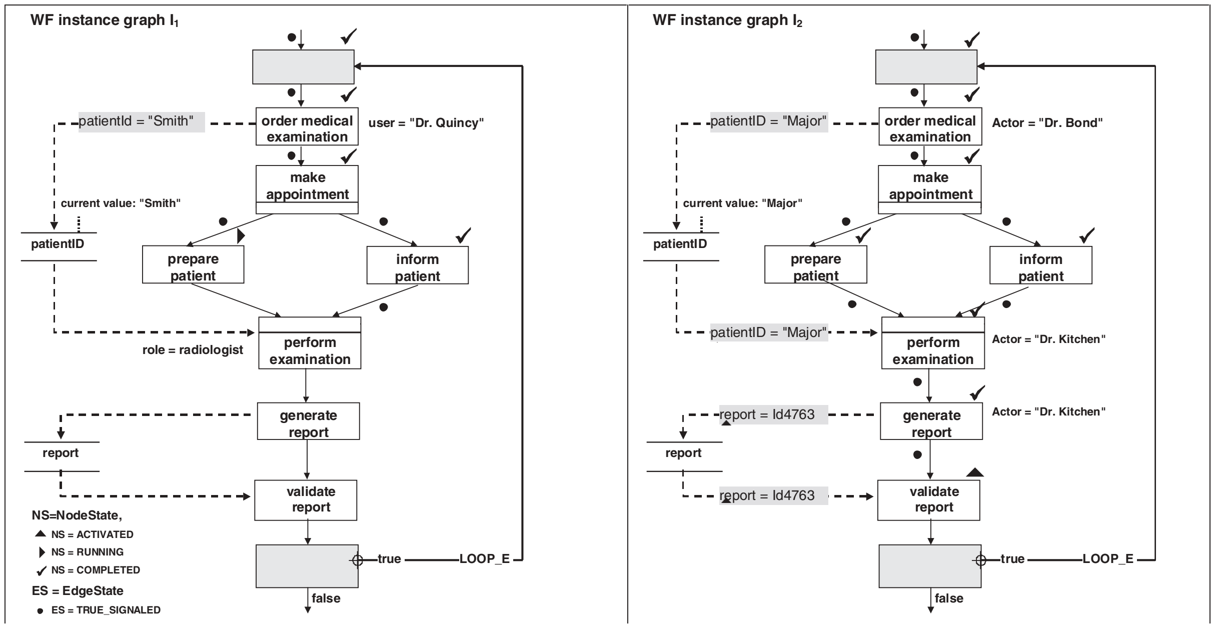


Fig. 2. WF instance graphs (with different marking)

by a write data edge. An example is depicted in Fig. 1. Activity “order medical examination” writes the data element “patientID” which is read by the subsequent activity “perform examination”. The total collection of data elements and data edges is called the data flow schema. For its modeling, a number of correctness properties must be satisfied. The most important one ensures that all data elements read by an activity X must have been written by preceding activities before X can be started, independently from the execution path leading to activation of X. Note that this property is crucial for the proper invocation of external activity programs during WF execution. In particular, it must be ensured in conjunction with forward jumps as well (cf. Sect. 3). Other correctness constraints concern the avoidance of lost updates due to parallel or subsequent write operations on data elements (see [39]). At runtime, if required, ADEPT stores different versions of a data object for each data element. In more detail, for each write access to a data element, always a new version of the data object is created and stored in the WfMS database; i.e., data objects are not physically overwritten. This allows us, for example, to use different versions of a data element within different branches of an AND-/XOR-branching (cf. Sect. 3). As we will see in Sect. 4, however, maintaining data object versions is not only important for the context-dependent reading of data elements but also for the correct rollback of WF instances in case of failures.

3 Forward jumps

In this section we present both, pre-planned and ad-hoc forward jumps for exception handling. While the former are known at buildtime and can therefore be captured in the WF schema, ad-hoc jumps become necessary to deal with unforeseen events. That means they cannot be pre-modeled and must therefore be defined by users at runtime. We motivate the need for both kinds of forward jumps, discuss general issues related

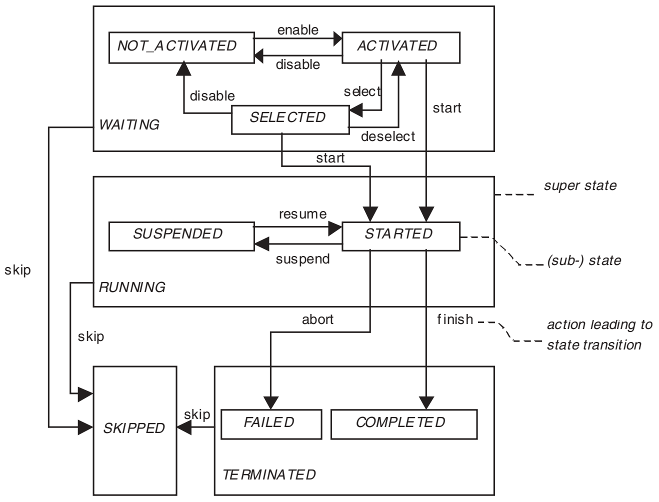


Fig. 3. Statechart for activity state transitions

to them, and show how forward jumps of different semantics have been realized in ADEPT.

3.1 Motivation

During WF execution it may be required to omit unnecessary activities or to immediately execute activities though not all steps normally preceding them in the flow of control have been finished yet.

Example 1 ((Forward jumps in a flow):). The processing of a medical examination for an inpatient normally comprises several steps: The examination must be ordered, an appointment with the examination unit be made, the patient be prepared and notified about potential risks, the intervention be performed, and medical reports be generated, obtained and validated. Even for this simple process chain, it must be possible for physicians and nurses to flexibly deviate from the standard procedure, in particular to handle exceptional situations (e.g., if the patient's state of health gets worse during the process or the physician finds out that some preparatory steps are unnecessary for the respective patient). In such cases it must be possible to skip steps or to immediately perform the examination; i.e., without making an appointment or waiting until all preparatory steps (required in the normal case) have been finished. Note that corresponding situations may occur at any time during process execution. (A presentation of more sophisticated examples is given in [49].)

Our experiences with clinical as well as engineering processes [8, 14, 49] have shown that very often deviations from standard processes can be pre-planned. If exceptional situations, in which a particular activity or a set of activities is to be processed ahead of the normal schedule, are known in advance, it must be possible to capture them at buildtime. This, in turn, enables the WfMS to offer respective activities as exceptional steps to users though the pre-conditions for their regular execution have not been fully met; i.e., users may work "untimely" on these pre-scheduled activities, but the WfMS indicates them that activity execution constitutes a deviation from the preferred execution path in the current WF state. How this can be accomplished is described in the following subsection.

3.2 Defining and changing execution priorities for WF activities

In this section we introduce simple, but useful extensions of the ADEPT base model, which allow the WF designer to differentiate between normal and exceptional execution paths.

3.2.1 Defining activity priorities at build-time

To enable WF designers to express whether a scheduled activity is to be offered as a "normal" or as an "exceptional" step within worklists, we allow them to associate *execution priorities* with activities. If an activity is to be treated as an exceptional step when it is activated, priority EXCEPTIONAL will have to be assigned to it at buildtime; otherwise the setting REGULAR will be chosen as priority for this activity (default setting). Internally, the WF engine schedules activities independently from their execution priority; i.e., execution of an activity with priority EXCEPTIONAL follows the same rules (with respect to activation and termination) as execution of regular activities. The way how activities are offered in user worklists, however, may depend on the priority assigned to them and is left to WF client applications. Possible worklist visualizations include the fade-in/fade-out of exceptional activities, the use of different colors for activities with different priorities, or the display of the work items related to these activities within different windows. Combined with AND-split/XOR-join branchings – called AND-/XOR-branching for short – activity priorities turn out to be very useful: When the AND-split node of such a branching completes, its outgoing branches are activated and can be worked on concurrently. As opposed to parallel branchings (with AND-split/AND-join), however,

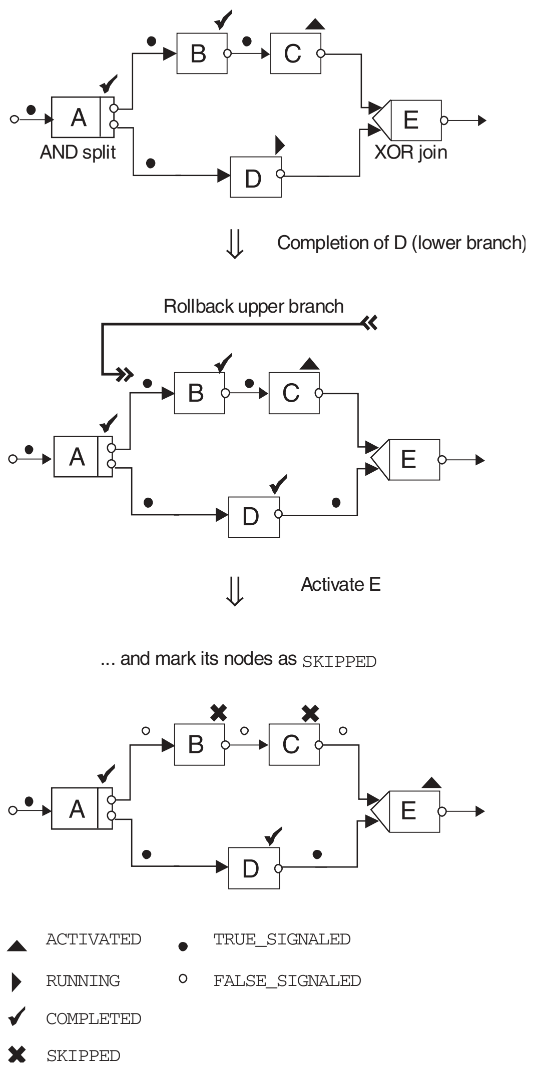


Fig. 4. Operational semantics of an AND-/XOR-branching

the workflow may proceed at the XOR-join as soon as one of its incoming branches is terminated.[2] In ADEPT, in such a case activities from other branches are removed from user worklists, aborted or compensated [3] depending on their current state. An example showing the operational semantics of an AND-/XOR-branching is depicted in Fig. 4.

Based on this, WF designers are able to differentiate between preferred execution paths and exceptional ones. A simple example is given in Fig. 5. In the depicted WF instance graph activities B and X are concurrently active. Due to the assigned priorities B is offered as regular step in user worklists whereas X is treated as exceptional activity. According to this,

[2] Synchronization of incoming branches at an XOR-join is serialized, i.e., there will be always one branch that terminates first. By default, this branch is selected as "winner" and the other branches are rolled back. Alternatively, ADEPT allows more than one incoming branch to be completed such that the user can explicitly select the most suited one (e.g., depending on the output data generated by related activities).

[3] Whether a completed activity can be compensated or not may depend on the kind of activity as well as on the current state of the WF.

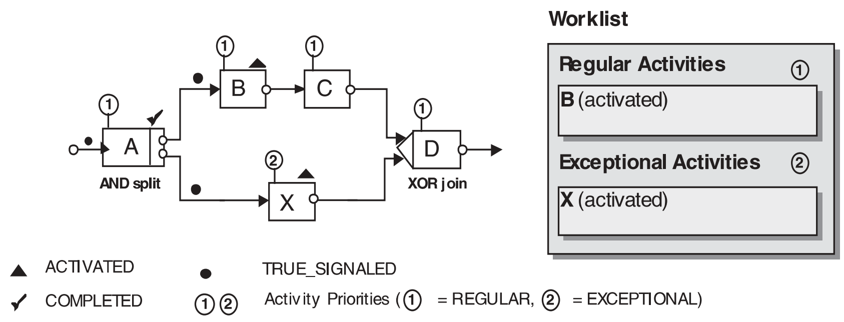


Fig. 5. Activity priorities

the preferred activity sequence is defined by A, B, C, and D (in the given order) whereas the sequence A, X, and D constitutes an exceptional path (indicated by priority EXCEPTIONAL of activity X); i.e., instead of B and C activity X may be executed.

3.2.2 Pre-planned changes of activity priorities during run-time

Depending on the state of a WF instance it must be possible to treat a particular activity differently with respect to its prioritization. Under certain circumstances its execution may constitute a deviation whereas in other WF states it can be treated as normal step. As an example, take an activity X which normally is to be executed after some preceding steps have been finished. Due to exceptional situations, however, it may become necessary to work on X ahead of the normal schedule. In this case, execution of X constitutes a deviation from the preferred activity sequence which must be indicated to users; i.e., this exceptional state must be preserved as long as not all activities normally preceding X in the flow have been completed. If the latter case occurs, however, priority of X is to be changed accordingly.

Static priorities are not sufficient for modeling such cases. In addition, it must be possible to dynamically modify priorities during WF execution. In order to be able to capture such priority changes in the WF model we introduce *prioritization edges* as an additional modeling concept. A prioritization edge $e_p = src \rightarrow dst$ links two activities src and dst, but without enforcing an execution order between them. Each prioritization edge e_p is associated with a priority $e_p{}^{priority} \in \{\texttt{REGULAR}, \texttt{EXCEPTIONAL}\}$ which has the following semantics: When the source node src is completed, edge e_p is signaled as TRUE and the priority of its destination node dst is modified to $e_p{}^{priority}$. As an example take an activity Y with (static) priority EXCEPTIONAL. If an incoming prioritization edge (with priority REGULAR) of this activity is signaled as TRUE during runtime, the priority of Y will be set to REGULAR as well; i.e., from this point in time the execution of Y no longer constitutes an exception.

Figure 6 shows an example for the combined use of prioritization edges and activity priorities. In the WF instance graph depicted in Fig. 6a, activities C and D are concurrently active whereas C is treated as normal step and the execution of D is considered as an exception (corresponding to the static priorities assigned to C and D). After completion of C its outgoing prioritization edge C $\rightarrow$ D (with priority REGULAR) is signaled as TRUE. According to this, priority of activity D is changed from EXCEPTIONAL to REGULAR such that D can

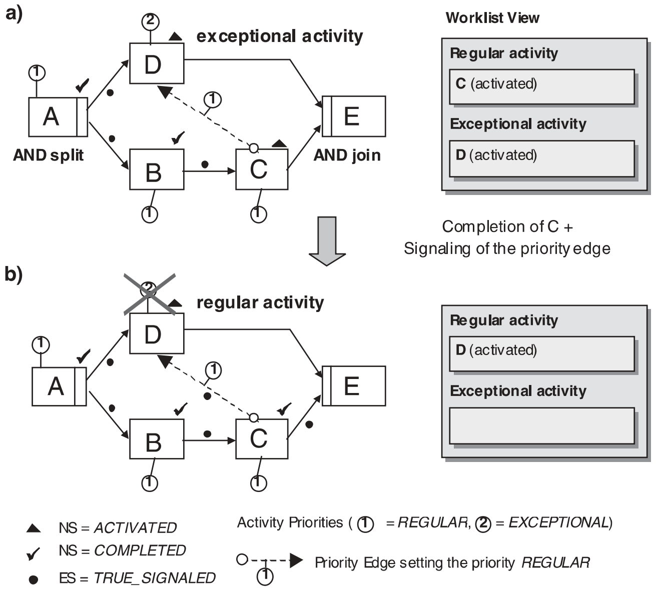


Fig. 6. Changing priorities during the execution of a WF instance

be offered as normal step in user worklists (cf. Fig. 6b). Recapitulating this, the preferred activity sequence is A, B, C, D, E, though D may be executed directly after completion of A.

3.3 Modeling forward jumps at buildtime

As motivated, it often becomes necessary to give priority to execution of activities though the steps normally preceding them have not been completely finished yet. ADEPT provides the needed flexibility by allowing users to jump forward to selected activities and, therefore, to bring forward their execution. To better understand the problems arising in this context, we must consider both, the viewpoint of the designer and of the end user. For the designer it must be possible to differentiate between the normal course a workflow shall take and exceptional deviations. Furthermore, by incorporating pre-planned jumps the clarity of the WF model must not suffer and the complexity for its creation must not be significantly increased. From the viewpoint of end users, it is very important that they are able to differ between activities scheduled along the normal flow and activities whose execution (currently) constitutes a deviation.

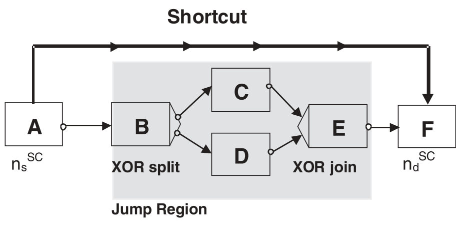


Fig. 7. Modeling shortcuts (viewpoint of the WF designer)

3.3.1 General issues

When deviating from the preferred activity sequence, one has to decide how bypassed activities are to be treated. Generally, they may either be skipped or be continued and finished concurrently to the untimely executed activities. ADEPT supports both variants as well as mixtures of them. In the following, we show how the modeling concepts from Sect. 3.2 can be used to define pre-planned forward jumps at buildtime. As a prerequisite the WF states in which the forward jump shall be applicable and the activities of which the execution shall be brought forward due to the jump must be known in advance. Pre-planned jumps can be defined at a high semantic level. For this, a graphical WF description language is offered which comprises the elements of the ADEPT base model (cf. Sect. 2) as well as modeling elements for defining jumps. For the definition of forward jumps a special edge type – called *shortcut* – can be used. Internally, a shortcut $n_s^{SC} \rightarrow n_d^{SC}$ is transformed into a representation of the ADEPT base model whereby a precise operational semantics can be guaranteed.

A simple example is depicted in Fig. 7. The diagrammed shortcut A $\rightarrow$ F reflects a forward jump as it is modeled by the designer. It has the following semantics: After successful completion of source activity n_s^{SC} (activity A in our example), both, direct successors of n_s^{SC} over normal control edges (activity B in our example) and the target activity n_d^{SC} of the shortcut (activity F in our example) are activated. While B is treated as a normal step (with priority REGULAR), the "untimely" execution of activity F is considered as an exceptional case. This exceptional status is preserved as long as F is not scheduled along the "normal" control flow. In our example, F is offered as exceptional step (with priority EXCEPTIONAL) in worklists until it will either be finished or its direct predecessor E will be completed.

Independent of the semantics of a shortcut $n_s^{SC} \rightarrow n_d^{SC}$ its transformation into an executable representation of the ADEPT base model requires the following conditions:

- The source node n_s^{SC} of the shortcut must be a predecessor of its target node n_d^{SC} over normal control edges (with edge type CONTROL_E). Formally: $n_d^{SC} \in c\_succ^*(n_s^{SC})$
- The shortcut must not partially overlap with a loop control block $(Loop_{Start}, Loop_{End})$, but loop control blocks and shortcuts may contain each other:[4]
  $n_s^{SC} \in L_{body} \Leftrightarrow n_d^{SC} \in L_{body}(with$
  $L_{body} := c\_succ^*(Loop_{Start}) \cap c\_pred^*(Loop_{End}))$

[4] Concerning the schema from Fig. 1, for example, shortcut "order medical examination" $\rightarrow$ "perform examination" would be allowed. As opposed to this, it would not be possible to model a shortcut leading from activity "order medical examination" to an activity succeeding the end node of the depicted loop.

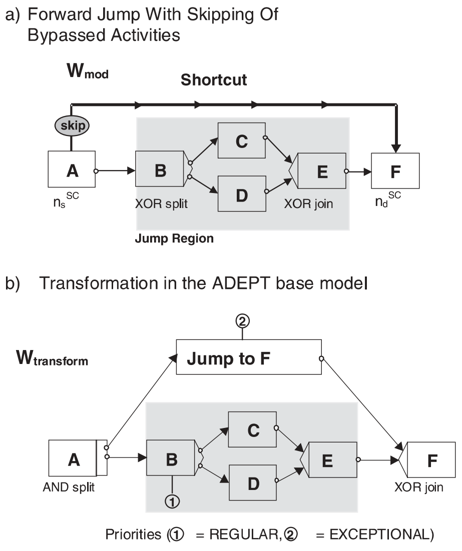

a) Forward Jump With Skipping Of Bypassed Activities
W_mod
Shortcut
skip
A
B
C
D
E
F
n_s^{SC}
XOR split
XOR join
n_d^{SC}
Jump Region
b) Transformation in the ADEPT base model
W_transform
②
Jump to F
A
B
C
D
E
F
AND split
①
XOR join
Priorities (① = REGULAR, ② = EXCEPTIONAL)

Fig. 8. Forward jump (with skipping bypassed activities)

Assume that a user wants to follow shortcut $n_s^{SC} \rightarrow n_d^{SC}$; i.e., he wants to jump forward to activity n_d^{SC} and work on it though not all steps normally preceding n_d^{SC} have been finished yet. When deviating from the preferred execution order, it must be clear how to deal with bypassed activities from the *jump region*; i.e., with activities located between the source and the target node of the shortcut (nodes B, C or D, E in our example). We present two alternative approaches which either allow to skip bypassed activities or to finish them concurrently to activity n_d^{SC}.

3.3.2 Skipping bypassed activities

One possibility to deal with bypassed activities is to undo, abort, or abandon their execution depending on their current state; i.e., if a user deviates from the preferred execution path by following a shortcut $n_s^{SC} \rightarrow n_d^{SC}$, all activities located between the source and the target activity of the shortcut will be compensated, aborted, or abandoned – these activities form the *jump region* of the shortcut and are defined by the set $N_{bypass} := c\_succ^*(n_s^{SC}) \cap c\_pred^*(n_d^{SC})$ (nodes B, C, D, and E in our example from Fig. 8). Afterwards, the execution of the flow will proceed with node n_d^{SC} (node F in our example).

The WF graph W_{mod} in Fig. 8a contains a shortcut (A $\rightarrow$ F) as it is modeled by the WF designer. The WF graph $W_{transform}$ in Fig. 8b shows the transformation of this shortcut into a representation of the ADEPT base model. As one can easily see, this transformation is based on the combined use of an AND-/XOR-branching (defined by nodes A and F in our example) and activity priorities. In more detail, the upper branch of the created AND-/XOR-branching consists of a single *jump activity* with priority EXCEPTIONAL and with label <"Jump to " + target node> ("Jump to F" in our example). The lower branch, in turn, constitutes the jump region and corresponds to that subgraph of W_{mod} induced by nodes from the set N_{bypass} ($\{$B, C, D, E$\}$ in our example). According to $W_{transform}$ the jump activity ("Jump to F") is selectable (with priority EXCEPTIONAL) as soon as the source activity of the shortcut (activity A in our example) is completed. If the jump activity is executed the upper branch of the AND-/XOR-branching will be immediately completed. According to the specified operational semantics, the lower branch (i.e., the jump region) will then be aborted and rolled back. Afterwards the flow can proceed at the target activity of the shortcut (node `F` in our example). As opposed to this, if activities from the lower branch (`B`, `C` or `D`, `E` in our example) are finished, activation of the jump activity will be cancelled and corresponding work items be removed from worklists.

Mapping the shortcut into a representation of the ADEPT base model at buildtime. Generally, the mapping of a shortcut $n_s^{SC} \rightarrow n_d^{SC}$ (with the described semantics) into a representation of the ADEPT base model can be accomplished by applying Algorithm 1 (which has complexity $O(n)$ where n denotes the number of activity nodes of the WF graph; the same complexity results from the algorithms necessary to check the conditions described in Sect. 3.3.1).

Algorithm 1. Transforming a shortcut $n_s^{SC} \rightarrow n_d^{SC}$ (with skipping of bypassed activities) into ADEPT base model:

1. If n_s^{SC} corresponds to a split node, insert a null activity n_1 (i.e., an activity without associated action) which takes over the output firing behavior and outgoing control edges of n_s^{SC}. Set the output firing behavior of activity n_s^{SC} to `ONE_Of_ONE` (i.e., n_s^{SC} is no split node anymore) and add control edge $n_s^{SC} \rightarrow n_1$ (cf. Fig. 9a)
2. Set the output firing behavior of activity n_s^{SC} to `ALL_Of_ALL`; i.e., n_s^{SC} is converted into an AND split node (cf. Fig. 9b).
3. If n_d^{SC} corresponds to a join node, insert a null activity n_2 which takes over the input firing behavior and incoming control edges of n_d^{SC}. Set the input firing behavior of activity n_d^{SC} to `ONE_Of_ONE` (i.e., n_d^{SC} is no join node anymore) and add control edge $n_2 \rightarrow n_d^{SC}$ (cf. Fig. 9c).
4. Set the input firing behavior of n_d^{SC} to `ONE_Of_ALL`; i.e., n_d^{SC} is converted into an XOR join node (cf. Fig. 9d).
5. Insert an additional branch between n_s^{SC} and n_d^{SC} which contains a jump activity (with priority `EXCEPTIONAL`) labeled as <"Jump to" + n_d^{SC} >. The other branch, in turn, corresponds to that sub-graph of W_{mod} induced by the nodes of the jump region (cf. Fig. 9e).

Formal pre-conditions and required checks at buildtime. Obviously, when transforming a shortcut with the described semantics into a representation of ADEPT base (cf. Algorithm 1), steps 1–5 generate an AND-/XOR-branching with two branches: One corresponds to the jump activity (with exceptional priority) and the other to the jump region. In order to ensure structural correctness (i.e., a proper block structuring of the WF graph and the absence of cycles; cf. Sect. 2) and a correct dynamic behavior of the flow (e.g., no deadlocks), in addition to the already mentioned pre-conditions (see above), the following restrictions must be met for the use of a shortcut $n_s^{SC} \rightarrow n_d^{SC}$:

- The subgraph induced by the node set $N_{skip} := N_{bypass} \cup \{n_s^{SC}, n_d^{SC}\}$ must constitute a regular control block; i.e., nodes from branchings and loops are either not contained within N_{skip} or they are completely covered by nodes from this set.

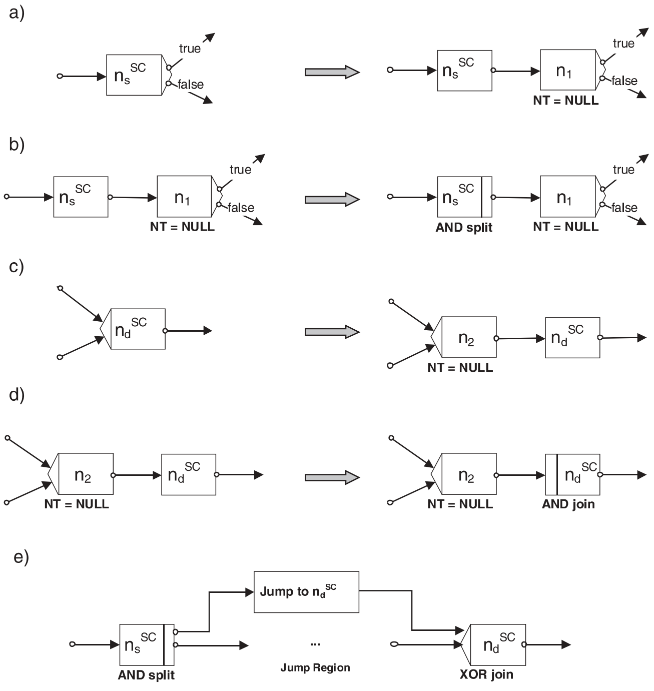


Fig. 9. Shortcut transformation

- Different shortcuts $n_s^{SC} \rightarrow n_d^{SC}$ and $m_s^{SC} \rightarrow m_d^{SC}$ must not overlap but may contain each other. Formally: $n_s^{SC} \in M_{skip} \Leftrightarrow n_d^{SC} \in M_{skip}$ (with $M_{skip} := c\_succ^*(m_s^{SC}) \cap c\_pred^*(m_d^{SC})$)

Both conditions can be easily checked. If the shortcut is applied to a correct control flow schema (i.e., proper block structure, no cycles except loop backs) and if the above conditions are satisfied, steps 1–5 of Algorithm 1 will result in a correct control flow schema again; i.e., structural and dynamic properties as required by the ADEPT base model will be further valid (for a formal treatment see [39]).

When transforming a shortcut into a representation of the ADEPT base model, in addition, we have to check whether the related data flow schema remains correct. As described in Sect. 2, the most important correctness property requires that all data elements read by an (arbitrary) activity `X` must have been written by at least one preceding activity before `X` can be started; in particular, this condition must hold independently of the execution path leading to activation of `X`. This property is ensured by corresponding data flow analyses at buildtime, which make use of the presented block structure and which have complexity $O(n^2)$. The presentation of algorithms, however, is outside the scope of this paper (see [39]).

Necessary data flow analyses can be reduced if the data flow schema statisfies the above correctness property already before the shortcut is defined. It is then sufficient to check whether there are successors of n_d^{SC} (incl. n_d^{SC}) which are data-dependent on (skipped) activities from the jump region. Only for this case there may be missing input data due to the defined shortcut. As an example take the scenario depcited in Fig. 10a). The upper WF schema for which shortcut $\texttt{A} \rightarrow \texttt{D}$ is defined contains data dependencies between `A` and `C` (`C` reads the data element d_1 written by `A`) as well as between `D` and `E` (`E` reads the data element d_2 written by `A`). As shown in the lower

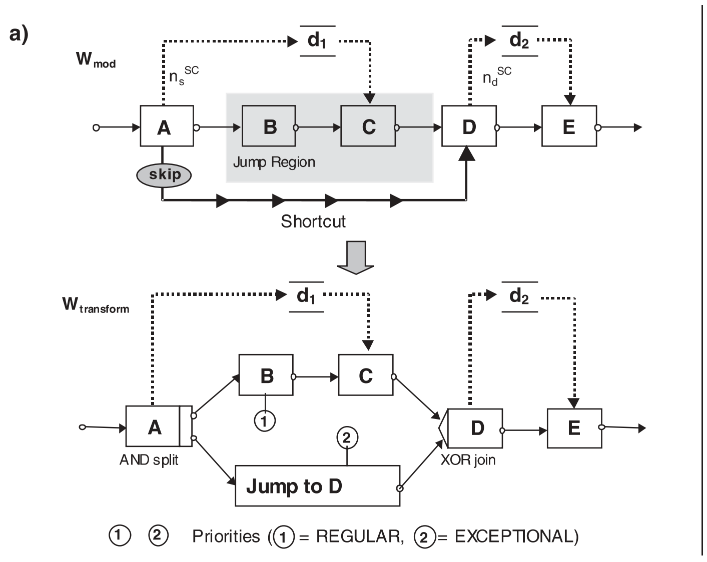


Fig. 10. Checking data flow correctness when transforming a shortcut

WF schema, these dependencies persist when the shortcut is transformed into an ADEPT representation. Furthermore, the data flow schema remains correct since d_1 (d_2) will always be written before C (E) is activated. This does not apply, for example, with respect to the scenario from Fig. 10b). It is very similar to the one shown in Fig. 10a) but takes E as target activity of the shortcut instead of D. Since D is now contained within the jump region it will be compensated, aborted or skipped when applying the forward jump ("Jump to E"). This, in turn, will lead to invocation of E or – more precisely – of its associated activity program with missing input data, which may cause inconsistencies (e.g., wrong outputs) or errors (e.g., program crashes). In ADEPT, therefore, the shortcut from A to E will be either not allowed or the designer will have to restore correctness of the data flow; e.g., by re-linking the corresponding input parameter of E to another data element.

3.3.3 Finishing bypassed activities

When a user wants to apply a jump in order to bring forward the execution of a certain activity, it is not always required to skip activities of the jump region. Instead, it may be desired to continue and finish them concurrently to the pre-scheduled activities. With respect to activities from the jump region, this means that the effects of already completed activities are to be preserved, the execution of already started activities be continued, and activities not yet activated be scheduled as planned. ADEPT allows the modeling of such forward jumps as well. For this, the designer has to specify a shortcut $n_s^{SC} \rightarrow n_d^{SC}$ and an activity n_n^{SC} for synchronizing bypassed steps. At runtime, authorized users may then bring forward the execution of n_d^{SC} as soon as n_s^{SC} is completed. As opposed to the shortcut semantics described above, activities from the jump region are further processed in case the shortcut is followed. In doing so, it is important to synchronize their execution with the overall flow. As mentioned, for this a synchronizing activity n_n^{SC} has to be specified which then may be only activated if its normal preconditions hold and – additionally to this – all

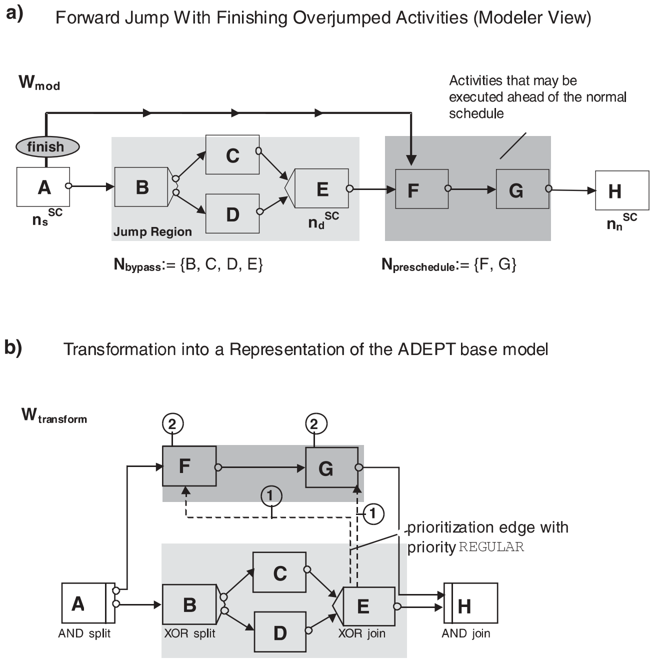

a) Forward Jump With Finishing Overjumped Activities (Modeler View)
Wmod
Activities that may be executed ahead of the normal schedule
finish
A
B
C
D
E
F
G
H
ns SC
nd SC
nn SC
Jump Region
Nbypass:= {B, C, D, E}
Npreschedule:= {F, G}
b) Transformation into a Representation of the ADEPT base model
Wtransform
prioritization edge with priority REGULAR
AND split
XOR split
XOR join
AND join

Fig. 11. Forward jump (with finishing bypassed activities)

activities from the jump region are completed. A simple example is depicted in Fig. 11a). The WF graph W_{mod} contains a shortcut A → F as it is modeled by the designer. In addition, activity node n_s^{SC} = H serves for synchronizing the execution of bypassed activities. According to this, F (and its successor G) may be executed ahead of the normal schedule (i.e., before activity E is completed) as soon as A is finished. In case this forward jump is followed, however, the processing of activities from the jump region (B, C or D, E in our example) must be continued and finished before H can be activated.

The transformation of this shortcut is shown in Fig. 11 b). Activity A now represents an AND-split and H the corresponding AND-join. The execution behavior is as follows: After completing A its successor B can be executed as activity with priority REGULAR and the shortcut's target F as activity with priority EXCEPTIONAL. If no deviation from the preferred execution order occurs during runtime (i.e., only activities with priority REGULAR are processed), the lower branch will be completed before F is started. For this case, the outgoing prioritization edges of E are signaled as TRUE such that the priorities of F and G are changed to REGULAR. As opposed to this, if a user starts F before E is completed this will correspond to a deviation from the preferred activity sequence (indicated by priority EXCEPTIONAL of activity F).

Mapping the shortcut into a representation of the ADEPT base model at buildtime. Assume that shortcut $n_s^{SC} \to n_d^{SC}$ is to be applied to a correct WF schema and bypassed activities are to be finished before the (synchronizing) activity n_n^{SC} may be activated. This semantics can be realized by the combined use the modeling concepts of the ADEPT base model and its extensions as described in Sect. 3.2. To transform a shortcut specification into an executable representation of the ADEPT base model, Algorithm 2 (which has complexity $O(n)$) must be applied (for an example see Fig. 12):

Algorithm 2. Transforming a Forward Jump (with Finishing Bypassed Steps) into ADEPT base model:

1. Determine the minimal control block (n_{start}, n_{end}) that contains activities n_s^{SC}, n_d^{SC}, and n_n^{SC}.
2. Create an AND split n_1 which represents a null activity and takes over the input firing behavior as well as the incoming control edges (of type CONTROL_E) of n_{start}. Link n_{start} as a direct successor to n_1.
3. Create an AND join node n_2 corresponding to n_1; n_2 shall represent a null activity and take over the output firing behavior as well as the outgoing control edges of n_{end}. Link n_{end} as a direct predecessor to n_2.
4. Detach the subgraph induced by $N_{preschedule} := n_d^{SC} \cup c\_succ^*(n_d^{SC}) \cap c\_pred^*(n_n^{SC})$ from its current graph context, and insert it as additional branch to the branching defined by (n_1, n_2). Let x_{start}:= n_d^{SC} be the start and x_{end} be the end node of this subgraph or branch.
5. Add two synchronization edges from n_s^{SC} to x_{start} and from x_{end} to n_n^{SC}.
6. Assign priority EXCEPTIONAL to each node from $N_{preschedule}$ and add prioritization edges (with edge priority REGULAR) from the end node of the jump region to each node of this set.
7. Apply ADEPT reduction rules to eliminate unnecessary nodes and edges (for details see [39,41]).

Formal pre-conditions and required checks at buildtime. For the correct use of a shortcut $n_s^{SC} \to n_d^{SC}$ with described semantics and its transformation into an executable representation of the ADEPT base model the following conditions must be checked (corresponding algorithms make use of the block structuring and have complexity $O(n)$):

- If n_d^{SC} is contained within a branch of an XOR-branching, node n_s^{SC} must be contained within the same branch; i.e., it is not allowed to model a forward jump from an activity preceding an XOR-branching to an activity contained within one of its branches.
- Let $N_{preschedule} := \{n_d^{SC}\} \cup c\_succ^*(n_d^{SC}) \cap c\_pred^*(n_n^{SC})$ comprise activities that may be executed ahead of the normal schedule according to the defined shortcut (cf. Fig. 12). In order to obtain a proper block structure we require that the subgraph induced by $N_{preschedule}$ itself constitutes a block (as it is the case in Fig. 12 where the respective subgraph corresponds to a sequence).
- Let $N^*_{preschedule}$ be another set of activities that may be executed ahead of the normal schedule according to another defined shortcut. Then $N_{preschedule}$ and $N^*_{preschedule}$ must not partially overlap. Formally:
  $(N_{preschedule} \subseteq N^*_{preschedule}) \vee$
  $(N_{preschedule} \supset N^*_{preschedule}) \vee$
  $(N_{preschedule} \cap N^*_{preschedule} = \emptyset)$
  For shortcuts with same target we require that these sets are identical. This may be relevant for forward jumps from different branches of an XOR-branching to the same target activity.

If these restrictions are satisfied the structural and dynamic properties of the WF graph (proper block structure, no cycles except loop backs, no deadlocks, etc.) can be guaranteed for the resulting control flow schema as well. Formal proofs can be found in [39]. Concerning the data flow schema, we must check whether the described transformations may lead to invocation of activities with missing input data or may cause data losses (due to lost updates). If the data flow schema is correct before applying Algorithm 2 the following checks will be sufficient:

- **Checks for avoiding missing input data:** Due to the described transformations, activities from the sets $N_{preschedule}$ and N_{bypass} may now be executed concurrently to each other. For each activity $x \in N_{preschedule}$, therefore, we must check whether data elements read by x are further written by preceding steps. Obviously, if this is the case before introducing the shortcut, it will be sufficient to restrict these checks to those data elements written by activities from N_{bypass}. Note that only order relations of activities from this set are rearranged with respect to nodes from $N_{preschedule}$.
- **Checks for avoiding data losses due to lost updates**: Let $D_{bypass}/D_{preschedule}$ denote the set of data elements to which activities from $N_{bypass}/N_{preschedule}$ have write access. Parallel write operations on data elements (and data loss due to lost updates) will not arise, if $D_{bypass} \cap D_{preschedule} = \emptyset$ holds. If this does not apply, however, the shortcut edge must either be removed or additional synchronization edges between nodes from N_{bypass} and $N_{preschedule}$ have to be inserted.

3.3.4 Concluding remarks

The presented modeling concepts can be generalized. ADEPT also allows the definition of hybrid forms of shortcuts. With respect to nodes from the jump region, for each activity the designer has the choice whether its execution is to be skipped or continued when the jump is applied. Though the graph transformations needed in this context are more complex, from the

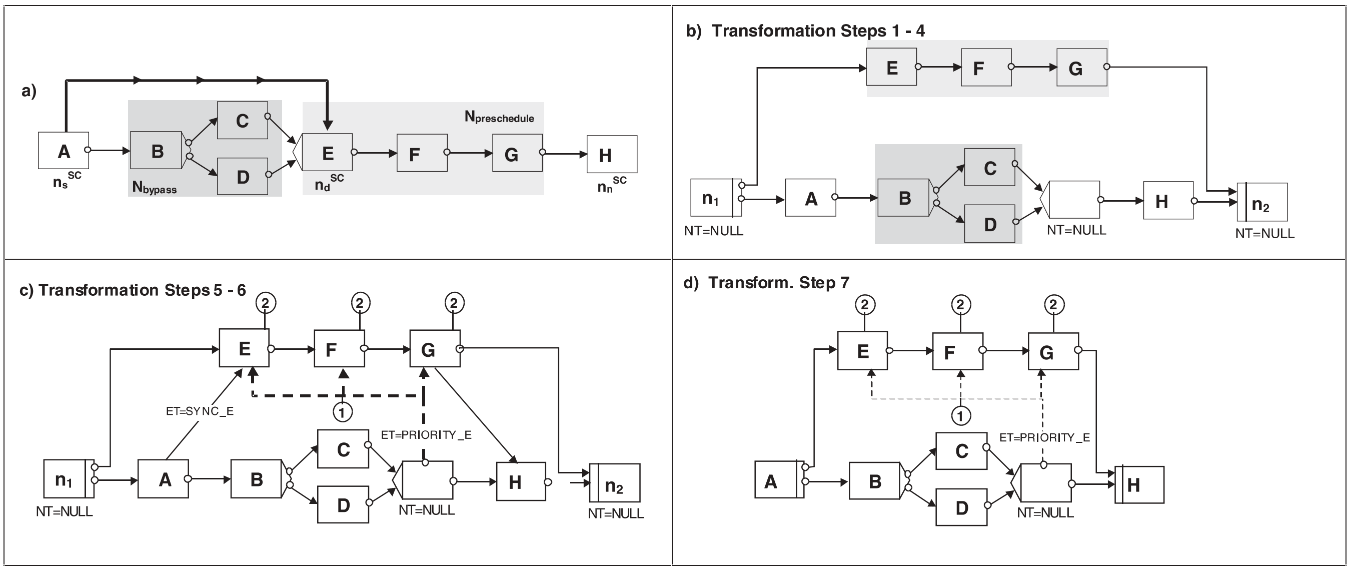


Fig. 12. Transforming a shortcut into ADEPT base representation

conceptual point of view no new issues arise. We, therefore, omit a more detailed presentation. In summary, pre-planned deviations as described above form a key part of process flexibility in WfMS. In addition, they do not require expensive user interactions as it is the case for ad-hoc changes. Unfortunately, it will not always be possible to capture all jumps in the WF schema at buildtime. For this reason, we require additional techniques that allow authorized users to perform forward jumps in an ad-hoc manner as well if need be.

3.4 Performing dynamic forward jumps during runtime

Our experience with clinical workflows has shown that the WF designer is generally not capable to predict all possible deviations in advance and to capture them in the WF schema [14]. To adequately cope with such unforeseen exceptions, in addition to the described concepts, ADEPT supports ad-hoc deviations from the pre-modeled WF schema at the instance level as well (e.g., to insert, to delete, or to shift activities). In the following, we restrict our considerations to dynamic forward jumps (e.g., skipping of a set of activities or immediate execution of an activity though not all predecessors have been completed yet). In this context, it is very important that change definition is not complicated for users; i.e., all complexity associated with missing activity input data (e.g., due to skipping of activities), data losses (e.g., due to lost updates), deadlocks (e.g., due to cyclic waits of activities), or state adaptations must be hidden to a large degree from users. Instead, they must be able to define a dynamic forward jump at a high semantic level without requiring that they are familiar with the used WF description formalism.

3.4.1 Dynamic forward jumps in ADEPT

Generally, dynamic jumps make only sense if there is no risk of activity program crashes, data losses, or any other obscure system behavior. We have spent much effort on the design of high-level change operations that allow users to adapt in-progress WF instances while preserving the mentioned correctness properties (cf. Sect. 2). As response to a dynamic change request, ADEPT, in essence, first checks data dependencies and ordering constraints to detect whether the problem of missing input values, lost updates, or cyclic waits (deadlocks) may occur in the modified WF instance graph. In case of missing input values, we offer the possibility to generate an electronic form and to prompt users for these values (either immediately or when needed). Only if no consistency problem occur or if it is explicitly tolerated by the user the change request will be accepted and the necessary graph transformations be performed. In addition, the markings of nodes and edges are automatically adapted when the change is applied. In ADEPT, users are not burdened with this. They may express a jump request in a rather declarative way and at a high semantic level (e.g., “work immediately on X”, “skip $X_1, X_2, ..., X_n$”). For this, high-level modification operators are offered which are based on the combined use of basic change operations.

Move operation: The move operation constitutes a basic change operation for the (dynamic) rearrangement of activities. In more detail, it allows to shift an activity (or a whole block) from its current position in the WF instance graph to another place provided that the actual state of the instance does not prohibit this. The restructuring of the instance graph, necessary in this context, may be more or less complex depending on the target position the activity is to be re-inserted. For example, an activity detached from its current position may be re-inserted between two succeeding activities, between two sets of succeeding activities, or parallel to a certain activity or control block. A simple example for the use of the move operation is depicted in Fig. 13. Assume that for the WF instance from Fig. 13a) a user wants to work immediately on activity C though B has not yet been finished. In principle, this is possible since C is not data-dependent on B. Depending on the concrete change scenario the user may desire to skip execution of B, first execute C and then work on B (i.e., to swap B and C), or continue the processing of B concurrently

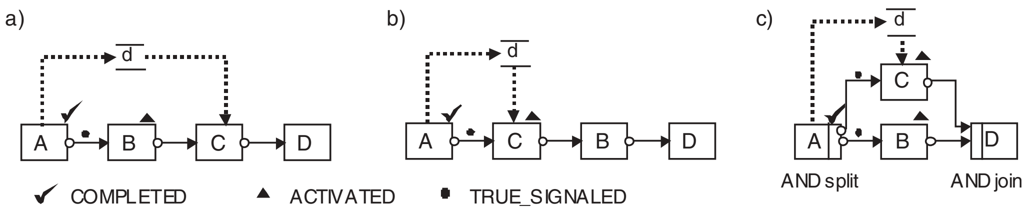


Fig. 13. a Original WF instance graph and WF instance graph resulting after moving C from its current position to the position, **b** ... located between A and B, **c** ... located parallel to B

to C. While the skipping of B is internally based on the delete operation [39] the other two changes can be realized by using the move operation. Figure 13b) shows the WF instance graph after detaching activity C from its current position and re-inserting it between A and B. Note that the re-evaluation of the WF state, which is automatically done in ADEPT, results in activation of C and deactivation of B. Figure 13c) shows the WF instance graph as it results when C is moved to a position parallel to B.

Concerning this example certain pre-conditions must be met in order to correctly apply the move operation. For instance, it would not be possible to move C to a position before A since A and its predecessors have been already finished and therefore an undefined marking would result. But even if activity A had not yet been started, the desired operation would be prohibited by ADEPT, since C would then be invoked with missing input data.

Jump forward operation: Based on the sketched move operation we have realized several high-level operations for dynamic forward jumps. Generally, these jump operations enable authorized users to pass the control or to jump forward to an activity n_{target} which has not yet been activated by the WfMS. When applying such a dynamic jump different policies are offered to users in order to deal with uncompleted activities from the jump region (i.e., uncompleted activities preceding the node n_{target} in the flow of control): $N_{bypass} := \{n \in pred^*(n_{target}) | NS(n) \in \{\texttt{NOT\_ACTIVATED}, \texttt{ACTIVATED}, \texttt{RUNNING}\}\}$.

Activities from this set may be aborted, skipped, or processed anyway depending on what the user desires. In the latter case, their execution must be synchronized with a dynamically specified activity $n_{sync} \in c\_succ^*(n_{target})$ similar to the static case described in Sect. 3.3. In any case the applied transformation steps lead to a proper block structure again in which additional synchronization edges may be used if necessary (see below). Since the structural constraints as well as the graph transformations necessary to realize dynamic forward jumps are very similar to the static case, we omit further details at this point (a more comprehensive treatment can be found in [39]). Instead, in the following subsection we present an example working out important issues related to dynamic forward jumps.

3.4.2 Example

Let us assume that at a certain point in time an instance graph looks like as the one depicted in Fig. 14: A and B are completed and C is currently executed. Let us further assume that an exceptional situation occurs which makes it necessary to immediately perform D (i.e., to jump forward to D) but to maintain the order relationship between all other activities (E after D; G after E and F, etc.).

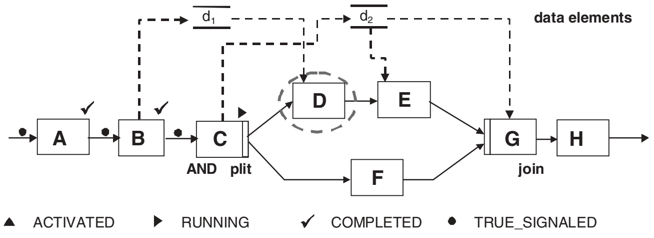


Fig. 14. WF instance graph (for which a user wants to jump forward to D)

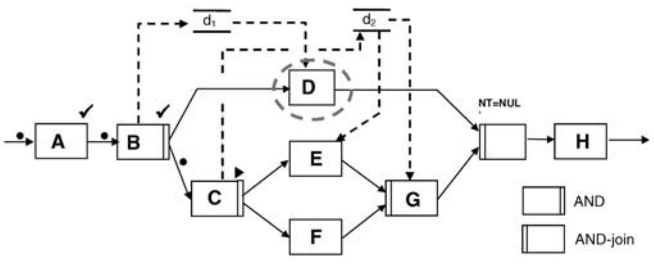


Fig. 15. Dynamic forward jump – intermediate WF instance graph

At first, data dependencies for activity D are checked: D is executable, in principle, because it receives its input data from the already completed activity B and it does not produce any output data such that no problem of lost updates occurs. Thus the restructuring of the instance graph can be started. In order to make it possible that D can be immediately activated, D must no longer be a successor of C. Instead, D has to be placed in a branch parallel to E. This means that the control edge from C to D has to be removed and replaced by a control edge from B to D.

Figure 15 illustrates how the (intermediate) WF instance graph looks like at this point in time. D has become a parallel step with respect to C, its direct predecessor now is B. This transformation alone would not be correct, however, because not all of the previously existing constraints are obeyed any longer. It would be possible, for example, that E is being started (once C has been completed) in parallel or even before D. To enforce the correct execution sequence, a synchronization edge from D to E is introduced which enforces that E cannot be started until D has been completed (cf. Fig. 15). Finally, the WF state is reevaluated. In particular, the control edge from B to D is marked with TRUE_SIGNALED which means that D can be immediately executed. The final WF instance graph is depicted in Fig. 16.

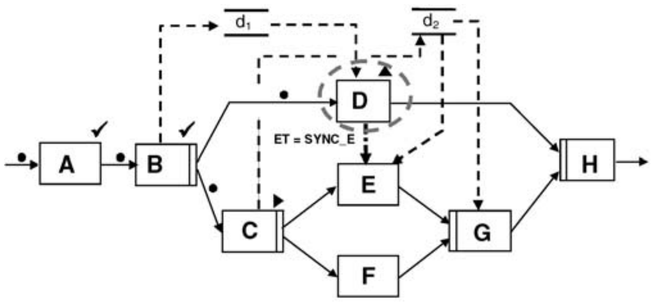


Fig. 16. Dynamic forward jump – final WF instance graph

It is important to mention that, in general, the applicability of a dynamic change depends on the current state of the WF instance graph as well. Concerning a dynamic forward jump, we require that its target activity has not yet been activated; otherwise the application of this operation would not make sense. With respect to other change operations, the required state constraints may be more or less complex, depending on the kind of change. For example, a new activity must not be inserted as a predecessor of an already running or completed step and a completed step must not be deleted. This would cause an inconsistent marking and therefore an undefined execution behavior of the WF instance graph. It is therefore prohibited in ADEPT. Generally, for each basic change operation ADEPT defines formal and easy to check compliance rules with (complexity $O(n)$) to decide whether the intended change can be applied in the current WF state or not.

Also very challenging is the question how to adapt the markings of a WF instance graph when a dynamic forward jump or, more general, a dynamic change is applied. ADEPT uses a sophisticated approach for setting markings of activity nodes and edges, and for adapting them in a consistent manner when a dynamic change is applied (an algorithm for this with complexity $O(n)$ is described in [43]). The corresponding rules are independent from the kind of change, which enables the WfMS to efficiently and automatically adapt markings even in case of complex changes. Taking our example from Figs. 15 and 16, for instance, the newly inserted control edge `B` $\rightarrow$ `D` is evaluated to `TRUE_SIGNALED` since `B` has already marked as `COMPLETED`. This, in turn, leads to re-evaluation of activity `D` of which the marking is set to `ACTIVATED`, such that users can now work on `D`. Note that this is exactly the desired result. Generally, when introducing a dynamic change, already activated activities may have to be deactivated or vice versa. In doing so, ADEPT updates user worklists accordingly.

3.5 Discussion

We have discussed general issues related to the support of pre-planned as well as dynamic forward jumps. In particular, we have shown how corresponding facilities can be offered to users. Our most important goal was to ease the use of corresponding facilities and to hide the complexity associated with them from users. By allowing designers to describe deviations already at buildtime, the flexibility of WF-based applications can be significantly increased. In order to enable users to adequately deal with unforeseeable events, in addition, ADEPT provides support for dynamic changes and dynamic forward jumps. Corresponding changes will be only allowed if they do not violate the correctness properties set out by the ADEPT base model. Furthermore, in the implemented ADEPT WfMS all changes and deviations are properly integrated with respect to authorization and documentation.

4 Backward jumps

In practice, it is very important to allow authorized users to perform backward jumps to former execution states if need be. In ADEPT, WF designers can model normal loop backs as well as failure backward jumps (called *backward jumps* for short). While loop backs specify iterative executions, backward jumps can be used to partially roll back the flow as response to semantic failures (cf. Example 2). More precisely, a rollback operation (partially) resets the effects of previous activity executions. Among other things this includes the resetting of markings, the undoing of write operations on data elements, and the compensation of external effects if possible and reasonable (e.g., by invoking compensating activities). ADEPT allows the modeling of different kinds of backward jumps. Additionally, to deal with unforeseen situations that cannot be pre-planned at buildtime, it enables users to perform ad-hoc backward jumps.

Example 2. (Semantic Failure of an Activity Execution): We refer to Example 1. Regarding the described workflow, a medical examination may fail due to several reasons. For instance, it may not be possible to examine the patient if preparatory measures have been omitted or the patient does not agree with the intervention. Depending on the concrete reason of a semantic failure, the actions necessary for exception handling may vary. For example, the workflow may have to be aborted or it may be sufficient to suspend its execution, to roll back the flow to a former state, and then to resume work in this state.

4.1 Semantic failures and failure backward edges

For handling (semantic) activity failures (cf. Example 2) and for the modeling of related backward jumps, we introduce two additional concepts: *failure codes* and *failure backward edges*.

To each activity the WF designer may assign an arbitrary number of *failure codes* provided that corresponding semantic failures are known at buildtime. In our example, activity "perform examination" may be associated with the two failure codes `OMITTED_PREPARATIONS` and `MISSING_AGREEMENT`. Generally, a failure code corresponds to a semantic failure that may occur during activity execution and that makes it impossible to successfully complete this activity. The task of the WF designer is to identify these semantic failures and to define adequate exception handling actions for them. Possible reactions supported by ADEPT include the repetitive execution of failed activities, the execution of alternative steps, the skipping of the failed activity, the (partial) rollback of the flow, or its controlled abortion. In the following we concentrate on issues related to partial rollback.

For the definition of rollback operations, failure backward edges (called *failure edges* for short) are offered. A

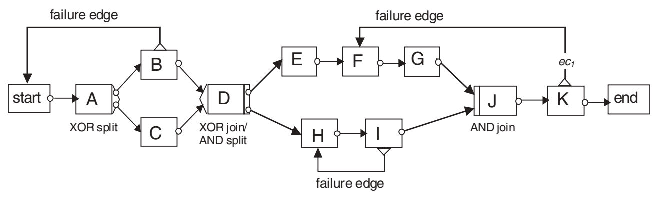


Fig. 17. Modeling backward jumps with failure edges

failure edge $f = n_{fail} \rightarrow n_{bwd}$ links two activities n_{bwd} and n_{fail} together (in backward direction) and is associated with a failure code ec of the source activity n_{fail}. If this activity fails and returns ec as failure code, the processing of the WF instance will be (partially) suspended, the failure edge f be signaled, and the flow be rolled back to the state that had been valid before the execution of n_{bwd} was started. In doing so, the scope of the rollback is limited to activities from the *backward region* which corresponds to that subgraph induced by the nodes from the set N_{bwd} with $N_{bwd} := (c\_succ^*(n_{bwd}) \cap c\_pred^*(n_{fail})) \cup \{n_{bwd}, n_{fail}\}$.

Examples for the use of failure edges are depicted in Fig. 17:

- I $\rightarrow$ H describes a partial rollback whose scope is restricted to a branch of the parallel branching (D, I); i.e., $N_{bwd} = \{$I, H$\}$. The signaling of I $\rightarrow$ H does not affect the processing of other branches.
- K $\rightarrow$ F defines a partial rollback from an activity outside a parallel branching (K in our example) back to activity F, which is located within a branch of this branching ($N_{bwd} = \{$F, G, J, K$\}$). If K $\rightarrow$ F is signaled at runtime, activities J, G, and F will be reset and the flow will continue with F. Note that activities from the lower branch (H and I) are not affected by this partial rollback.
- A special semantics is captured by the failure edge B $\rightarrow$ start. It links a failure code of B with a rollback operation to the start activity of the flow ($N_{bwd} = \{$start, A, B$\}$). When signaling this edge the flow will be aborted and its execution be completely rolled back.

How the state markings and data elements of a WF instance graph are reset when a failure edge is signaled is shown in Fig. 18. To ensure a correct WF execution behavior afterwards, the use of a failure edge $n_{fail} \rightarrow n_{bwd}$ must meet the following restrictions:

- The target activity n_{bwd} of the backward jump must precede the source node n_{fail} in the (normal) flow.
- If n_{bwd} is contained within a branch of an XOR-branching n_{fail} must be contained within the same branch. Otherwise the backward jump may refer to an activity that was not previously executed and therefore lead to an undefined state. Formally: Let j be the XOR-join of an XOR-branching with corresponding split node s. Then we require:
  $n_{bwd} \in Z := c\_succ^*(s) \cap c\_pred^*(j) \Rightarrow n_{fail} \in Z$
- If n_{bwd} is contained within a loop body we require that n_{fail} must be contained within the same subgraph. Otherwise it remains unclear to which iteration of the loop the WF is to be rolled back.

These conditions can be easily checked; algorithms with complexity $O(n)$ are presented in [39].

4.2 Automatic backward jumps

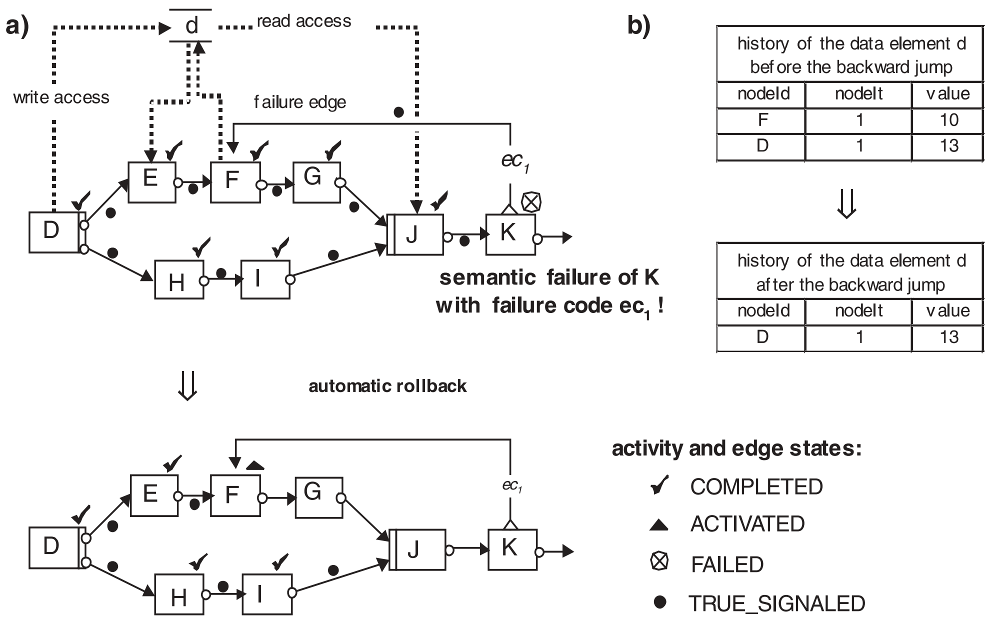
Automatic backward jumps can be linked to semantic failures of WF activities. They can be simply modeled by the use of failure edges and codes. When performing the backward jump at runtime, markings of nodes from the backward region (i.e., nodes from the set N_{bwd}) will be reset. This becomes necessary in order to correctly proceed with the flow afterwards. Furthermore, for all completed or running activities from the backward region, write operations on data elements will be undone. In addition, external effects of these activities (e.g., modifications of application data) may have to be undone as well by invoking corresponding compensation programs. However, whether compensation is possible or not is highly dependend on the respective application. Concerning compensation and data context management, ADEPT follows an approach similar to Sagas [21] and Contracts [42]. In the following, however, we put the focus on modeling aspects; a treatment of transactional properties and other runtime issues is outside the scope of this paper.

An example for a pre-planned, automatic backward jump is depicted in Fig. 18. Let us assume that during the execution of K a semantic failure (with failure code ec_1) occurs. This, in turn, leads to the signaling of the failure edge K $\rightarrow$ F, which causes a partial rollback of the flow. In more detail, activity K is aborted and activities J, G, and F are undone (by invoking associated compensation steps). In doing so, their effects on state markings as well as on data elements are reset. For example, the write operation of activity F on d is undone and the old value of d (written by activity D) is restored. Note that the rollback defined by K $\rightarrow$ F only concerns nodes from the backward region ($N_{bwd} = \{$F, G, J, K$\}$) and therefore does not affect activities from the lower branch of the parallel branching (H and I in our example).

4.3 User-initiated backward jumps

Up to now, we have only dealt with automatic backward jumps which are applied when an activity execution fails. In exceptional situations, for authorized users it must be also possible to directly intervene in the control of a flow by aborting the WF instance or by jumping back to already passed regions. Since corresponding jumps are very often performed as response to external events they can be more or less determined with respect to context and predictability. Depending on this, we have to differentiate between (user) backward jumps, which can be pre-planned and therefore be captured in the WF schema at buildtime, and ad-hoc backward jumps.

Example 3. (User-initiated Backward Jumps): We refer to Example 1. Assume that the patient's state of health gets worse or the physician wants to revise previous decisions concerning patient treatment. In such cases it must be possible for him or her to regain control and to jump back to previously executed steps. (This should at least be possible as long as the medical examination has not taken place.) This, in turn, may require that running activities (e.g., "preparation of the patient") may have to be aborted or completed ones (e.g., "making an appointment") may have to be undone. In many cases, the context for the application of such user initiated backward jumps is



history of the data element d before the backward jump		
nodeId	nodeIt	value
F	1	10
D	1	13

history of the data element d after the backward jump		
nodeId	nodeIt	value
D	1	13

Fig. 18. Partial rollback of a flow due to the signaling of a failure edge. **a** Adaptation of state markings and **b** reset of write operations on data elements

known in advance such that they can be considered at buildtime.

4.3.1 Pre-planned user backward jumps

Similar to shortcuts, ADEPT offers a special edge type to designers – called *regainControl* – for the modeling of user-initiated backward jumps. A regainControl $n_s^{RC} \rightarrow n_d^{RC}$ links an activity n_s^{RC} with a preceding activity $n_d^{RC} \in pred^*(n_s^{RC})$ and has the following semantics: After completing n_d^{RC} and before activating n_s^{RC}, users may initiate a rollback of the flow to the state which had been valid before n_d^{RC} was started. An example is depicted in Fig. 19a). Comparable to the modeling of shortcuts it shows the viewpoint of the WF designer. A backward jump from C to A is defined by the regainControl edge C $\rightarrow$ A. It is selectable after completing A and before activating C. If the backward jump is chosen by a user, the flow will be rolled back to the state that had been valid before A was started. Similar to a shortcut, a regainControl is internally transformed into a representation of the ADEPT base model in order to guarantee a precise and correct execution behavior (cf. Fig. 19b). Since the basic principles of such a transformation have been already discussed in conjunction with shortcuts, we omit a more detailed presentation at this point. The same applies with respect to pre-conditions that must hold for the correct applicability of a regainControl.

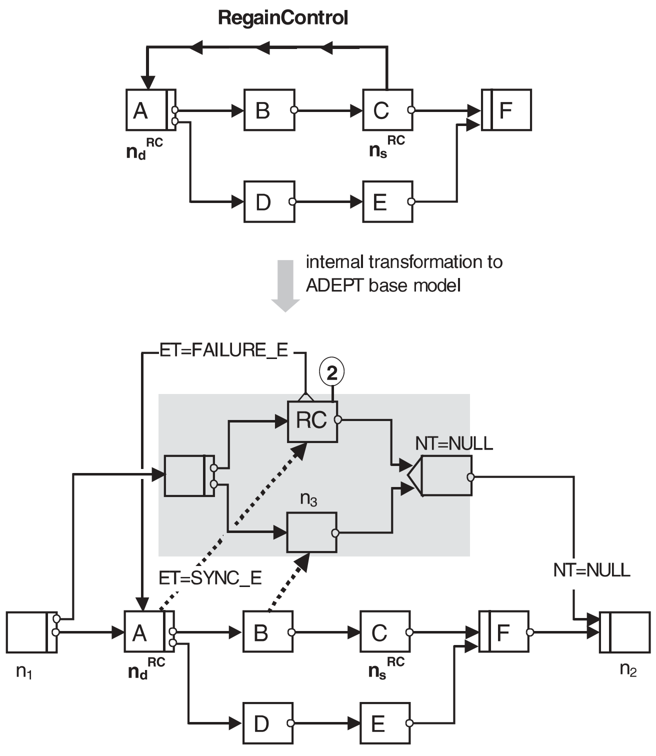


Fig. 19. RegainControl (modeler view) and its internal realization in ADEPT

4.3.2 Ad-hoc backward jumps

Generally, it will not always be possible to foresee and to pre-model all semantic failures that might occur during WF execution. Furthermore there are backward jumps for which a pre-modeling is not possible or is too complex. As an example take a backward jump to an activity that is contained within a branch of an XOR-branching. Since at buildtime it is not known whether the corresponding branch will be selected for execution, the pre-modeling of such a backward jump does not make sense. To adequately deal with such cases as well, ADEPT allows authorized users to perform ad-hoc backward jumps in order to roll back the WF to a previous execution state if need be. Basic to such an ad-hoc deviation (in backward direction) are the WF instance graph, the execution history, and the data element history of the respective WF instance. The execution history, for example, logs data about start and completion of activity executions, which can be used when a rollback has to be performed (cf. Fig. 20). To initiate an ad-hoc backward jump the user may dynamically select a target activity from the list of already completed or running activities. Afterwards the flow can be rolled back to the state that had been valid before this activity instance was started. It is also possible to roll back the WF instance to earlier iterations of a loop, i.e., the scope of the rollback may comprise activity executions from different loop iterations. A simple example illustrating this is depicted in Fig. 20. Note that markings have to be adapted in case of a backward jump as well.

4.4 Discussion

The presented concepts allow us to capture a broad spectrum of pre-planned as well as dynamic backward jumps. Concerning transactional guarantees in combination with backward jumps, usually, for long-running workflows we cannot ensure full ACID properties (and we also do not need this in most cases in practice). We, therefore, have adopted concepts from the field of extended transaction models [21, 42] by allowing the WF designer to (optionally) associate activities with compensation programs, which are then invoked in case these activities have to be undone due to a rollback. Currently, we are working on several other issues arising in this context. They include the handling of backward jumps in modified WF instance graphs (e.g., backward jump to a previously inserted activity), the undoing of temporary changes (e.g. modifications caused by a dynamic forward jump) in connection with rollback operations, and the definition of compensation scopes in order to flexibly adapt the compensation behavior of WF instances depending on their state.

5 Related work

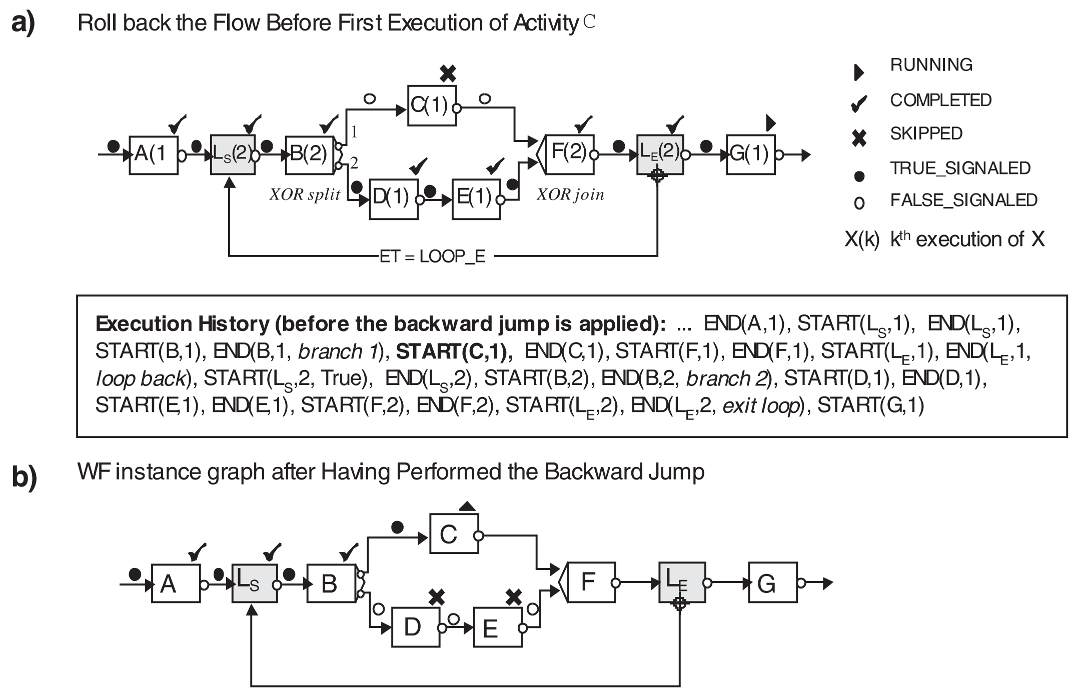
A widespread formalism for WF modeling is offered by Petri Nets [38,58]. Petri Nets have proven themselves as very useful if structural and dynamic properties of WF models have to be analyzed and verified (e.g., liveness, reachability of marking situations, net invariances). Concerning exception handling, many approaches assume that all exceptions which might occur during WF execution are known in advance and can therefore be captured in the Petri Net model [37]. Most of them, however, do not offer special modeling elements for this. In particular, backward jumps as described in this paper have not been addressed at all. Instead, exceptions and deviations have to be described with the same language elements that are used for modeling the "standard" token flow. Very often this leads to complex net structures, which are difficult to understand and to maintain for users [19]. The adaptation of nets is additionally aggravated by the fact that Petri Nets formalisms very often lack a clear separation between control and data flow and do not provide concepts for the explicit modeling of loop backs. Recent work from the field of Petri Nets has picked up this criticism and has offered more flexible concepts for exception handling. In detail, these approaches support the late binding and modeling of WF subnets [24, 25], the dynamic adaptation of net markings during WF execution [1, 20], the ad-hoc migration of a WF instance between two net configurations [3,55], or the dynamic change of the net structure during runtime [18, 19, 56, 57].

Simple, but useful approaches are offered by HOON [25] and MOVE [24]. HOON uses high-level Petri Nets for WF modeling and execution. In particular, late binding of activity components or subnets is supported to increase flexibility. Deviations from the pre-modeled net or dynamic changes, however, are not possible once a subnet is bound to an activity. A more advanced approach is offered by MOVE, which supports late modeling of subnets [24]. At buildtime, the designer may specify activity nodes for which a subnet may be dynamically defined or modified during runtime as long as the corresponding activity is not activated. While this approach is sufficient for supporting dynamically evolving workflow structures, it does not provide the necessary flexibility to deal with exceptional situations. MOBILE [28] follows a descriptive approach for WF modeling. In particular, the WF designer may omit those aspects of a WF model (e.g., the order of tasks), which have to be defined by end users during run-time. Although this approach allows to combine activities in a very flexible way, it is only applicable as long as the activity programs are encapsulated and autonomous such that they may be executed in arbitrary order.

One of the first approaches for the dynamic adaptation of net markings has been offered by MWMS [1,2]. MWMS uses a simple Petri Net formalism. In particular, MWMS allows users to apply backward and forward jumps within a net instance during runtime. Such "roll forwards" and "roll backs" [1] can be applied without violating the consistency of the flow (e.g., no deadlocks). This is achieved, however, by abandoning important elements needed for WF modeling in practice (e.g., no loops, no conditional branchings, no data flow tokens). Though this approach is interesting from a theoretical point of view, due to its restrictive WF meta model it will not work in real-world environments.

One of the first frameworks for dynamic WF changes has come from the Petri Net community [19] as well. For WF modeling so-called flow nets are used. A dynamic change is accomplished by substituting a marked subnet (old change region) by another marked net (new change region). The most interesting issues arising in this context are how to adapt net markings at all and how to accomplish this in a consistent manner; i.e., without causing an undesired dynamic behavior (e.g., deadlocks) in the sequel (cf. Fig. 21). Many of the proposals made in this context assume that users are familiar with Petri Nets or that they are able to manually shift tokens from the old to the new net or to add new tokens, if required [20]. Concerning clinical workflows, for example, such an approach does not work in practice. Apart from this, correctness and consistency checks necessary in this context require expensive reachability analyses for each WF instance. To avoid "dynamic change bugs" [54,57], several proposals have been made: In [54], for example, the authors require that instances of a given net may be only changed if they are not currently executed on modification hit regions. To correctly adapt net markings, in addition, it is proposed to introduce functions that map markings between the old and the new net [55]. Corresponding mapping functions either have to be specified by the designer or are limited to special kinds of changes (i.e., completeness is not ensured). In our experience this is not realistic when looking at real-world processes.

As opposed to these approaches, ADEPT automatically preserves the consistency of markings when a jump operation


a) Roll back the Flow Before First Execution of Activity C
A(1) L$_S$(2) B(2) C(1) D(1) E(1) F(2) L$_E$(2) G(1)
1
2
XOR split
XOR join
ET = LOOP_E
RUNNING
COMPLETED
SKIPPED
TRUE_SIGNALED
FALSE_SIGNALED
X(k) kth execution of X
b) WF instance graph after Having Performed the Backward Jump
A L$_S$ B C D E F L$_E$ G

Execution History (before the backward jump is applied): ... END(A,1), START(L$_S$,1), END(L$_S$,1), START(B,1), END(B,1, *branch 1*), **START(C,1),** END(C,1), START(F,1), END(F,1), START(L$_E$,1), END(L$_E$,1, *loop back*), START(L$_S$,2, True), END(L$_S$,2), START(B,2), END(B,2, *branch 2*), START(D,1), END(D,1), START(E,1), END(E,1), START(F,2), END(F,2), START(L$_E$,2), END(L$_E$,2, *exit loop*), START(G,1)

Fig. 20. Dynamic backward jump and concomitant adaptation of state markings

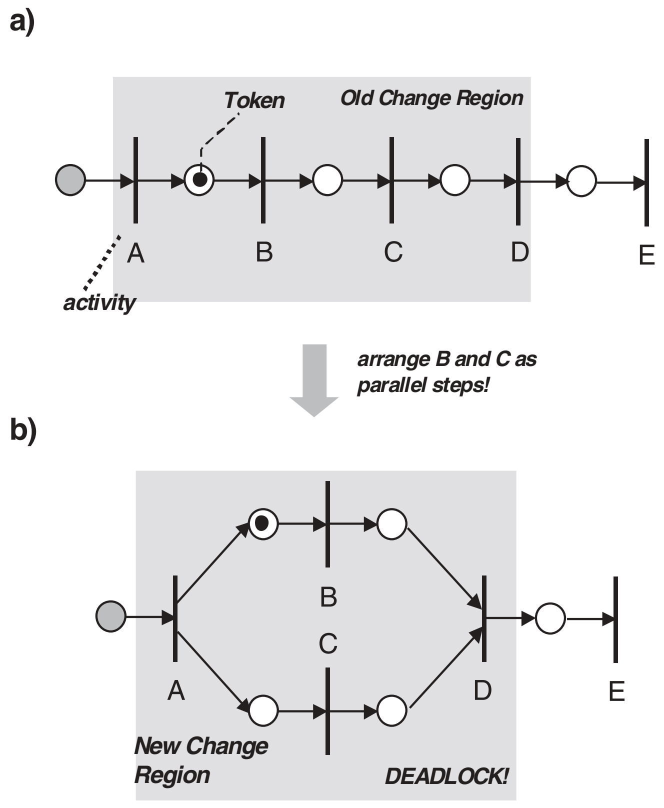


Fig. 21. Dynamic structural changes in Petri nets and the problem of adapting markings

is applied. In particular, node and edge markings are automatically adapted in an efficient and consistent way. Basic to this are the described execution properties of the ADEPT base model and the chosen marking rules. When compared to existing approaches, ADEPT covers a broader spectrum of deviations. In addition, we have dealt with many challenging issues that have not been systematically addressed in the WF literature so far (e.g., concerning the modeling of backward and forward jumps, the correct application of jump operations in connection with loops and conditional branchings, the support of jump operations with different semantics, or the use of automatic jumps).

Several approaches use State- and Activity-Charts for WF modeling [22, 51, 60]. HieraStates [51], for example, tries to increase flexibility by allowing users to dynamically add new states or state transitions during runtime. Furthermore, simple forward jumps can be modeled by the use of a special transition type. As opposed to ADEPT, backward jumps cannot be expressed in a direct way, and dynamic jumps are not supported at all. Another interesting approach has been presented in [22]. It uses Statecharts for WF modeling and deals with issues related to semantic preserving changes. This may be of interest, for example, when more complex modifications become necessary.

Other approaches from the literature combine graphical WF description formalisms with ECA rules in order to increase flexibility [11, 12, 30, 36]. AgentWork [36], for example, uses global rules to enable the WfMS to dynamically restructure the control flow of a WF instance at the occurrence of logical exceptions. The implementation of AgentWork has been partially based on the ADEPT prototype [29, 53]. Hence the two approaches complement one another. In MOKASSIN [30] rules serve as the basis for extending the WF meta model by user-defined constructs, if need be. In addition, WF designers may configure the WF behavior (e.g., with respect to dynamic changes) in a very flexible manner. Although this approach provides a great flexibility for WF designers, it will be not applicable in many application domains, since no correctness or consistency guarantees can be made.

In the WF community, several other work on issues related to dynamic WF changes has been done (e.g., [9,10,17,26,31,35,48,59]). WIDE [10], TRAM [31], WASA$_2$ [59] and BREEZE [48], for example, focus on WF schema evolution and on the migration of in-progress WF instances to the new schema, if possible and desired. In Obligations [9] a WF instance graph is dynamically composed of multiple process templates which reflect different views on the WF instance. Exception handling is possible through the dynamic addition or removal of templates. Correctness issues are not discussed. Finally, in CONSENSUS [4] combined e-negotiations are modeled and enacted using WF technology. To support dynamism, runtime WF modifications are supported. Like AgentWork, CONSENSUS has used the ADEPT WfMS for implementation purposes.

Finally, several other proposals for the support of adaptive workflows have been made in the literature. They are based on different formalisms, like graph grammars [9,27,44], process grammars [23], planning techniques [7], inheritance mechanisms [56], or transactional models [17,21,34].

6 Summary

We have presented a sophisticated approach for the flexible support of pre-planned as well as dynamic deviations in WfMS. For WF designers, high-level concepts are offered for the modeling of forward and backward jumps. By transforming them into an executable representation of the ADEPT base model, a correct and efficient execution by the ADEPT WF engine can be ensured. Furthermore, the separation of standard process descriptions from exceptional paths contributes to a better structuring of the WF models. We have shown that a flexible execution behavior can be achieved when capturing deviations in WF models at buildtime provided that they are known in advance. To increase WfMS flexibility at runtime and to enable users to deal with unforeseen exceptions, in addition, we offer high-level operations to dynamically intervene into the control of in-progress workflows. Authorized users may work on activities ahead of the normal schedule by jumping forward to them or they may roll back the flow to a former execution state by applying a corresponding backward jump. In any case, ADEPT ensures that the correctness and consistency of the flow is preserved when applying such a jump and that the complexity arising from the support of dynamic changes is hidden to a large degree from users.

In this paper we have concentrated on issues related to forward and backward jumps in WfMS. How to define other kinds of changes, how to "physically" perform them, how to adapt state markings and user worklists in this context, or how to deal with concurrent change definitions (and with synchronization issues arising in this context) are other important questions addressed by our research as well. Some of them have been already reported in other papers of our group (e.g. [41,43]) and been considered in our prototypical WfMS implementation as well [29]. Indeed, the ADEPT prototype proves that one can really build a flexible and robust WfMS which offers the described functionality within one system.

Acknowledgements. We would like to thank the anonymous reviewers and our colleague Stefanie Rinderle for their valuable suggestions.

References

1. Agostini, A., De Michelis, G.: Simple workflow models. In: Proc. of the Workshop on Workflow-Management: Net-bases Concepts, Models, Techniques, and Tools., Lissabon, Juni 1998, pp. 146–163
2. Agostini, A., De Michelis, G.: Improving flexibility of workflow management systems. In: Proc. Business Process Management (BPM'2000), LNCS 1806, Lissabon, June 2000, pp. 218–234
3. Badouel, E., Oliver, J.: Reconfigurable nets – a class of high level petri nets supporting dynamic changes. In: Proc. of the Workshop on Workflow-Management: Net-bases Concepts, Models, Techniques, and Tools, Lissabon, Juni 1998, pp. 129–145
4. Bassil, S., Benyoucef, M., Keller, R., Kropf, P.: Addressing dynamism in e-negotiations by workflow management systems. In: Proc. DEXA Workshop, September 2002
5. Bauer, T., Dadam, P.: Efficient distributed workflow management based on variable server assignments. In: Proc. CAiSE '00, Stockholm, June 2000, pp. 94–109
6. Bauer, T., Reichert, M.: Dynamic change of server assignments in distributed workflow management systems. Datenbank Spektrum, 2(4): 59–67, 2002
7. Beckstein, C., Klausner, J.: A planning framework for workflow management. In: Proc. Workshop on Intelligent Workflow and Process Management., Stockholm, August 1999
8. Beuter, T., Dadam, P., Schneider, P.: The WEP model: Adequate workflow-management for engineering processes. In: Proc. Europ. Concurrent Engineering Conf. 1998, Erlangen, April 1998
9. Bogia, D.: Supporting Flexible, Extensible Task Descriptions In and Among Tasks. PhD Thesis, University of Urbana, Illinois, 1995
10. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data and Knowledge Engineering, 24(3): 211–238, 1998
11. Casati, F., Fugini, M., Mirbel, I.: An environment for designing exceptions in workflows. Information Systems, 24(3): 255–273, 1999
12. Chiu, D., Li, Q., Karplapalem, K.: A meta modeling approach to workflow management systems supporting exception handling. Information Systems, 24(2): 159–184, 1999
13. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Werawarana, S.: Business Process Execution Language for Web Services (V 1.0). Initial Public Draft Release, 2002, see `http://www.ibm.com/developerworks/library/ws-bpel`, 2002
14. Dadam, P., Reichert, M., Kuhn, K.: Clinical workflows – the killer application for process-oriented information systems? In: Proc. 4th Int'l Conf. on Business Information Systems (BIS '00), Poznan, Poland, 2000, pp. 36–59
15. Dijkstra, E.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages. Academic Press, 1968
16. Dumas, M., ter Hofstede, A.H.M.: UML activity diagrams as a workflow specification language. In: Proc. Int'l Conf. on the Unified Modeling Language, Toronto, 2001
17. Eder, J., Liebhart, W.: Contributions to exception handling in workflow-management. In: Proc. EDBT Workshop on Workflow Management Systems, Valencia, 1998, pp. 3–10
18. Ellis, C.A., Keddara, K.: A workflow change is a workflow. In: Proc. Business Process Management (BPM'2000), LNCS 1806, 2000, pp. 201–217
19. Ellis, C.A., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. Int'l Conf. on Org. Comp. Sys., Milpitas, CA, August 1995, pp. 10–21
20. Ellis, C.A., Maltzahn, C.: The Chautauqua workflow system. In: Proc. 30th Int'l Conf. on System Science, Maui, 1997

21. Elmargarmid, A.K. (ed.): Database Transaction Models for Advanced Applications. Morgan Kaufmann Publ., 1992
22. Frank, H., Eder, J.: Equivalence transformations on statecharts. In: Proc. 12th Int'l Conf. on Software and Knowledge Eng., Chicago, July 2000, pp. 150–158
23. Glance, N., Pagani, D., Pareschi, R.: Generalized process structure grammars (gpsg) for flexible representations of work. In: Proc. 7th Intl Conf. on Computer Supported Cooperative Work (CSCW96), Boston, MA, 1996, pp. 180–189
24. Hagemeyer, J., Hermann, T., Just-Hahn, K., Striemer, B.: Flexibility in workflow management systems. In: Proc. Software-Ergonomie'97, Dresden, Germany, 1997, pp. 179–190 (in German)
25. Han, Y.: Software Infrastructure for Configurable Workflow Systems. PhD Thesis, TU Berlin, 1997
26. Han, Y., Sheth, A.: On adaptive workflow modeling. In: Proc. 4th Int'l Conf. on Information System Analysis and Synthesis, Orlando, 1998
27. Heimann, P., Joeris, G., Krapp, C., Westfechtel, B.: DYNAMITE: Dynamic task nets for software process management. In: Proc. 18th Int'l Conf. Software Engineering, Berlin, 1996, p. 331341
28. Heinl, P., Schuster, H., Stein, K.: Support of ad-hoc workflows in the mobile workflow model. In: Proc. Softwaretechnik in Automation und Kommunikation, Munich, March 1996, pp. 229–242
29. Hensinger, C., Reichert, M., Bauer, T., Strzeletz, T., Dadam, P.: ADEPT$_{workflow}$ – advanced workflow technology for the efficient support of adaptive, enterprise-wide processes. In: Proc. Software Demonstration Track (EDBT '00), Konstanz, March 2000
30. Joeris, G.: Defining flexible workflow execution behaviors. In: Proc. GI-Workshop, Enterprise-wide and Cross-enterprise Workflow-Management (Informatik '99), October 1999, pp. 49–55
31. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Proc. CoopIS '99, Edinburgh, September 1999, pp. 104–114
32. Lei, Y., Singh, M.P.: A comparison of workflow metamodels. In: Proc. ER'97 Workshop on Behavioral Models and Design Transformations, Los Angeles, 1997
33. Leymann, F., Roller, D.: Production Workflow. Prentice Hall, 2000
34. Liu, L., Pu, C.: Methodical restructuring of complex workflow activities. In: Proc. 14th Int'l Conf. On Data Engineering (ICDE98), Orlando, Florida, 1998, pp. 342–350
35. Luo, Z., Sheth, A.: Defeasible workflow, its computation and exception handling. In: Proc. CSCW'98 Workshop Towards Adaptive Workflow Systems, Seattle, WA, November 1998
36. Müller, R., Rahm, E.: Dealing with logical failures for collaborating workflows. In: Proc. Int'l 5th Conf. on Cooperative Information Systems, Eilat, 2000, pp. 210–223
37. Oberweis, A.: Specification of techniques for handling exceptions with petri nets. Automatisierungstechnik – at, 40(1): 21–30, 1992 (in German)
38. Oberweis, A.: Modeling and Execution of Workflows Based on Petri Nets. Teubner, 1996 (in German)
39. Reichert, M.: Dynamic Changes in Workflow-Management-Systemen. PhD thesis, University of Ulm, 2000 (in German)
40. Reichert, M., Bauer, T., Fries, T., Dadam, P.: On modeling predictable deviations in workflow management systems. In: Proc. Modellierung '02, pp. 183–194, Tutzing, March 2002 (in German)
41. Reichert, M., Dadam, P.: ADEPT$_{flex}$ – supporting dynamic changes of workflows without losing control. JIIS, Special Issue on Workflow Management Systems, 10(2): 93–129, 1998
42. Reuter, A., Schwenkreis, F.: Contracts – a low-level mechanism for building general-purpose workflow management systems. IEEE Bulletin of the Technical Committee on Data Engineering, 18: 4–10, 1995
43. Rinderle, S., Reichert, M., Dadam, P.: Efficient compliance checks and automatic migration of workflow instances to support workflow schema evolution. Informatik, Forschung und Entwicklung, 17(4): 177–197, 2002 (in German)
44. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation – Volume 1: Foundations. 1997
45. Saastamoinen, H.: On the Handling of Exceptions in Information Systems. PhD thesis, University of Jyvaskyla, Finland, 1995
46. Sadiq, S.: On Verification Issues on Conceptual Modeling of Workflow Processes. PhD thesis, University of Queensland, Australia, 2001
47. Sadiq, S., Orlowska, M.: Analyzing process models using graph reduction techniques. Information Systems, 25(2): 117–134, 2000
48. Sadiq, S., Orlowska, M.: On capturing exceptions in workflow process models. In: Proc. 4th Int'l Conf. on Business Information Systems (BIS '00), Poznan, Poland, April 2000
49. Schultheiss, B., Meyer, J., Mangold, R., Zemmler, T., Reichert, M.: Process design for therapeutic treatment of inpatients. DBIS-5, University of Ulm, November 1995 (in German)
50. Strong, D., Miller, S.: Exceptions and exception handling in computerized information processes. ACM ToIS, 13(2): 206–233, 1995
51. Teege, G.: Flexible workflows. In: Proc. Workshop Flexibilität und Kooperation in Workflow-Management-Systemen, p. 1321, 1998 (in German)
52. ter Hofstede, A.H.M., Orlowska, M., Rajapakse, J.: Verification problems in conceptual workflow specifications. Data & Knowlege Engineering, 24(3): 239–256, 1998
53. Rahm, E., Greiner, U.: Webflow: A system for the flexible execution of web-based, cooperative workflows. In: Proc. Database Systems For Business, Technology and Web (BTW'2003), Leipzig, February 2003 (in German)
54. van der Aalst, W.M.P.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers, 3(3): 297–317, 2001
55. van der Aalst, W.M.P.: How to handle dynamic change and capture management information: An approach based on generic workflow models. Int'l Journal of Computer Systems, Science, and Engineering, 16(5): 295–318, 2001
56. van der Aalst, W.M.P., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoretical Computer Science, 270(1–2): 125–203, 2002
57. van der Aalst, W.M.P., Jablonski, S.: Dealing with workflow change: Identification of issues an solutions. Int'l Journal of Comp. Systems, Science and Engineering, 15(5): 267–276, 2000
58. van der Aalst, W.M.P., van Hee, K.: Workflow Management. MIT Press, 2002
59. Weske, M.: Flexible modeling and execution of workflow activities. In: Proc. 31st Int'l Conf. on System Sciences, Hawaii, 1998, pp. 713–722
60. Wodtke, D., Weikum, G.: A formal foundation for distributed workflow execution based on state charts. In: Proc. Int'l Conf. on Database Theory (ICDT'97), Delphi, Greece, 1997

Manfred Reichert is assistant professor in the Department Databases and Information Systems at the University of Ulm, Germany. He finished his PhD thesis on flexible workflow management in May 2000. He is author and co-author of many articles and conference papers on hospital information systems and workflow management. Current research topics include enterprise-wide and cross-organizational workflows, enterprise application integration and workflow, and different aspects related to workflow technology.

Peter Dadam has been full professor at the University of Ulm and director of the Department Databases and Information Systems since 1990. Before he came to the University he had been director of the research department for Advanced Information Management (AIM) at the IBM Heidelberg Science Center (HDSC). At the HDSC he managed the AIM-P project on advanced database technology and applications. Current research areas include distributed, cooperative information systems, workflow management, and database technology and its use in advanced application areas.

Thomas Bauer is a researcher at the DaimlerChrysler Research Centre in Ulm where he is working in the area of engineering process management. Before, he was a member of the Department Databases and Information Systems at the University of Ulm where he finished his PhD thesis on the efficient management of enterprise-wide workflows in 2001. Current research areas include engineering processes, workflow management, and enterprise application integration.