ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Loosing Control

Manfred Reichert
Peter Dadam
University of Ulm

Technical Report No. 97-07
Ulmer Informatik-Berichte
April 1997

ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Loosing Control

Manfred Reichert, Peter Dadam

University of Ulm

Abteilung Datenbanken und Informationssysteme

D-89069 Ulm, Germany

e-mail: {reichert,dadam}@informatik.uni-ulm.de

Abstract

Current process-oriented workflow technology is only applicable in a secure and reliable manner if the business process (BP) to be supported is well-structured and there is no need for dynamic extensions or ad hoc deviations at runtime. As only few BPs are statically in this sense, this limits the applicability of today's workflow management systems (WFMSs) significantly. On the other hand, to support deviations from the premodelled task sequences at runtime should not mean that the responsibility for the avoidance of consistency problems (e.g., unintended lost updates) or even runtime errors (e.g., program crashes due to the invocation of task modules with invalid or missing parameters) is now completely shifted to the (naive) end user. Instead, a formal foundation must be established that allows the runtime system to decide whether an intended deviation can be handled in a proper and secure manner.

In this paper we present a conceptual and operational framework for the support of dynamic structural changes of workflows (WFs) in WFMSs. Based upon a simple WF model (ADEPT) we define a complete and minimal set of change operations (ADEPT$_{flex}$) that support users in modifying the structure of WFs at runtime, while maintaining their correctness and consistency. Correctness properties defined by the ADEPT model are used to determine whether a specific change can be applied to a given WF or not. If these properties are violated the change is either rejected or the correctness must be restored by handling the exceptions resulting from the change. Further, we discuss basic issues with respect to the management of changes and the undoing of temporary changes when backward operations are applied. The change facilities presented in this paper will form a key part of process flexibility in future WFMSs.

Keywords: Flexible WFMSs, Dynamic Change, Adaptive Workflows, Ad hoc Workflows, Exception Handling

1 Introduction

Process-oriented workflow management systems (WFMSs) [GHS95, Hsu95, LeAl94] offer a promising approach for the development of business applications which directly follow the execution logic of the underlying business process (BP). The separation of the application's control structures from the implementation of its task programs contributes to simplify and to fasten application development and enables the runtime system to assist users in coordinating and scheduling the tasks of a BP.

Current process-oriented WFMSs, however, are only applicable in a reliable and secure manner if the BP to be supported is well-structured and there is no need for ad hoc deviations or dynamic extensions at runtime [BaWa95, BlNu95, EKR95, ElNu93, Sie96]. As only few BPs are statically in this sense, this limits the benefit and the applicability of workflow (WF) technology significantly. As an example, consider BPs from the clinical domain [e.g., Mey96], where it is often not convenient and cost effective to capture all possible task sequences in advance. There are several reasons for this: firstly, there are many WFs whose planning and execution overlap (*dynamically evolving WF*) or which are completely specified at runtime (*ad hoc WF*). Secondly, unplanned events and exceptions frequently occur leading to *ad hoc deviations from the preplanned WFs*. Exceptions cover cases such as requests to deviate from standard processes due to an external event (e.g., in case of an acute emergency), failed tasks (e.g., when prerequisites for a medical intervention are violated), incomplete or erroneous information in task inputs and outputs (e.g., incomplete medical orders), or situations that arise from mismatches between the real processes within the organization and their computerized counterparts (e.g., due to incomplete or faulty WF specifications or due to organizational changes) [StMi95, Mey96]. Since WF designers are generally not capable to predict all possible exceptions and events beforehand and to capture them in the design of a WF [StMi95], the runtime system does not always have sufficient knowledge to handle these situations alone. Instead human involvement with the runtime system becomes necessary to resolve exceptions resp. to deal with unplanned events. Hence the resulting requirements are far more challenging than those faced by standard transaction technology and current advanced transaction models [WoSh97, Elm92].

A basic step towards more flexibility is the effective and efficient support of ad hoc modifications and well-aimed extensions of processes during their execution. So a WFMS must provide functions for adding or deleting tasks resp. task blocks and for changing predefined task sequences, e.g., by allowing users to skip tasks with or without finishing them later, to work on tasks although the conditions for their execution are not yet completely satisfied, or to serialize two tasks that were previously allowed to run in parallel. Ad hoc changes may also concern single attributes of a WF object (e.g., a task). Examples are the reassignement of a task or the modification of a task's deadline. As these changes are less critical to handle than structural changes, we do not consider them further in this paper.

1.1 Problem Description

To allow users to deviate from premodelled task sequences of a WF at runtime is a two-edge sword. On the one hand, it captures the natural freedom of humans to work on a process and to deal with exceptional situations and unplanned events. On the other hand, unrestricted changes to the structure of a long-running program – possibly in the midst of its execution – makes it difficult to have the system behave in a predictable and reliable manner. For this reason, supporting dynamic structural changes should not mean that the responsibility for the avoidance of *consistency problems* or even *runtime errors* is now completely shifted to the (naive) end user or to the application programmer. Instead, a clear theoretical basis and correctness criteria must be established which enable the runtime system to reason about the correctness of a requested change and to assist users in managing it in a proper and secure manner.

First of all, this requires that all types of *structural dependencies* between tasks (e.g., control, data and temporal dependencies) are taken into consideration when the WF is restructured. Otherwise, changes such as the deletion or the addition of a task, for example, may cause severe inconsistencies (e.g., unintended lost updates) or even program crashes (e.g., invocation of task modules with invalid or missing parameters). Changes must consider the *state* of the WF resp. its tasks, too. For example, users should not be allowed to delete a task or to change its attributes, if it has already been completed. Convenient rules which should not appear as too restrictive to users must be defined in order to avoid an improper and uncontrolled use of change operations. Finally, for *security reasons* it must be possible to restrict the use of change operations to selected users (resp. roles), to specific process types resp. regions of a process graph (e.g., a single task), to certain states of a process, or to any combination of them.

Normally, several instances of a specific process type are active at the same time. As changes of different kinds may be applied to the individual processes during their execution, several issues must be addressed. First of all, process instances of the same type (i.e., the same starting schema) may have to be represented by different *execution graphs*. Secondly the runtime system must manage changes of different nature concerning their durability. This is especially important for long-running processes where applied changes may be permanent or temporary. *Permanent changes* must be preserved until completion of the process. By contrast, *temporary changes* may have to be undone if the control of the process is passed back to a previous point of control (e.g., when a loop enters a new iteration). Consequently a technical challenge is how to represent these different type of changes at the system level and how to undo temporary changes in a proper and secure manner. This requires sophisticated mechanisms for change management and a close integration of change operations with other core services of the WFMS. Finally, requested changes should be made "on the fly" without loss of runtime *performance* and without disturbing process participants not actively involved in the change.

In summary, dynamic structural changes represent serious interventions into the control of a process which cannot be handled without extensive system support. In providing operational support for dynamic changes, whether used by the WF administrator or, in some form, the process participants, it is crucial that these facilities will be manageable and usable in a proper and secure manner.

1.2 Contribution of this Paper

In this paper we present a conceptual and operational framework for the support of dynamic structural changes of WFs in WFMSs. Basic to our approach is a conceptual, graph-based WF model (ADEPT[1]) which has a formal foundation in its syntax and (operational) semantics. Based upon this model we develop a *complete* and *minimal* set of *change operations* which support users in modifying the structure of WFs at runtime, while preserving their *correctness* and *consistency* ($\text{ADEPT}_{\text{flex}}$). If a change leads to the violation of correctness properties, it is either rejected or the correctness of the process graph (e.g., concerning the flow of data) must be restored by handling the exceptions resulting from the change (possibly leading to concomitant changes). We further show how temporary and permanent changes are managed at the system level and which precautions must be made in order to enable the runtime system to undo temporary changes in case of backward operations.

The contribution of this paper is demonstrating the principle feasability of our approach and to give some insights into fundamental research issues related to the dynamic change problem. This includes the following three results:

- we demonstrate the suitability of our WF model for WF specification and for the efficient support of dynamic structural changes
- we show how even complex, dynamic structural changes can be applied to a WF during its execution and which precautions must be made to do this in a secure and reliable manner
- we discuss technical challenges and possible solutions concerning the management of temporary as well as permanent changes

At this point we have a prototype running that supports the basic concepts and the change operations presented in the following. For the rest of the paper we concentrate on ad hoc changes and dynamic extensions applied to individual WF instances at runtime. We do not explicitly consider *changes at the schema level* and their propagation to processes whose execution started with the old schema [cmp. CCP96, EKR95]. However, many of the presented concepts can be applied to this type of change, too.

Section 2 gives an overview of the ADEPT WF model. In *section 3* we present a complete and minimal set of change operations which can be used to modify the structure of a WF during its execution. *Section 4* addresses issues concerning the management of changes and their undoing in case of backward operations. *Section 5* discusses related work. We conclude with a summary, an overview of related issues not addressed within this paper and an outlook on future work in *section 6*.

[1] ADEPT stands for Application Development Based on Encapsulated Premodeled Process Templates

2 Fundamentals of the ADEPT Workflow Model

A variety of WF models and WF description languages have been discussed in the literature [CKO92]. Some of them are based on formal models such as high level petri nets [DGS94, ElNu93, EKR95, KHK91, LeAl94], state- and activitycharts [WoWe97], temporal logics [MaPn92, Att93], or process algebras [Henn89], for example. One strength of these approaches lies in the offered formalisms for specifying, analysing and verifying the properties of static process structures, e.g., regarding state transitions, deadlocks, reachability of tasks resp. states, and so on. Mechanisms for modifying these structures at runtime, however, are missing for the most part [EKR95].

To support dynamic structural changes we plead for the use of a formal model, too. For several reasons, we do not believe that the general-purpose models mentioned above do build the right basis for this. Firstly, their generality makes debugging and testing of large, complex processes extremely costly [HOR96, HsKl96, SGJ96], which may cause a significant overhead when complex structural changes become necessary during runtime. Secondly, to effectively support users – possibly non-computer experts – in performing structural changes at runtime, a WF model must allow an intuitive and structured representation of a BP, which is hardly to achieve with these models.

The ADEPT workflow model presented in this section follows a more structured approach. Essential for the specification and the execution of WFs is the concept of *symmetrical control structures*, which is well-known from structured programming languages [cmp. Rein93]: task sequences, branchings (with different execution semantics) and loop backs are specified as "symmetrical blocks" with well-defined start and end nodes. These blocks may be arbitrarily nested, but are not allowed to overlap (i.e., the nesting must be regular). In addition, ADEPT provides support for the synchronization of tasks from parallel branches. The use of symmetrical control structures provides the basis for the syntax-driven design of a BP [Kir96] and for the efficient analysis of its flow of control and data [Hen97]. The latter aspect is crucial for the support of complex ad hoc changes during runtime. So ADEPT offers a good compromise for the trade-off between the expressive power of a WF model and the complexity of model checking. A detailed description of the ADEPT model is beyond the scope of this paper. Due to lack of space we restrict our considerations to the concepts and correctness properties provided for the specification of the control and data flow of a process. Other important aspects, e.g., the modeling of temporal and organizational aspects and their interplay with dynamic changes are described in [Gri97, Hen97, Kir96, Mey96, Rei97].

2.1 Workflow Modeling

In this section we (informally) introduce the basic concepts offered by ADEPT for the modeling of a *WF schema*. Such a schema comprises a set of *tasks* and *control* resp. *data dependencies* between them. We restrict our considerations to *simple tasks*, i.e., activities which cannot be further divided and whose execution is requested by external (not necessarily human) agents.

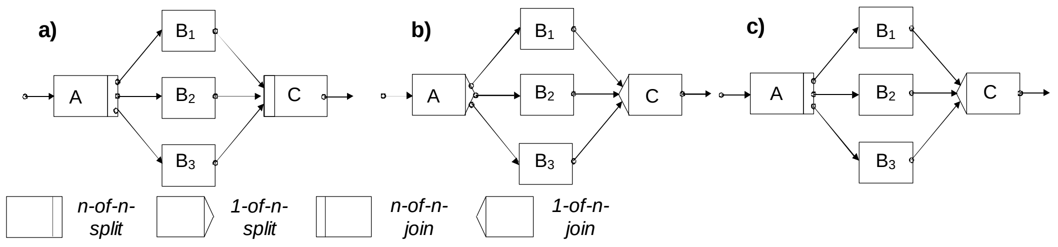


Figure 1: *Branchings in ADEPT*: the execution semantics of the branching shown in c) is as follows: after the completion of A its outgoing edges are signaled. This, in turn, triggers the execution of B_1, B_2 and B_3. As soon as one of these tasks completes, the others are removed from the corresponding worklists, aborted or undone and C is triggered.

Flow of Control

First of all, we represent a WF schema as as a directed, structured graph (N, E), where tasks are abstracted as a set of nodes N (of different types NT) and control dependencies between them as a set of directed edges E (of different types ET). The use of nodes and edges has to meet the restrictions defined by the control structures presented in the following.

Each WF schema has a unique *start node* (NT = STARTFLOW) and a unique *end node* (NT = ENDFLOW). The start node has no predecessor and the end node no successor. All other nodes from N must be preceded and succeded by at least one node in the graph.

The *sequential execution* of two tasks is modelled by connecting them with a control edge (ET = CONTROL_E). The modeling of *branches* is depicted in figure 1. Branches start with a split node and are synchronized symmetrically at a unique join node. ADEPT supports three types of branching: *parallel processing* (n-of-n split / n-of-n join), *conditional routing* (1-of-n split / 1-of-n-join) and *parallel branching with final selection* (n-of-n-split / 1-of-n-join)[2]. The routing decision of a *conditional branching* may either be value-based[3] or is made by users. In the latter case all successors of the split node are triggered when it fires. As soon as one of them is selected for execution, however, the work items of the others are removed from the corresponding worklists. This allows us to model situations, where multiple tasks are activiated, but only one of them may be executed. *Parallel branchings with final selection* have the following execution semantics: when the split node fires, all successor branches are triggered and can be worked on concurrently. In contrast to parallel processings, the flow may proceed at the join node as soon as one of its predecessors terminates. Depending on their current state the tasks of the other branches are then removed from the corresponding worklists, aborted or undone. Undoing a branch does not necessarily lead to the execution of compensation tasks [cmp. Kir96]. In any case, the corresponding tasks are reset in their state (cmp. section 2.2) and their effects on data

[2] In this paper we omit m-of-n-splits found in many WF models. The necessary extensions are described in [Rei97].

[3] The branch to be selected is determined by the value of a data element (decision parameter DP) which must be written by a preceding task. The outgoing edges of the split node S are associated with different selection codes (SC). When S fires, the edge whose SC corresponds to the value of DP is signaled as TRUE, while the other edges are signaled as FALSE. In order to guarantee the progress of the flow, a default edge must be specified which is signaled if none of the specified SCs corresponds to DP.

elements (e.g., write operations) of the WF are undone. By the use of this type of branching, for instance, the WF designer is able to premodel "shortcuts" in the flow of control. When a user follows such a shortcut during runtime, the system deals with skipped steps. As an important extension we also allow that first of all more than one branch may complete. In this case the "winner" branch must be selected by an authorized user before the flow can proceed.

Up to now we have only considered acyclic process graphs. The repetitive execution of a set of tasks can be modelled by the use of *loops*. Like branchings a loop corresponds to a symmetrical block with a unique start node (NT = STARTLOOP) and a unique end node (NT = ENDLOOP), which are connected by a loop edge (ET = LOOP_E). In addition, the end node is associated with a loop condition, which is evaluated each time the end node is triggered. As we will see in section 4, the use of loops raises some challenging issues with respect to exception handling and dynamic structural changes. For example, when a task is inserted into the body of a loop, it must be clear whether this insertion is only valid for the current iteration (temporary change) or for following iterations, too.

To handle task failures at the modeling level, ADEPT provides a second type of backward edge (ET = FAILURE_E): a *failure edge* f links a task $n_{failure}$ with a preceding node $n_{restart.}$ If the execution of the task $n_{failure}$ fails, the edge f is signaled. As a consequence, all nodes succeeding $n_{restart}$ (incl. $n_{restart}$) and preceding $n_{failure}$ (incl. $n_{failure}$) in the flow of control are reset to their initial state. In contrast to loop edges, the effects of the corresponding tasks on data elements of the WF (see below) are undone. Afterwards the flow proceeds with the execution of $n_{restart.}$ The symmetrical structuring and the regular nesting of control structures do not apply to failure edges as a task may have multiple outgoing failure edges, possibly linking it with nodes from different branches of a parallel branching. To achieve a well-defined semantics for the use of failure edges, we further require that if $n_{restart}$ is contained within the body of a loop (resp. within a branch of a branching with 1-of-n join) the node $n_{failure}$ must be contained within the same loop body (resp. branch). As the use of failure edges is therefore not always possible in conjunction with these control structures, we support the dynamic rollback of a WF, too. Generally, a process resp. a process region may be reset to the state it had before a certain task entered its i.th iteration [cmp. Hen97].

The expressive power of the control structures presented so far is not sufficient for the modeling of WFs with long-running, concurrent executions. To be able to synchronize tasks from different branches of a parallel processing two types of *synchronization edges* (sync edges) are supported:

- A "soft" synchronization $n_1 \rightarrow_{SOFT} n_2$ (ET = SOFT_SYNC_E) can be used to specifiy a *delay dependency* between the two tasks n_1 and n_2: n_2 may only be executed if n_1 is either completed or if it cannot be triggered anymore. This type of synchronization does therefore not necessarily require the successful completion of n_1. An example is given in figure 2, where H is triggered when G is completed and E is either completed or skipped (i.e., the corresponding branch is not selected for execution).

- On the other hand, a "strict" synchronization $n_1 \rightarrow_{STRICT} n_2$ (ET = STRICT_SYNC_E) between n_1 and n_2 requires that n_1 must be successfully completed before n_2 is allowed to start. A strict synchronization may be used to synchronize tasks from branches of a parallel branching with final selection with tasks from branches of a conditional routing.[4]

The use of sync edges has to meet certain constraints in order to avoid redundant control dependencies between tasks, cycles, or even termination problems of the WF. In any case, only nodes from different branches of a parallel processing (with n-of-n join) may be synchronized by the use of sync edges. Furthermore, a sync edge may not connect a node from inside a loop body B with a node not contained within B. A detailed presentation of these constraints and algorithms for their validation can be found in [Rei97].

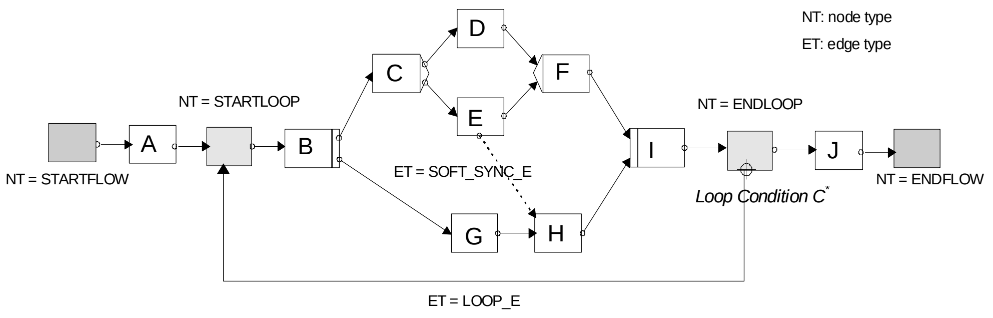


Figure 2: Workflows are modelled by the use of symmetrical control structures. In addition, the execution of tasks from different branches can be synchronized by the use of sync edges.

Flow of Data

The input and output data of a WF schema resp. its tasks and the flow of data between tasks are an important functional view of the system. Nevertheless, the modeling of the data flow and the exchange of data between the tasks of a WF are often poorly supported in WFMSs [SGJ96]. This does not only leave significant complexity to application developers, but it also makes it impossible to provide system support for verifying the correctness of the data flow resp. for adjusting it when dynamic structural changes are applied to a WF. In ADEPT the exchange of data between tasks is based on global WF variables. A WF schema P is associated with a set of data elements D, where each element $d \in D$ has a unique identifier id^d and a domain dom^d. The *data flow* between tasks is defined by connecting their input resp. output parameters with elements from D. For simplification, the input (resp. output) data of P are logically treated as the output (resp. input) data of its start (resp. end) node.

In practice, there are often great differences in the format and in the representation of data which is the output of one task and the input to another. To relieve task programmers from performing the necessary adjustments within task implementations, each task $n \in N$ can be associated with a set of auxiliary services S_n. The execution of these services is closely

[4] As an example assume that n_1 belongs to a branch B of a conditional routing and n_2 to a branch of a parallel branching with final selection. Then n_2 won't be triggered, if B is not selected for execution.

connected to the execution of the task. An auxiliary service $s \in S_n := S_n^{prec} \cup S_n^{succ}$ is either triggered when n is started ($s \in S_n^{prec}$) or when it is terminated ($s \in S_n^{succ}$) and does therefore not appear as a separate work item in any worklist. Services from the set S_n^{prec} may also be used to request input data for a task from the user initiating it, which has turned out to be quite important in our context (see section 3). Furthermore, a task n (resp. the program associated with it) is only allowed to be executed after all services from S_n^{prec} have been successfully completed. On the other hand, if a task fails resp. is undone the effects of its associated services on data elements are undone, too. Formally:

Definition 1 (Data Flow Schema): Let (N,E) be the control flow graph of a WF schema P and let D denote the finite set of data elements associated with P. Let further Pars(X): = InPars(X) $\cup$ OutPars(X) denote the set of input and output parameters associated with the task resp. the service X.

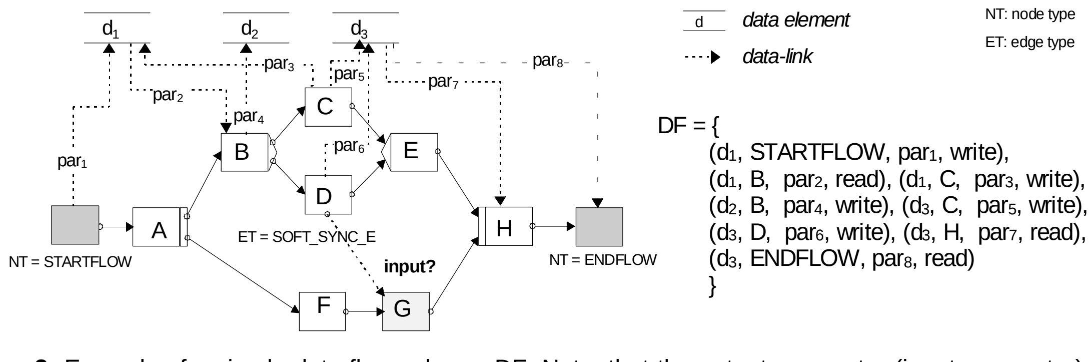
A *data link* df between a parameter par^{df} and a data element d^{df} is described by the 4-tuple:

$$\mathbf{df = (d^{df}, n^{df}, par^{df}, access\_mode^{df})} \text{ with}$$

$$\mathbf{d^{df} \in D, n^{df} \in N \cup S\ (S := \bigcup_{n \in N} S_n),\ par^{df} \in Pars(n^{df}),\ access\_mode^{df} \in \{read, write\}}$$

The set of all data links DF, connecting task resp. service parameters with global data elements, is called the *data flow schema* of P.

Note that the read resp. write access of services from S form a part of the data flow schema of P. The intuitive meaning of a data link (d, n, p, read) $\in$ DF is that the value of the input parameter p $\in$ InPars(n) is read from the data element d when n is started. In contrast, the link (d, n, p, write) indicates that the value of the output parameter p $\in$ OutPars(n) is written into d after the (successful) completion of n. An example for a simple data flow schema is depicted in figure 3. In section 2.3 we indroduce a set of properties defining a valid data flow schema DF. These properties constitute the basis for detecting possible exceptions resulting from a change and for adjusting the data flow schema when the WF is restructured.



DF = {
(d_1, STARTFLOW, par_1, write),
(d_1, B, par_2, read), (d_1, C, par_3, write),
(d_2, B, par_4, write), (d_3, C, par_5, write),
(d_3, D, par_6, write), (d_3, H, par_7, read),
(d_3, ENDFLOW, par_8, read)
}

Figure 3: Example of a simple data flow schema DF. Note, that the output parameter (input parameter) of the start node (end node) corresponds to the input parameter (output parameter) of the WF.

With respect to the management of data elements we follow an approach similar to that presented in [ReSc95]. When a task (resp. an associated service) updates a data element d its current value is not overwritten. Instead a new version is generated which may be accessed by succeeding tasks resp. services. This does not only allow us to restore previous values of data elements in the case of a partial rollback with forward recovery, but it also makes it possible for tasks from different branches of a parallel processing (with 1-of-n resp. n-of-n synchronizations) to work on different copies of the same data element d. As an example, assume that in figure 3 the task G has read access to the data element d_1. Although task C may write d_1 before G is started, this value would not be visible to G. G may only access the value of d_1 written by the start node of the flow. Generally, a task may only read those values of a data element which have been written by a task resp. service preceding it in the flow of control [cmp. Bla96, Hen97].

In summary, a WF schema P is described by a 5-tuple (N, E, S, D, DF), with finite and non empty sets N of tasks and E of directed edges between them. S denotes the set of services preceding resp. succeeding the execution of tasks. Furthermore D denotes the set of data elements and DF defines the set of data links, connecting task resp. service parameters with elements from D.

2.2 Workflow Execution

The state of a WF instance (i.e., the enactment of a WF schema) is one of the major criteria for deciding whether a specific structural change may be applied to it or not. As an example consider the deletion of a task which should not be allowed if the task has aready been completed. Furthermore after a structural change has been performed on a process graph, concomitant changes to the states of its nodes resp. edges may become necessary to proceed with the flow of control. The state of a newly inserted task, for instance, may have to be changed depending on the states of its predecessors (cmp. section 3).

The ADEPT WF model is based on a well-defined operational semantics to support this. The state of a WF instance p_i is defined by the current state of the nodes and edges of its execution graph, the values stored for its data elements (possibly in different versions) and relevant information regarding its execution history. The state of a single task n is described by the current state NS^n of its node ($NS^n \in$ {NOT_ACTIVATED, ACTIVATED, RUNNING, COMPLETED, FAILED, SKIPPED}), the total number It^n of its previous executions and relevant information on these executions. Finally, each edge e of the execution graph is in one of the states $ES^e \in$ {NOT_SIGNALED, FALSE_SIGNALED, TRUE_SIGNALED}.

When a WF instance p_i is created, first of all, the graph of its starting schema (N, E, S, D, DF) is initialized. The state of all nodes is set to NOT_ACTIVATED and the state of edges to NOT_SIGNALED. Furthermore the input data of p_i are stored in the corresponding data elements, i.e., elements from the set

$$\{d \in D \mid \exists\, df \in DF: n^{df} = \text{STARTFLOW} \wedge \text{access\_mode}^{df} = \text{write}\} \subseteq D.$$

When p_i is started, the start node of the graph is marked as COMPLETED and its outgoing control edge is set to the state TRUE_SIGNALED.

Each time an edge $n_1 \rightarrow n_2$ (of arbitrary type) is signaled the state of its destination node n_2 is reevaluated according to the *execution rules* defined by the ADEPT model. Executions rules define the conditions under which a node may be activated (i.e., routed to the corresponding worklists). If the destination node n_2 has the input firing behavior $V_{in}(n_2)$ = n-of-n, for instance, it is set to the state ACTIVATED in case it meets the following conditions: n_2 is in the state NOT_ACTIVATED and all ingoing control edges (ET = CONTROL_E) are in the state TRUE_SIGNALED. Furthermore all sync edges $n \rightarrow_{STRICT} n_2$, $n \in N$ with ET = STRICT_SYNC_E must be signaled as TRUE and all sync edges $n \rightarrow_{SOFT} n_2$, $n \in N$ with ET = SOFT_SYNC_E must be signaled as either TRUE or FALSE. On the other hand, n_2 is skipped if at least one of its ingoing control edges is signaled as FALSE. Corresponding execution rules exist for all node types (e.g., start resp. end nodes of loops).

On the other hand, the completion resp. skipping of a node leads to the signaling of its outgoing edges. Upon the successful completion of a task n with output firing behavior $V_{out}(n)$ = n-of-n, for instance, its outgoing control and sync edges are signaled as TRUE. This in turn, may trigger the execution of other tasks, and so on. On the other hand, if a task is skipped its outgoing edges are signaled as FALSE. The marking of edges follows well-defined *signaling rules*, which are based on the operational semantics defined for the different control structures of the ADEPT model.

We omit a detailed presentation of these rules and present two examples instead. Figure 4 shows the application of execution and signaling rules in conjunction with loops.

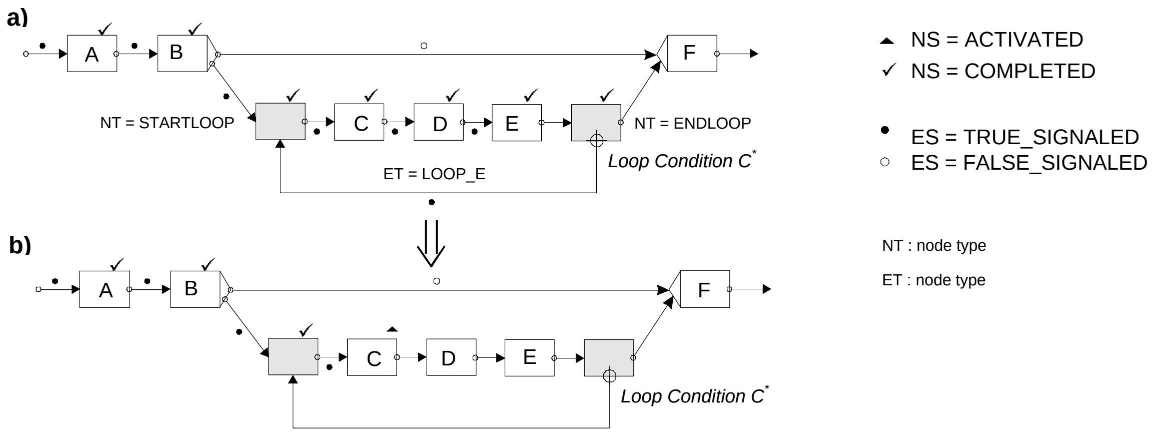


Figure 4*: Execution semantics of a loop*. After E was completed and the loop condition C^* was evaluated to TRUE the loop edge is signaled (fig. a). This, in turn, triggers the execution of the start node of the loop, whereupon the state of all nodes and edges of the loop body (incl. the loop's end node and the loop edge) are reset and C is triggered (fig. b).

As a second example consider figure 5. Assume that upon receiving a node termination event from B its outgoing control edge $B \rightarrow C$ is signaled as TRUE and the edge $B \rightarrow D$ as FALSE. This, in turn, leads to the revaluation of the nodes C and D which are activated resp. skipped. After D is skipped its outgoing control and sync edges are signaled as FALSE. Consequently the state of G is reevaluated and set to ACTIVATED (fig. 5b).

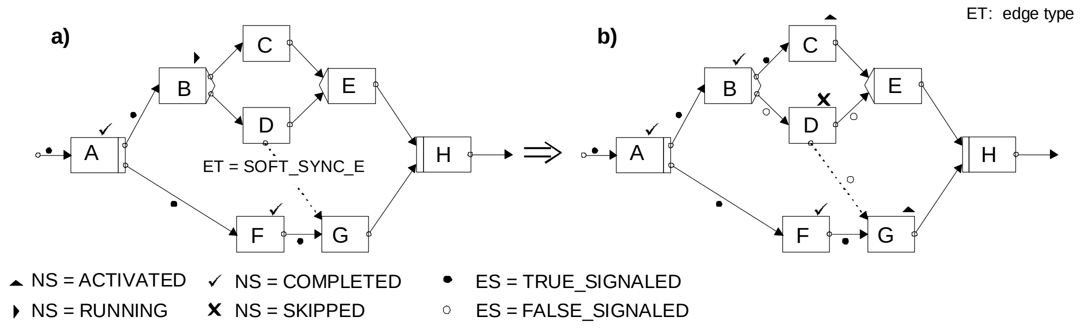


Figure 5: Example of a "soft" synchronization between two nodes from different branchings of a parallel processing. G is triggered after its ingoing control edge F $\rightarrow$ G was signaled as TRUE and the sync edge D $\rightarrow_{SOFT}$ G was signaled as FALSE.

Finally, a WF instance terminates when the ingoing control edge of its end node is signaled. As we will show in section 3, the presented execution and signaling rules are fundamental for the reevaluation of the state of a process graph after structural changes have been applied to it.

2.3 Correctness and Consistency Properties

As motivated in section 1, formal criteria are needed to enable the runtime system to decide whether a structural change can be applied to a WF resp. to identify the exceptions resulting from the change. In this section we give an overview of some of the correctness criteria defined by ADEPT. We focus on the flow of data. Properties regarding the correctness of the control flow are sketched at the end of this section.

Flow of Data

In the following we assume that for the correct execution of an action A (i.e., a task resp. an auxiliary service associated with the execution of a task) all input parameters must be supplied and that after its successful completion all output parameters are written. ADEPT imposes a set of restrictions which govern the nature of a correct data flow schema DF. For each data link df (cmp. definition 1) the domains of d^{df} and par^{df} must be type compatible. In addition, each input resp. output parameter of an action must appear in exactly one data link df $\in$ DF with $access\_mode^{df}$ = read resp. write. In order to avoid the invocation of actions with missing input data the following restriction has to be added:

Rule DF-1: Let (N, E, S, D, DF) be the schema of a workflow P. For n $\in$ N $\cup$ S let V^n denote the set of all valid action sets (incl. elements from N as well as from S) whose elements precede n in the flow of control and are completed before n is started. For n $\in$ N $\cup$ S, d $\in$ D we then require:

$$\text{Reads}(n, d) \Rightarrow (\forall V \in V^n: \exists n^* \in V: \text{Writes}(n^*, d))$$

The predicates used for the definition of this rule are defined as follows:

Writes (n,d):$\Leftrightarrow$ $\exists$ df $\in$ DF with n^{df} = n, d^{df} = d, $access\_mode^{df}$ = write

Reads (n,d):$\Leftrightarrow$ $\exists$ df $\in$ DF with n^{df} = n, d^{df} = d, $access\_mode^{df}$ = read

This rule ensures that all input parameters of a task resp. an auxiliary service are supplied before it may be executed. Trivially, for a given task $n \in N$, $NT^n \neq$ STARTFLOW which has read access to a data element d, rule DF-1 is satisfied when d is written by the start node of the process P or when it is written by a preceding auxiliary service $s \in S_n^{prec}$. Furthermore, this rule guarantees that the output parameters of P (i.e., the input parameters of the end node) are supplied. In [Hen97] we present a direct and efficient algorithm for checking the completeness of a data flow schema DF with respect to rule DF-1. The algorithm makes use of the symmetrical structuring of process graphs, but considers synchronizations between tasks from parallel branches (i.e., sync edges), too.

For a basic understanding, however, an example is more suitable. In the process graph depicted in figure 3 the task G may read the data elements d_1 and d_2, but is not allowed to access d_3 as this data element is not written within all task sets of V^G = { {STARTFLOW, A, B, D, F}, {STARTFLOW, A, B, F} }. The task H, however, may read the data elements d_1, d_2 and d_3 as each of them is written within all task sets from V^H = {{STARTFLOW, A, F, G, B, C, E}, {STARTFLOW, A, F, G, B, D, E} }.

In order to avoid unintended lost updates of data elements we do not allow tasks from different branches of a parallel processing (with n-of-n join) to have write access to the same data element, unless they are synchronized by a sync edge. In the example from figure 3, G is not allowed to write d_3 as this data element may be written by the concurrent task C. Write-after-write conflicts may also occur if two succeeding tasks have write access to the same data element and no read access occurs between them. The following two constraints aim at avoiding these cases:

Rule DF-2: Let P = (N, E, S, D, DF) be the schema of a workflow.

For $n_1, n_2 \in N$ with $\text{Writes}(n_1,d) \wedge \text{Writes}(n_2,d)$ we require

(1) $\left(n_1 \in \overline{succ}(n_2) \vee n_2 \in \overline{succ}(n_1)\right)$ **or**

$\left(\exists\, n_s \in N: V_{in}(n_s) = \text{1-of-n} \wedge n_s \in M := \left(\overline{succ_c}(n_1) \cap \overline{succ_c}(n_2)\right) \wedge \forall\, n \in M, n \neq n_s: n \in \overline{succ_c}(n_s)\right)$[5]

(2) $n_2 \in \overline{succ}(n_1) \Rightarrow \exists\, n_3 \in \left(\overline{succ}(n_1) \cap \overline{pred}(n_2)\right) \cup \{n_2\}$ with $\text{Reads}(n_3,d)$

We define

$succ: N \rightarrow P(N)$ with

$$succ(n) = \left\{n' \in N \mid \exists\, e \in E:\ e = n \rightarrow n' \wedge ET^e \in \{\text{CONTROL\_E, SOFT\_SYNC\_E, STRICT\_SYNC\_E}\}\right\}$$

$\overline{succ}: N \rightarrow P(N)$ with

$$\overline{succ}(n) = \left\{n' \in N \mid n' \in succ(n) \vee (\exists\, n'' \in succ(n): n' \in \overline{succ}(n''))\right\}$$

[5] If n_1 and n_2 do not succeed each other in the control flow, we require that they must be contained within different branches of a branching with a 1-of-n join. n_s denotes the corresponding join node with the input firing behavior $V_{in}(n_s)$=1-of-n.

succ(n) defines the set of direct successors of the node $n \in N$, i.e., the set of nodes which are the destination of a control resp. sync edge with source n. $\overline{succ}$ denotes the transitive closure of this function. $\overline{succ}(n)$ comprises those tasks of the process graph that are reachable from n by following control as well as sync edges. On the other hand the set $\overline{succ_c}(n) \subseteq \overline{succ}(n)$ comprises only those nodes from N which are reachable from n by following control edges. In the process graph shown in fig. 3 we have

$$\overline{succ}(B) = \{C, D, E, G, H, ENDFLOW\}, \quad \overline{succ_c}(B) = \{C, D, E, H, ENDFLOW\}.$$

As the meaning of the corresponding predecessor functions and their transitive closures is intuitive we omit their definition here.

Simplistically, we have omitted write operations of elements from S in the presentation of this rule.

Of course, the constraints resp. assumptions made for the definitions of rule DF-1 resp. rule DF-2 may be relaxed in several respects. In [Hen97], for example, we describe a more flexible approach which distinguishes between optional and mandatory input resp. output parameters of tasks and WFs. This has turned out to be quite important in our experience, since not always all input parameters must be supplied for the correct processing of a task program. Furthermore, concurrent write operations to the same data element must be allowed under certain conditions and referenced data must be handled [cmp. Rei97].

Note, that structural changes of a WF may violate the presented rules if no further precautions are made. The deletion of a task, for instance, is accompanied by the deletion of the data links connecting its output parameters with elements from D. This, in turn, may lead to missing parameter data for succeeding steps and therefore to a violation of rule DF-1. On the other hand, the dynamic insertion of a task and the addition of data links connecting its output parameters with elements from D may lead to lost updates and therefore to the violation of rule DF-2. We will come back to this in section 3.

Flow Of Control

In addition to the properties defining a valid data flow schema DF, the process graph P = (N, E, S, D, DF) must meet further *constraints* in order to ensure the correct and consistent execution of a WF at runtime. First of all, we require that P is *safe* [cmp. HOR96]. This means that from every reachable state of the WF a terminal state can be reached, i.e., there is a valid sequence of signaling events leading from the current state to the execution of the end node of P. Secondly, each node $n \in N$ must be *reachable* from the start node of P. For acyclic graphs which are based on sequential executions and symmetrical branchings these properties are satisfied by construction. This does not apply to a control flow graph (N, E) whose control structures contain backward resp. sync edges. For example, the use of sync edges must not lead to cycles resp. termination problems of the WF. Conditions under which a graph (N, E) satisfies these properties and algorithms for their analysis are outside the scope of this paper and are presented in [Rei97].

3 ADEPT$_{\text{flex}}$ – Supporting Dynamic Structural Changes Of Workflows

Based upon the ADEPT model we have developed a set of operations (ADEPT$_{\text{flex}}$) which serve as the framework for dynamic structural changes of WFs during their execution. The main emphasis in designing these operations was put on

- *correctness* / *consistency*: the application of a change operation to a specific WF instance should result in a WF with a syntactically correct schema and with a "legal" state, i.e., the change should not cause inconsistencies and runtime errors.
- *adequacy* / *completeness*: each operation should be applicable to any WF instance with arbitrarily structured schemas; the set of operations should be *complete* in the sense of being able to realize each possible form of (correct and consistent) restructuring of a process graph
- *minimality*: the operations needed to achieve completeness should form a *minimal* set

Other design goals, which we do not discuss in detail in this paper, concern *efficiency* and *security* issues as well as *ease of use*. With efficiency we mean that it must be possible to make changes "on the fly" without disturbing other process participants in their work. Security issues deal with the restriction of change operations to selected users (resp. roles), to specific process types, to specific regions of a process graph (e.g., a single task), to certain states of a process or to any combination of them. Finally, adding only functionality to current WF technology, without understanding how the user will be able to utilise it, will certainly not be very helpful. Change operations, whether used by programmers or, in some form, by end users must be usable and manageable in a secure and proper manner.

In summary, ADEPT$_{\text{flex}}$ comprises operations for inserting and deleting tasks (resp. task blocks) into resp. from a process graph, for fast forwarding the progress of a process by skipping tasks, for jumping to currently inactive parts of a process graph, for serializing tasks that were previously allowed to run in parallel (and vice versa) and for the dynamic iteration resp. rollback of a process resp. process region (incl. the undoing of temporary changes). These operations, in turn, provide the basis for implementing higher-level operations such as the replacement of a certain process region by a new one.

The insert operation shall serve as an illustrative example and will be discussed in more detail. The other operations are sketched at the end of this section.

3.1 Dynamic Insertion Of Tasks

The *addition of a new task to a WF at runtime* becomes necessary due to several reasons. The support of ad hoc as well as dynamically evolving WFs, unplanned events and missing or incomplete data name a few examples. The dynamic addition of a task to a WF is some-what comparable to the addition of a new procedure to a program in the midst of its execution. When a task is inserted into a process graph, new nodes and edges (including data links) must be added, while maintaining the correctness and consistency of the process. Current state-of-the-art systems do not provide a sufficient level of flexibility and consistency with respect to this operation. Typically they only allow the addition of an activity upon completion of a task and before the activation of its successors [e.g., HsKl96, CCP96, VoEr92]. Important aspects, e.g., concerning data integrity, are mostly ignored, leading to the problems mentioned in the introduction section. For the flexible support of processes a more generic approach is required. Generally, it must be possible

- to add new tasks or even premodelled task blocks to a WF at any point of time during its execution
- to work on an inserted task concurrently to other tasks of the WF; for example, the insertion of a new activity as a successor of a task n should not necessarily delay the execution of the other successors of n
- to synchronize the execution of an inserted task with the execution of other tasks from the process graph
- to insert tasks into process regions which have not yet been activiated (e.g., in the case of dynamically evolving WFs)
- to dynamically map the parameters of the inserted task to existing or to newly generated data elements
- to allow authorized (not necessarily human) agents to add a new task to a WF which then has to be worked on by other agents

There is no problem to provide an operation for inserting a new task as a direct predecessor resp. successor of a given node, for adding a task as a new branch between a split node and its corresponding join node, and so on. However, this approach would not yield to a satisfactory solution, as it does not reconcile with our design goals minimality and ease of use. Supporting the addition of tasks to a process graph raises the challenge to find a *single*, generic operation that is *complete* in the sense of being able to realize each possible form of insertion. It is obvious that the addition of a task as a direct successor of another task is too weak to meet the presented requirements.

We therefore follow a more generic approach: a new task X (together with associated auxiliary services S_X, data elements D_X and data links DF_X) may be inserted into the graph of a process instance by synchronizing its execution with two node sets M_{before} and M_{after}. The execution semantics of the added task should be as follows: X is triggered as soon as all tasks from the set M_{before} are either completed or cannot be triggered anymore (i.e., the tasks defined by M_{before} delay the execution of X). This allows us to synchronize X with (preceding) tasks from different execution branches of the process. On the other hand, tasks from M_{after} may only be activated after X has been completed. Note, that the addition of a new task transforms the schema (N, E, S, D, DF) as well as the state (NS, ES) of the WF to a new schema (N', E', S', D', DF') resp. state (NS', ES') (cmp. figure 6). It has to be ensured that such transformations are leading to a WF with a syntactically correct schema (incl. the flow of data) and with a legal state. In order to achieve this, several constraints regarding the definition of the sets M_{before}, M_{after}, D_X, S_X and DF_X as well as the structure and the state of the WF must be made. Before we discuss them in detail, we sketch the algorithm of the insert operation. First of all, we concentrate on the restructuring of the flow of control. Afterwards we discuss relevant aspects regarding the correctness of the flow of data when a new task is inserted.

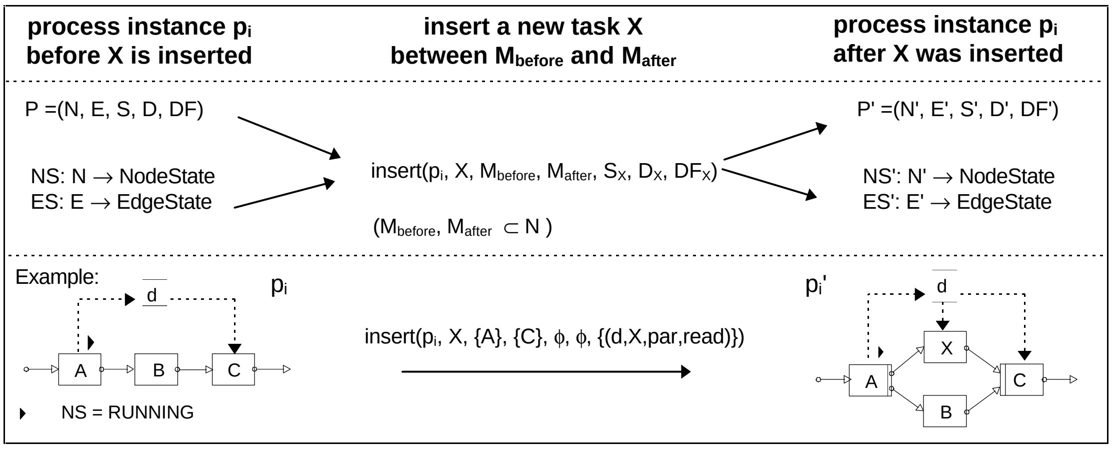

process instance p_i before X is inserted
insert a new task X between M_{before} and M_{after}
process instance p_i after X was inserted
P =(N, E, S, D, DF)
NS: N → NodeState
ES: E → EdgeState
insert(p_i, X, M_{before}, M_{after}, S_X, D_X, DF_X)
(M_{before}, M_{after} ⊂ N)
P' =(N', E', S', D', DF')
NS': N' → NodeState
ES': E' → EdgeState
Example:
p_i
d
A
B
C
NS = RUNNING
insert(p_i, X, {A}, {C}, ϕ, ϕ, {(d,X,par,read)})
p_i'
X

Figure 6: Dynamic insertion of a new task X (together with associated auxiliary services S_X, data elements D_X and data links DF_X) between two task sets M_{before} and M_{after}

Algorithm

In the following, let (N,E) be the syntactical correct control flow graph of a WF instance p_i. The following algorithm can be used to insert a new task X between the two node sets M_{before} and M_{after}:

1. Find the *minimal block* B ⊆ (N,E) that contains all nodes from $M_{before} \cup M_{after}$ (excluding the start node and the end node of (N, E))[6]. Let n_{begin} denote the start node and let n_{end} denote the end node of B.
2. Insert an *empty n-of-n split node* n_1 that represents a null task[7] as the predecessor of the node n_{begin}. The node n_1 takes over the input firing behavior and the ingoing control edges of n_{begin}
3. Insert an *empty n-of-n join node* n_2 representing a null task as the successor of the node n_{end}. The node n_2 takes over the output firing behavior and the outgoing control edges of n_{end}
4. Insert a node, representing X, as a new branch between the nodes n_1 and n_2
5. *Synchronize* X with the tasks from M_{before} and M_{after}, i.e., for each $B \in M_{before}$ (excl. the start node of (N, E)) add a sync edge $B \rightarrow_{SOFT} X$ with ET = SOFT_SYNC_E and for each $A \in M_{after}$ (excl. the end node of (N, E)) add a sync edge $X \rightarrow_{SOFT} A$ with ET = SOFT_SYNC_E[8]
6. Apply *reduction rules* and *reevaluate* the state of nodes and edges (see below)

[6] B is defined as the minimal subgraph of (N,E) that satisfies the following conditions: B has a unique start and a unique end node. If any node (except the end node) of B corresponds to a split node resp. to the start node of a loop, the correponding join node resp. end node of the loop must be contained within B, too. Finally, if any node (except the start node) of B corresponds to a join node resp. to the end node of a loop, the correponding split node resp. start node of the loop must also be contained within B.

[7] We have adopted this notion from [CCP96]. A null task does not correspond to any action in the real world. Its execution semantics is as follows: after the node of a null task is triggered, its outgoing edges are signaled immediately.

[8] If X is inserted between the start node and the end node of the process graph (N, E), no additional synchronizations are required.

This algorithm has to be extended if a user wants to insert a new task between the start resp. end node of a loop and an arbitrary node contained within the body of this loop. Two special cases, the dynamic addition of new branches to branchings with 1-of-n join, are also not covered by the presented algorithm. The necessary extensions are described in [Hen97].

We omit further details of this algorithm and present two examples instead. The first one is depicted in figure 7 and shows how a task X is inserted between two sets of nodes. One can easily see that the symmetrical structuring of the process graph is preserved and that the insertion of the sync edges does not violate the safeness (cmp. section 2.3) of the WF. The example further shows that empty nodes and sync edges might be added to the process graph, which are not necessarily required to achieve the desired execution semantics. These nodes resp. edges may be removed from the resulting graph by applying a set of well-defined reduction rules. Examples for such rules are depicted in figure 8. The effect of their application to the graph from figure 7c) is shown in figure 7d). A second example is given in figure 9 where a new task is inserted between a n-of-n split node and its corresponding join node. Using the presented insert algorithm and applying reduction rules the expected result is obtained (cmp. figure 9c).

State Constraints

Up to now we have not considered the state of the WF. To avoid the insertion of a new task as a predecessor of an already running or even terminated task, we require that all elements from the set M_{after} must be in one of the states NOT_ACTIVATED or ACTIVATED. The nodes from M_{before}, however, may be in an arbitrary state. When a task is inserted as a

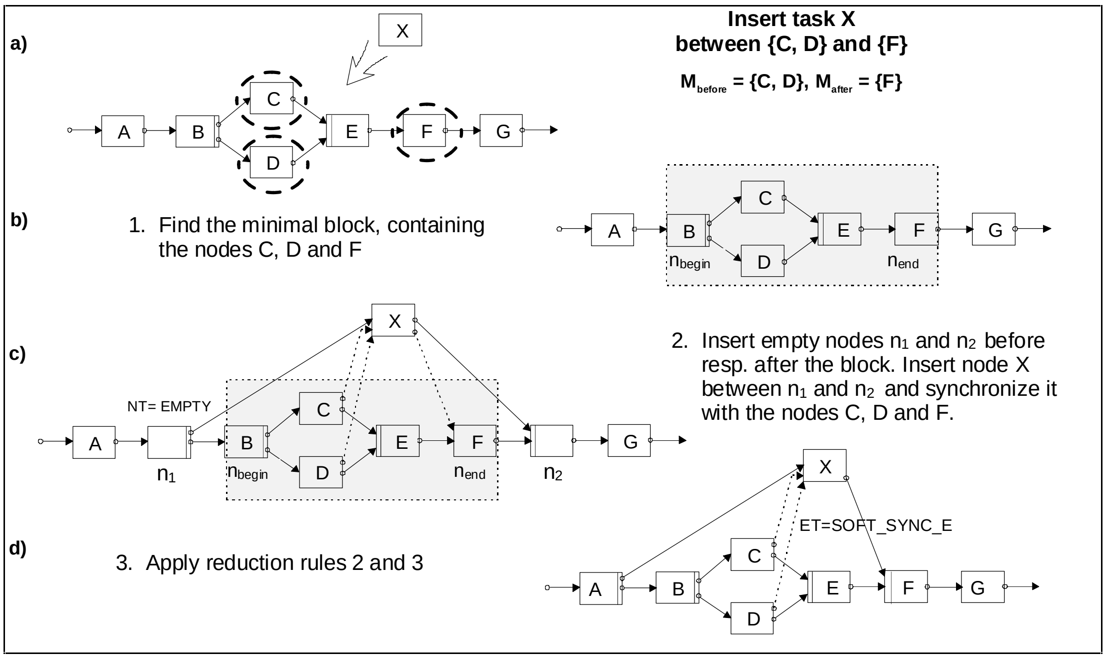

a)
X
Insert task X
between {C, D} and {F}
M_before = {C, D}, M_after = {F}
A
B
C
D
E
F
G
b)
1. Find the minimal block, containing the nodes C, D and F
n_begin
n_end
c)
NT= EMPTY
n1
n2
2. Insert empty nodes n1 and n2 before resp. after the block. Insert node X between n1 and n2 and synchronize it with the nodes C, D and F.
d)
3. Apply reduction rules 2 and 3
ET=SOFT_SYNC_E

Figure 7: Insertion of a new task between two sets of nodes: first of all, the minimal block that contains all nodes from the set {C,D,F} is determined (fig. b). In the next step, an empty split node n_1 is inserted between the predecessor A and the start node B of the block. In the same way a corresponding join node n_2 is added. Finally, X is inserted as a new branch between n_1 and n_2 and synchronized with the nodes C, D and F by adding the sync edges C $\rightarrow_{SOFT}$ X, D $\rightarrow_{SOFT}$ X and X $\rightarrow_{SOFT}$ F (fig. c). The application of reduction rules to the graph from fig. c) leads to the graph depicted in fig. d).

Reduction Rule 1:

Combine the empty, parallel split node and its successor A (V_{out}^{A}=1-of-1) in one node.

Reduction Rule 2:

Combine the empty, parallel join node and its predecessor B (V_{in}^{B} = 1-of-1) in one node.

Reduction Rule 3:

Combine node A (V_{out}^{A}=1-of-1) and its empty successor node in one node, taking over the output firing behavior of the empty node.

Reduction Rule 4:

Combine the empty node and its successor B (V_{in}^{B} = 1-of-1) in one node, taking over the input firing behavior of the empty node.

Reduction Rule 5:

Remove "unnecessary" control edges between a n-of-n split node A and its corresponding join node B. If the number of remaining branches is 1, set the output resp. the input firing behavior of A resp.B to 1-of-1.

Reduction Rule 6:

Combine the empty nodes with their successor A (V_{out}^{A}=n-of-n) resp. predecessor B (V_{in}^{B} = n-of-n)

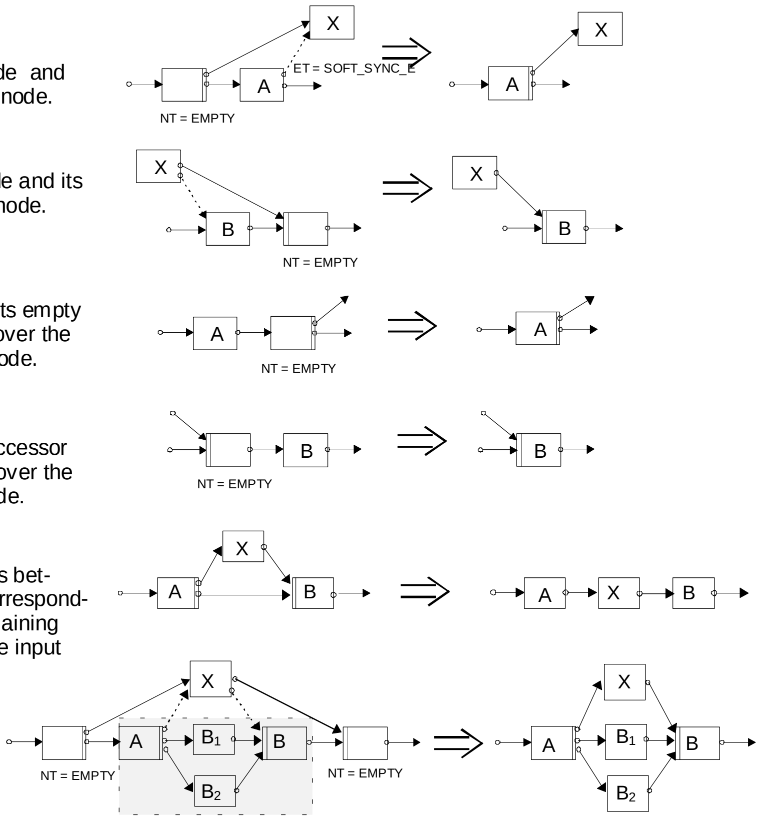


Figure 8: Reduction rules (RR). RR may be applied to the empty nodes (i.e., the null tasks) originating from the insertion of a node and to their direct successors resp. predecessors. Their application preserves the execution behavior of the process graph, i.e., the set of valid task sequences remains unchanged.

predecessor of an already activated task, it is checked whether this task was already routed to any worklist. If this is the case the corresponding work items are removed from the worklists before the insertion can take place. As an example, taking the graph depicted in figure 9a) a new task X may not be inserted between the nodes D and E. To insert a new task between D and F, first of all, the execution of F would have to be aborted by the user.

After a task X was added to the process graph, the state of its nodes and edges (incl. the newly inserted ones) is reevaluated This reevaluation is based on the execution and signaling rules presented in section 2.2. Whether X is triggered immediately or later depends on the current state of the graph. The former is the case, if at insertion time all nodes from M_{before} are in one of the states COMPLETED or SKIPPED. It is obvious that for this case the node n_{begin} must be in one of these states, too, so that the added split node n_1 is marked as COMPLETED and its outgoing control edges (with destinations X resp. n_{begin}) are set to the state TRUE_SIGNALED. As all elements from M_{before} are in a final state, the inserted sync edges connecting these nodes with X are signaled, too. So the rules for the execution of X are satisfied and X is triggered. As a simple example consider the graph shown in figure 9b) whose reevaluation yields the result depicted in figure 9c). Note, that the insertion of a new task does not mean that it will be activated at all. If the task is inserted into a process region which is currently not active, the execution of the task may depend on future routing decisions.

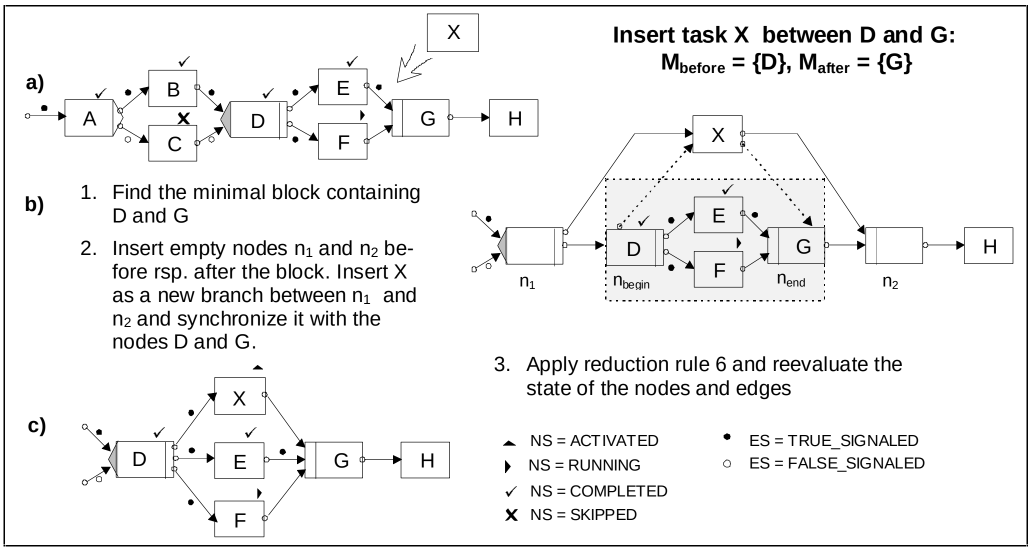

Insert task X between D and G:
$M_{before} = \{D\}$, $M_{after} = \{G\}$
a)
b)
1. Find the minimal block containing D and G
2. Insert empty nodes n_1 and n_2 before rsp. after the block. Insert X as a new branch between n_1 and n_2 and synchronize it with the nodes D and G.
3. Apply reduction rule 6 and reevaluate the state of the nodes and edges
c)
A
B
C
D
E
F
G
H
X
n_1
n_{begin}
n_{end}
n_2
NS = ACTIVATED
NS = RUNNING
✓ NS = COMPLETED
NS = SKIPPED
● ES = TRUE_SIGNALED
○ ES = FALSE_SIGNALED

Figure 9: Example for adding a new task X between a n-of-n split node D and its corresponding join node G. Note, that X is added as a new branch between D and G, although the successors of D, the nodes E and F, have already been completed resp. started.

Structural Constraints – Flow of Control

The presented examples demonstrate that the application of the insert operation does not violate the syntactical correctness of the graph (N,E) and does not lead to termination problems. The following theorem defines the conditions for this:

Theorem 1 (Syntactical Correctness and Safeness of the Resulting Control Flow Graph): Let (N, E) be the syntactically correct control flow graph of a WF schema P which is (1) safe and for which (2) every node $n \in N$ is reachable from the start node of P (cmp. section 2.3). Furthermore let M_{before}, $M_{after} \subset N$ be two disjoint sets with

(I1) $\forall n_b \in M_{after}, \forall n_a \in M_{before} : n_b \in \overline{succ}(n_a)$ i.e., for all $n_a \in M_{before}$, $n_b \in M_{after}$ we require that n_a precedes n_b in the flow of control

(I2) The region covered by the nodes from the set $M_{before} \cup M_{after} \cup \left(\overline{succ}(M_{before}) \cap \overline{pred}(M_{after})\right)$ may only contain complete loop control structures

Then, the application of the insert operation to add a new task X between the sets M_{before} and M_{after} results in a syntactically correct control flow graph (N', E') again, satisfying the properties (1) and (2), too.

We only sketch the idea for the proof of this theorem (for details see [Rei97]) without considering reduction rules. On the whole, the insert operation substitutes a (logical) block B of the graph (N,E) by a symmetrical block, namely a parallel branching with the inserted task X and B as its branches. The symmetrical structuring of the graph is therefore preserved and the insertion of the empty nodes resp. null tasks does not influence the execution resp. termination behavior of the WF. The restrictions for the use of sync edges (cmp. section 2.1) are further satisfied as the added edges do only synchronize tasks from different branches of a parallel branching (namely the task X with tasks from B), do not link a node contained within a loop body with the inserted task X (cmp. condition I2) and do not lead to cycles resp. termination problems. The latter is guaranteed by the ordering of tasks from the sets M_{before} and M_{after} (cmp. condition I1). Based on this and on the properties (1) and (2), which are valid for the starting graph (N, E), one can easily show that (N', E') satisfies these properties, too. Note that only sync edges of the type ET = SOFT_SYNC_E are used. □

Finally, to avoid unnecessary synchronizations between the inserted task and nodes from M_{before} resp. M_{after} the following minimization rules may be applied to these sets before the insertion is performed:

- delete all elements $n_a \in M_{before}$ with $\exists\, n_a^* \in M_{before}$: $n_a^* \neq n_a$ and $n_a^* \in \overline{succ}(n_a)$
- delete all elements $n_b \in M_{after}$ with $\exists\, n_b^* \in M_{after}$: $n_b^* \neq n_b$ and $n_b^* \in \overline{pred}(n_b)$

These rules guarantee that two nodes from M_{before} (resp. M_{after}) do not succeed each other in the flow of control (i.e., they are contained within different execution branches of the process graph and are not reachable from each other). When X is inserted into a sequence of tasks, for instance, M_{before} as well as M_{after} contain exactly one node after the application of these minimization rules.

The restrictions presented so far are checked before the insertion of the corresponding task takes place. Note, that the presented constraints do not yet concern the flow of data.

Structural Constraints – Flow of Data

As already mentioned, a new task X may be "plugged" into the graph, together with associated auxiliary services S_X, data elements D_X, and data links DF_X. So when a task is added to the WF schema (N, E, S, D, DF) this does not only lead to the modification of the control flow graph (N, E) and its state, but generally requires extensions of the sets S, D and DF, too. Regarding the flow of data, the resulting WF schema (N', E', S', D', DF') has to meet the correctness properties defined in section 2.3. First of all, according to rule DF-1, it must be ensured that all input parameters of X are supplied before X may be executed. A simple approach to achieve this would be to request the input data from the user initiating X. For this, X has to be connected with a preceding provider service s (cmp. section 2.3), whose output parameters correspond to the input parameters of X. In our current prototype implementation such a service is supported by the dynamic generation and processing of an electronic form which makes use of the interface description of X. The following procedure shows how the sets S_X, D_X and DF_X must be defined in order to obtain a syntactically correct data flow schema that satisfies rule DF-1.

```
$D_X := \emptyset$;  $DF_X := \emptyset$;
create a *provider service* s with $OutPars(s) := \emptyset$;
**for all** $par \in InPars(X)$ **do**
    create a data element $d_p$ with $(Id^{d_p} \neq Id^d\ \forall d \in D \cup D_X) \wedge (dom^{d_p} = dom^{par})$ and insert it into $D_X$:
        $D_X := D_X \cup \{d_p\}$
    create a parameter p with $(Id^p = Id^{par} \wedge dom^p = dom^{par} \wedge dir^p = "OUT")$ and add it to OutPars(s)
        $OutPars(s) := OutPars(s) \cup \{p\}$
    insert corresponding data links into $DF_X$
        $DF_X := DF_X \cup \{(d_p, s, p, write), (d_p, X, par, read)\}$
**end**
$S_X = S_X^{prec} := \{s\}$          (i.e., s will be triggered when X is initiated, cmp. section 2.1)
```

If the original WF schema (N, E, S, D, DF) satisfies rule DF-1 this also applies to the schema (N', E', S', D', DF') with $S' := S \cup S_X$, $D' := D \cup D_X$ and $DF' := DF \cup DF_X$. In practice, however, this simple approach would not always yield to a satisfactory solution, since unnecessary and redundant data entries may result in the course of a WF execution, potentially leading to data inconsistencies. Especially for the support of ad hoc and dynamically evolving WFs a more generic approach is required, allowing the "intelligent" mapping of input resp. output parameters of the inserted task to already existing data elements from D, too. This, however, raises a variety of challenging issues with respect to dynamic parameter mapping as well as the management of data elements, which can only be sketched here. First of all, the set of possible data elements C_X to which input parameters from X may be mapped must be identified. According to rule DF-1, we obtain

$$C_X = \{d \in D \mid \forall V \in V^X: \exists n^* \in V: \text{Writes}(n^*, d)\}.$$

An algorithm for the determination of the set C_X is presented in [Hen97]. As an example consider the process graph depicted in figure 3 and assume that a task X should be inserted between the nodes B and C. Then we obtain $C_X = \{d_1, d_2\}$. Note, that this approach does not consider the state of the WF. It is therefore ensured that all data elements from C_X are supplied when X is activated, independently from any routing decision or backward operation made in the course of execution. On the other hand, there are scenarios in which it would be quite useful to relax this assumption and to consider the state of the WF (incl. the state of its data elements), too. In this case, the set of data elements to which input parameters from X may be linked is extended to

$$C_X^* = C_X \cup \{d \in D \mid \exists n^* \in \overline{\text{pred}}(X): NS^{n^*} = \text{COMPLETED} \wedge \text{Writes}(n^*, d)\}^{9,10}$$

As an example, take the insertion shown in figure 9 and assume that B is the only task that writes a specific data element $d_1 \in D$. Since B is completed at the time X is added, we have $d_1 \in C_X^*$. Input parameters from X may therefore be mapped to d_1, although this data element is not contained in the set C_X (cmp. figure 10). Following this approach, however, it might become necessary to undo the insertion of X in case of a backward operation (cmp. section 4). As we will see in section 4, in this context, it makes a big difference whether the insertion of X is of temporary nature, i.e., X should be executed at most once (*temporary insertion*), or should be valid until completion of the WF (*permanent insertion*).

Up to now we have only dealt with the question which data elements may be considered when input parameters of X shall be mapped to elements from D. A specific input parameter $p \in \text{InPars}(X)$ may be linked to a data element $d \in C_X$ (resp. C_X^*) if their domains correspond to each other. Of course, this purely syntactical approach would be insufficient in practice and would leave significant complexity to the user. A more sophisticated approach, which aims at the semi-automatic mapping of parameters to data elements, is

[9] As a restriction, this set may not contain data elements written within branches of a parallel branching with a 1-of-n join node s, if s has not yet been triggered and X succeeds s in the flow of control (cmp. section 2.1). Furthermore, the runtime system checks whether the corresponding data elements have been really written by the corresponding tasks.

[10] For simplification we have omitted write operations from elements of S in the definition of this set.

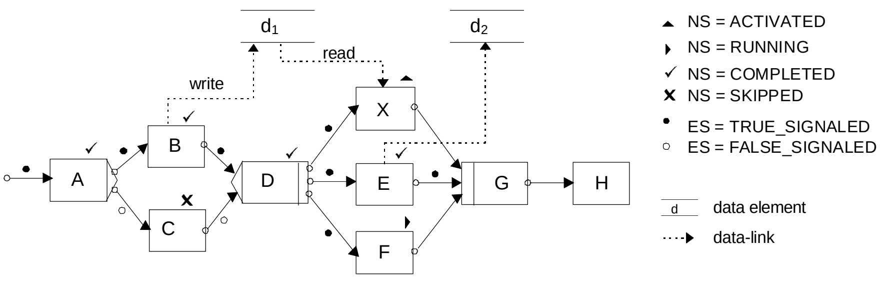


Figure 10: Linking an inserted task to existing data elements.

presented in [Bla96]. Basic to this approach is a controlled vocabulary which is used for the naming of data elements and parameters as well as the data structures they are built upon. The vocabulary is organized as a semantic network and does therefore also consider semantic relationships between the underlying concepts of data elements resp. parameters. A presentation of this semantic mapping approach, however, is outside the scope of this paper. In [Bla96] we also deal with the problem of heterogeneous structures and formats of parameter data from different tasks.

Similar reflections must be made regarding linkages of the output parameters of an inserted task to existing resp. newly inserted data elements. In order to avoid unintended lost updates, an output parameter of an inserted task can only be linked to a data element if rule DF-2 is further satisfied. In the process graph shown in figure 10, for instance, the output parameters of the newly inserted task X may not be mapped to the data element d_2.

Further Issues

In this section we have concentrated on correctness and consistency issues regarding the dynamic addition of a task to a WF schema resp. to its flow of control and data. For the sake of completeness, some important aspects, which are not further addressed in this paper, have to be mentioned.

First of all, in our experience it has turned out to be quite important to allow process participants to fix a date or a deadline for the inserted task. The necessary extensions for this are described in [Gri97].

Secondly, for security reasons, ADEPT$_{flex}$ allows WF designers (as well selected process participants) to restrict the use of the insert operation to specific process types resp. categories, to selected users resp. roles, to specific regions of a process graph, to selected states of a process, to specific activity types resp. categories, or to any combination of them. Generally, we do also not require that the user who adds a task to a WF must then work on it. This provides additional flexibility to process participants, as they are allowed to add tasks to a WF, whose execution is then explicitly or implicitly delegated to other process participants. Basic to this is a powerful meta model for capturing organizational entities and relationships between them. The interested reader is refered to [KoRe96].

Finally, for the implementation of flexible client applications and worklist handlers a corresponding set of (generic) API calls is offered to application programmers [Hen97]. The provided functions can also be used to query information about the context in which the insertion should be applied.

Now we have got an operation that satisfies our main design goals with respect to correctness, consistency, and minimality. Using the insert operation, it is easy to cover a broad spectrum of applications and to implement a variety of user-friendly operations. Some of them are summarized in table 1.

insertion of	choice
an intermediate step between a node A and its successors (A may be a split node of arbitrary type)	M_{before} = {A} and M_{after} = succ(A)
an intermediate step preceding the execution of a task B; B may have already been routed to worklists, but may not yet be worked on by a user	M_{before} = pred(B), M_{after} = {B}
a new branch to a parallel branching with split node S and join node J	M_{before} = {S}, M_{after} = {J}
a new task without synchronizing it with other tasks of the WF	M_{before} = {STARTFLOW}, M_{after} = {ENDFLOW}

Table 1: Examples for the use of the insert operation.

The insert operation may also serve as the basis for composing *higher-level operations*. For example, multiple concurrent instantiations of the same task type (*dynamic task*) can be realized by the repetitive use of this operation within one change transaction. The generality of the insert operation also provides the basis for the *ad hoc definition of WFs*: a WF starts with a single (stop) node between the start and the end node of the process graph and may be dynamically extended by the (possibly concurrent) application of the insert operation. The WF does not terminate until an authorized user initiates the stop node. A similar application is the support of dynamically evolving WFs, where the planning and the execution of individual WF instances interleave. As a last interesting aspect, we use the insert operation for internal exception handling, too. For example, if the deletion of a task X leads to incomplete or missing parameter data of succeeding, data-dependent tasks, a corresponding provider task, taking over the data links from X, may be plugged into the graph and be synchronized with these tasks (cmp. section 3.2)

These examples demonstrate that our approach is adequate for supporting a variety of insertion scenarios with the same generic operation. In the next section we sketch other change operations and some interesting aspects in conjunction with them.

3.2 Overview of other change operations

As said before, ADEPT$_{flex}$ comprises a set of basic change operations which allow authorized users to add tasks to a WF, to delete tasks from a WF, to skip the execution of tasks (with or without finishing them later), to jump forward to process regions which have not yet been activated, to serialize tasks that were previously allowed to run in parallel, and to perform backward operations on a process graph (incl. the undoing of temporary changes). Due to space limitations we omit a presentation of the whole set of operations here and refer to [Rei97, Hen97] for their definition, constraints for their use and implementation issues. In the following, we summarize some interesting aspects concerning the deletion of tasks and the dynamic modification of premodelled task sequences.

Dynamic Deletion of Tasks

A task may have to be removed when the conditions for its execution become unnecessary. Of course, not all tasks may be deleted from a process graph. Firstly, nodes which are an integral part of the WF structure (e.g., the start and the end node of the WF schema) cannot be deleted at all. Secondly, WF designers may customize a WF schema with respect to the applicability of this operation to individual tasks resp. process regions.

The deletion of a task X is only allowed, if it is either in the state ACTIVATED or NOT_ACTIVATED. In the former case, the work items associated with the task are removed from the corresponding worklists. Tasks in the state RUNNING, COMPLETED, FAILED, or SKIPPED may not be deleted.

Regarding the adjustment of the control flow graph, the delete operation is realized by transforming the corresponding node into an "empty" node resp. a null task (cmp. section 3.1). This approach can be handled in a simple and effective manner, as the node itself as well as its ingoing and outgoing control resp. sync edges are still part of the WF structure. As we will see in section 4, this also facilitates the undoing of task deletions.

When a task X is deleted, however, its associated auxiliary services and data links must be removed from the sets S resp. DF. This, in turn, may lead to missing or incomplete input data of succeeding, data-dependent steps and therefore to a violation of the correctness properties defined in section 2.3.

Let $N^{*} \subset \overline{\text{succ}}(X)$ denote the set of tasks, whose input parameters are not completely supplied due to the deletion of X. The following exception handling policies are offered by ADEPT$_{\text{flex}}$ to deal with the missing data:

- concomitant deletion of tasks from the set N^{*}, which in turn may require the deletion of other tasks from N (cascading delete).
- dynamic insertion of a provider task X_{prox} into the flow of control (with $M_{\text{after}} = N^{*}$). X_{prox} takes over the data links of the deleted task and must be completed before any task of the set N^{*} may be triggered.
- dynamic addition of corresponding provider services (i.e., dynamically generated forms) to the sets S_n^{prec}, $n \in N^{*}$ (cmp. section 3.1) – this, however, must not lead to the violation of rule DF-2!
- abortion of the delete operation

Of course, these policies may be used in combination with each other, too. To relieve users from performing the necessary adjustments of the data flow schema themselves, ADEPT supports the specification of *success dependencies* between (succeeding) tasks. If a task X is deleted from the process graph at runtime, all suceeding tasks which are success-dependent on X are deleted, too, and so on (cascading delete). Concerning the flow of data, this approach does not require any additional exception handling, if for each task the set of its success-dependent steps corresponds to that of its data-dependent steps. Note, that this approach is similar to the concept of *spheres of control* proposed in [Dav78, Ley95], but is applied to the structure of the WF.

Changing Task Sequences At Runtime

As mentioned in the introduction section, changes to premodelled task sequences frequently become necessary in exceptional situations. Since WF designers are generally not capable to predict all possible deviations in advance, operations are required that allow users to dynamically skip the execution of tasks (with or without finishing them later) or to work on tasks of which the execution conditions are not yet satisfied. As an example, take the process graph depicted in figure 11a) and assume that an authorized user wants to jump forward to task G and to proceed with the flow of control at this node, although the ingoing edges of this task have not yet been signaled. Assume further, that the skipped steps D, E and F have to be finished resp. worked on concurrently, but must be completed before task J may be triggered. To achieve this, the process graph must be restructured as shown in figure 11b). Note, that this restructuring leads to the parallelization of tasks that were previously constrained to be executed serially. Generally, it should be possible to pass the control resp. to jump forward to a node n_{target}, which may not yet have been activated (NS = NOT_ACTIVATED). ADEPT$_{flex}$ supports different policies for dealing with uncompleted tasks, preceding the node n_{target} in the flow of control, when such a jump operation is performed:

$$M = \{n \mid n \in \overline{pred}(n_{target}) \wedge NS^n \in \{NOT\_ACTIVATED, ACTIVATED, RUNNING\}\}$$

Tasks from this set may be aborted, omitted, or as in our example be further worked on. For the latter case, their execution may be synchronized with successors of n_{target}. In our example, all tasks from M = {D, E, F} must be completed before the node n_d = J may be triggered. An algorithm for this as well as syntactical constraints concerning the choice of the nodes n_{target} and n_d are presented in [Hen97].

Finally, changes to premodelled task sequences may lead to an invalid data flow schema, if no further precautions are made. The rules presented in section 2.3 contribute to identify such critical cases and to provide corresponding exception handlers. Due to lack of space this aspect cannot be discussed here.

We conclude that more user-friendly operations providing elegant and efficient operational support for dynamic structural changes can be based upon the presented operations.

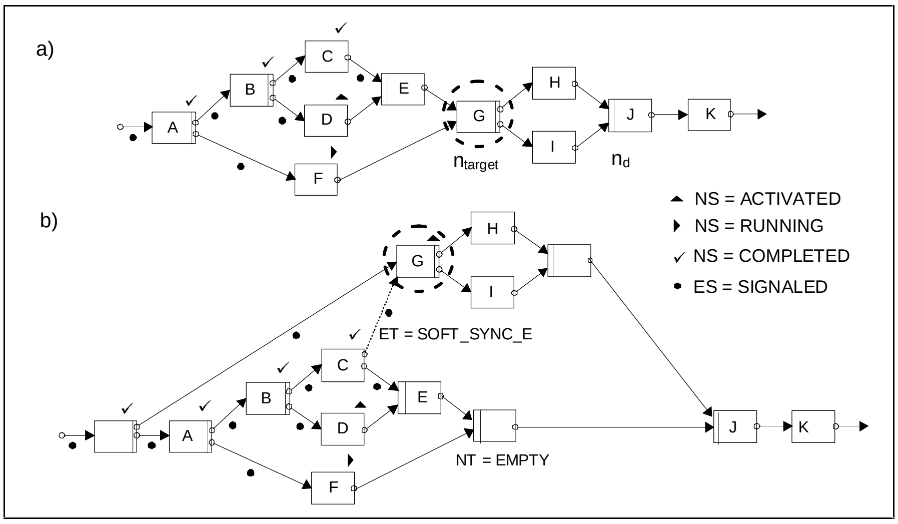


Figure 11: Parallelization of tasks that were previously constrained to be executed serially due to a *jump forward operation*.

4 Management of Dynamic Structural Changes

Many instances of a specific process type may be active at the same time. As multiple changes of different kinds may be applied to the individual WFs during their execution, several aspects must be considered:

- WF instances of the same type (i.e., the same starting schema) may have to be represented by different process graphs
- changes applied to an individual WF may depend on previously made changes
- intended changes of a WF may require concomitant changes in order to preserve the correctness and the consistency of the process graph (cmp. section 3); the necessary modifications of the graph must be carried out within the same change transaction in order to allow forward recovery in the case of failures
- changes on a process graph have to be undone under certain conditions

With respect to the management of dynamic structural changes it makes a big difference whether a performed modification must be preserved until the completion of a WF (*permanent change*) or is only of temporary nature (*temporary change*). This division is particularly important for the management of long-running processes, where changes may affect regions which may be entered multiple times, e.g., due to loop iterations or the partial rollback of a WF. When a task is inserted into the body of a loop, for instance, it must be specified whether this insertion is only valid for the current iteration of the loop or for following iterations, too. In the first case, the inserted task is executed at most once. The change must be undone (i.e., the inserted task must be removed together with its associated data links and services) before the loop can enter its next iteration. For the rest of this section, we simplistically assume that the durability of a change – temporary or permanent – is specified at the time it is applied to the WF[11].

Management of Changes

Ideally, the undoing of temporary changes and the necessary adjustments of the process graph should be completely handled at the system level without costly user interactions. To achieve this, the runtime system must have precise information about previously made changes. In our approach, we maintain the following information for each WF instance p_i.

- a process graph P_{all} reflecting the current *structure* (i.e., the schema) and the current *state* of p_i. P_{all} considers all changes applied to the WF – temporary as well as permanent ones.
- a process graph P_{perm} which has resulted from the application of permanent changes to the starting schema of p_i. Temporary changes and the state of p_i are not considered by this graph.

[11] For clinical processes [cmp. Mey96], for example, there are situations in which a change may have to be preserved until a user explicitly decides to undo it. Such cases can be simply realized by some extensions to the approach presented in the following and are therefore not further considered in this paper.

- a *change history* C which is used analogously to the WF history. It records data on the changes applied to p_i in a chronologically ordered list. These data may later be used to undo changes. Each list entry contains the following information: (1) the *type* of the change operation (including its call parameters), e.g., insertion of a task; (2) the *durability* of the change (temporary vs. permanent); (3) the *initiator* of the change; (4) the *start region* of the change, i.e., a set of nodes that may be used by the runtime system to decide whether the (temporary) change must be undone or not when a backward operation is applied (see below); (5) the list of *concomitant modifications*, e.g., addition of auxiliary services or cascading deletion of data-dependent tasks (cmp. section 3); (6) the list of *change primitives* (and their input parameters) which were applied to perform the change – each change operation is mapped to a set of graph modifications primitives such as the addition or deletion of individual nodes, edges, data elements and data links[12].

The execution of a WF is based upon the graph P_{all}. Note, that this process graph must be kept for each individual WF as its original schema may have been instantiated several times and different kinds of changes may have been performed on the WF instances. We require the additional graph P_{perm} to verify whether an intended permanent change can be handled in a proper and secure manner. If this verification was solely based on the graph P_{all}, dependencies on temporary changes as well as on the current state of p_i would be inevitable, which in turn would complicate the correct undoing of temporary changes without affecting permanent ones.

Applying Temporary and Permanent Changes

The introduction of temporary and permanent changes to a WF instance p_i requires different procedures. To perform a *temporary change* c_t we must check whether it can be applied to the process graph P_{all} while maintaining its correctness and consistency (cmp. section 3). If unresolvable exceptions occur the change operation is aborted. Otherwise c_t is performed on P_{all} and a corresponding entry is added to the change history C. Note, that a temporary change may be based upon previously made temporary as well as permanent changes. In addition, it may consider the state of the WF (cmp. section 3.1).

The introduction of a *permanent change* c_p requires additional checks. First of all, we must verify that the application of c_p to the process graph P_{perm} does not violate the correctness of this graph. Note that in contrast to temporary changes, this verification is performed inde-pendently from the state of p_i as well as from temporarily applied changes. Otherwise c_p may be based on wrong assumptions, which may cause severe problems when the state of p_i is reset or a temporary change is undone. For example, the insertion of the task X shown in figure 10 makes use of a previously made routing decision and can therefore be only introduced as a temporary change. If c_p can be applied to P_{perm} we must check its applicability to the graph P_{all}, too. If both checks are successful, c_p is applied to P_{all} as well as to P_{perm} and a corresponding entry is added to C.

[12] Note that these primitives modify the sets N, E, S, D and DF. Generally, the application of an individual graph modification primitive does not preserve the syntactical correctness and the consistency of the process graph.

Undoing Temporary Changes

Up to now we have only described how changes are managed and how they are put into effect. In the following we sketch the basic steps which become necessary for undoing temporary changes (i.e., to remove them from the process graph P_{all}) when the control of the WF is passed back to a previous task $n_{restart}$. Due to lack of space we will restrict our considerations to the dynamic insertion and deletion of tasks (cmp. section 3) and to their undoing.

Basic to the decision which changes must be undone or not are the *start regions* which are kept with each entry of the change history C. The start region of an insert operation is defined by the set M_{before} (cmp. section 3.1), whereas the start region of a delete operation consists of the empty node replacing the removed task (cmp. section 3.2). For simplification, we require that a temporary change must be undone if each node of its start region is in a finite state (NS $\in$ {COMPLETED, SKIPPED, FAILED}) and is contained within the backward region. The backward region comprises those nodes from the graph P_{all} whose state must be reset due to the backward operation. In case of a loop iteration it corresponds to the nodes of the loop body (cmp. section 2.2) whereas the backward region of a rollback operation comprises those successors of $n_{restart}$ which are in a state different from NOT_ACTIVATED (cmp. section 2.1).

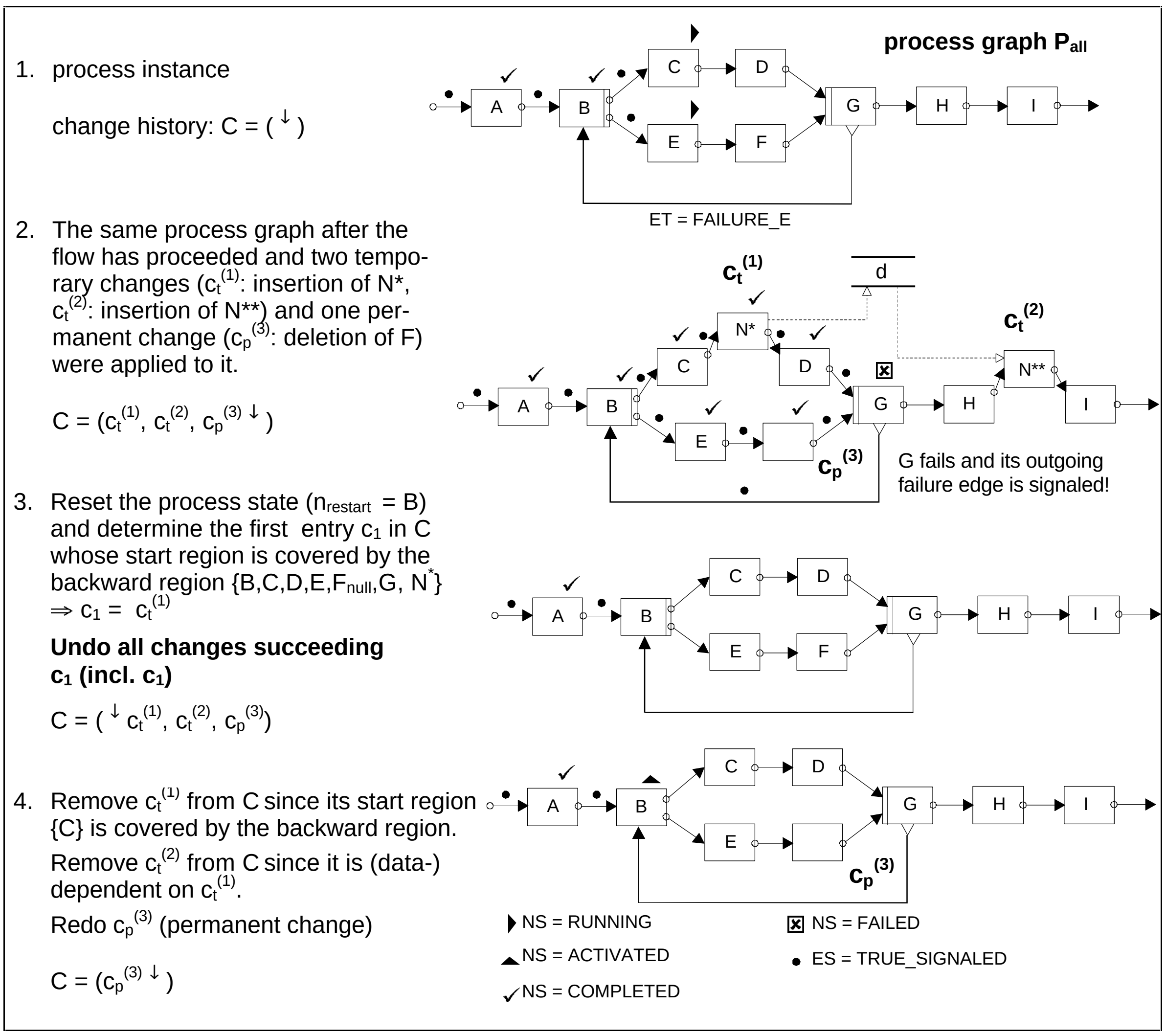
There is no problem to find the corresponding entries in the change history C and to undo the modifications associated with them. However, this simple approach would not yield to a satisfactory solution as other temporary changes may exist, which have been based upon these modifications and are therefore dependent on them, but whose start region is not covered by the backward region. These dependent changes must be undone, too, in order to preserve the correctness of P_{all}. Note, that dependencies between temporary changes are rather normal and may be explicitly desired by users. They therefore must be considered when temporary changes are undone. With this in mind and based on the assumptions made, the following steps must be performed when a backward operation is applied:

1. Find the first (i.e., the oldest) entry c_1 in C that must be undone due to the backward operation (i.e., whose start region is covered by the backward region). If no such entry appears in C then omit the following two steps.
2. Traverse C in inverse order (i.e., beginning with the latest change) until c_1 is reached. For each visited entry remove the corresponding change (temporary as well as permanent) from the process graph P_{all} – a change is removed by undoing the previously applied modification primitives in reversed order.
3. Now traverse C in forward direction (beginning with c_1). If a visited entry e corresponds to a permanent change, we reapply it to P_{all}[13]. In case of a temporary change, first of all, we check whether its start region is covered by the backward region. If this is the case, the

[13] There are rare cases in which it is not possible to redo a permanent change. Due to lack of space we do not discuss this aspect here.

change is not redone and e is removed from C[14]. Otherwise we try to redo the change on P_{all} by making use of the information stored with the corresponding change entry (incl. information on concomitant changes). If the correctness and the consistency of P_{all} cannot be preserved (e.g., due to dependencies on other removed changes), however, the redo will not be performed and the initiator of the change will be informed.

The example depicted in figure 12 illustrates the principle feasability of our approach. We omit implementation issues and possible optimizations of the presented algorithm here.



1. process instance

   change history: C = ($\downarrow$)

2. The same process graph after the flow has proceeded and two temporary changes ($c_t^{(1)}$: insertion of N*, $c_t^{(2)}$: insertion of N**) and one permanent change ($c_p^{(3)}$: deletion of F) were applied to it.

   C = ($c_t^{(1)}$, $c_t^{(2)}$, $c_p^{(3)} \downarrow$)

3. Reset the process state ($n_{restart}$ = B) and determine the first entry c_1 in C whose start region is covered by the backward region {B,C,D,E,F_{null},G, N^*}
   $\Rightarrow c_1 = c_t^{(1)}$

   Undo all changes succeeding c_1 (incl. c_1)

   C = ($\downarrow$ $c_t^{(1)}$, $c_t^{(2)}$, $c_p^{(3)}$)

4. Remove $c_t^{(1)}$ from C since its start region {C} is covered by the backward region.
   Remove $c_t^{(2)}$ from C since it is (data-) dependent on $c_t^{(1)}$.
   Redo $c_p^{(3)}$ (permanent change)

   C = ($c_p^{(3)} \downarrow$)

G fails and its outgoing failure edge is signaled!

Figure 12: Undoing temporary changes after a failure edge has been signaled [cmp. Hen97]: in step 2 the nodes N* and N** (together with the data element d and corresponding data links) were temporarily inserted. Afterwards the reduction rules presented in section 3 were applied. In addition, node F was permanently removed from the graph. Step 4 shows the resulting graph after the undoing of changes. Although its start region {H} is not covered by the backward region, the change $c_t^{(2)}$ is undone, too, since it is dependent on $c_t^{(1)}$.The entries of the change history C preceding the arc $\downarrow$ correspond to the changes currently applied to the graph P_{all}.

The assumptions made in this section may be relaxed. For example, in some cases it may be desirable to preserve a temporary change when a rollback operation is applied, but to undo it if a loop encounters a new iteration. Different policies for handling such cases and

[14] The initiator of a temporary change will be informed when the change is undone due to the partial rollback of the process.

their implemenation are described in [Rei97]. As a last interesting aspect, the availability of a change history contributes to increase the user friendliness of the system, since changes – temporary as well as permanent – can be simply undone (*UNDO of structural changes*) by the user (e.g., the initiator of the change). This should only be possible, however, if the change has not yet influenced the execution of the WF and no further changes have been based upon it.

The issues addressed in this section indicate that dynamic structural changes make great demands on the runtime system and on the management of performed changes. There are a variety of other important issues not addressed by this paper. In a multi-user environment, for example, simultaneous changes on processes must be supported in an efficient and reliable manner, too. Process participants not actively involved in a change operation should not be disturbed in their work. Errors or exceptions arising from change operations must be avoided resp. locally handled. Finally, dynamic changes must be performed at the minimum cost to application designers and programmers.

5 Related Work

It is widely recognized that state-of-the-art WF technology does not provide a sufficient level of flexibility and reliability to their users regarding exception handling and dynamic structural changes of WFs [BaWa95, BlNu95, ElNu93, EKR95, Mey96, SGJ96, Sie96]. Both, in research and in commercial WFMSs, several directions can be made out which try to overcome these limitations. These approaches focus on

- extensions of WF technology with services for exception handling, for dynamic structural changes and for the support of ad hoc resp. dynamically evolving WFs
- the integration of WFMSs with groupware technology to combine formal and well-structured processes with informal group processes
- WF schema evolution, i.e., the support of WF designers in modifying the schema of a WF and in propagating the applied changes to WFs that started with the old schema

Exception Handling and Dynamic Structural Changes in WFMSs

We are mainly interested in process-oriented WFMSs (as opposed to e.g., Lotus Notes or Groupflow [NaOt96]) as we believe that the functionality offered by these systems is needed for the development and the operational support of complex, long-running business applications. Many systems from this category (e.g., FlowMark [LeAl94], DOMINO [KHK91]), however, do only address a small part of the issues discussed in this paper. Although most of them allow online modification of task and staff definitions or the exchange of program modules during WF execution, they are rather weak with respect to exception handling and dynamic structural changes.

Several approaches exist which provide support to users regarding these issues. The proposals made by ProMInanD [VoEr92], ObjectFlow [HsKl96], WIDE [CGP97], MOBILE [Hei96], and maybe others are worth mentioning. ProMInanD is a representative of WFMSs based on the object migration model [e.g., KRW90]. A process together with its execution graph is regarded as an object ("electronic circulation folders") which is sent from user to user according to the modelled flow of control. Only the user who is currently in charge of the folder may deviate from the course of execution, e.g., by adding intermediate tasks, by skipping a task or by sending the "folder" back to a previously involved user. This restriction already limits the applicability of this model, as (authorized) users, which are not currently involved in the processing of a task, cannot intervene in the control of the WF. Another potential disadvantage of systems from this category is the simplicity of the used WF model (e.g., parallel and iterative executions are not explicitly supported by the model) and the lack of a clear theoretical basis. Furthermore, the restructuring of processes only considers the flow of control, but ignores other important aspects of a WF. The flow of data is limited to the exchange of files between tasks, so that the runtime system has minimal control over it. This, in turn, leaves significant complexity to the application programmer who himself has to adjust the flow of data when the WF is restructured.

A comparable functionality is offered by ObjectFlow [HsKl96]. Users may temporarily change the course of execution (e.g., fast forward the progress of the flow) or dynamically add intermediate tasks to a WF. In addition, ObjectFlow supports dynamic tasks, i.e., the multiple concurrent instantiations of the same task types at specific points of a predefined WF. A limited mechanism for exception handling is offered: the actions which are necessary to handle abnormal events have to be modelled as additional paths in the process graph. When a user detects an exception he must abort active tasks and modify the flow structure to transfer the control to the exception handling path. The premodeling of exceptional actions within the same model, however, might lead to an exponential explosion of the WF specification as the offered language constructs are not high-level [cmp. ElNu93]. Both approaches, ProMInanD and ObjectFlow, share the problem that the semantics of the offered change facilities is by far not sufficient when compared to our approach. A new task, for example, can only be added upon completion of a process step and by the user who has worked on this step. The inserted task must terminate before the normal flow is allowed to proceed. While this semantics might be sufficient for office environments, it is not adequate regarding the support of ad hoc resp. dynamically evolving WFs [cmp. section 3].

The WIDE WF model offers a trigger-based approach for the handling of exceptions. With each individual task as well as with the WF itself a set of exception handlers can be associated [CGP97]. An exception handler is triggered by the system at the occurence of a specific event such as the cancellation resp. rejection of a task or the break of the normal flow when a user jumps forward / backward in the process graph. In contrast to ObjectFlow and ProMInanD, the WF may proceed while the exception is handled. For each type of exception WIDE provides a simple default handler (e.g., to notify users or to cancel tasks resp. the WF) which may be overwritten by the WF programmer. However, no strategy is offered to programmers with respect to the implementation of such handlers and the maintenance of the consistency and correctness of the WF. So the responsibility for the avoidance of consistency problems and errors is shifted to application programmers which, in turn, complicates application development and may introduce new errors and exceptions into the model. Furthermore, WIDE does not allow end users to add (resp. delete) tasks to (from) a WF during its execution, which limits the applicability of this model significantly. Like ObjectFlow, WIDE supports dynamic tasks [CCP95].

A more competitive approach is offered by the MOBILE workflow model [Hei96], where the designer of a prescriptive WF may include templates for ad hoc WFs at predefined points (i.e., nodes) of the flow. Such a template resp. ad hoc WF is described in terms of goals as well as partially defined process patterns, which may be refined during runtime. Dynamic changes to the schema of a WF are restricted to the process regions representing ad hoc WFs. The authors, however, give no idea in which way users are supported in specifiying resp. refining ad hoc WFs and which operations are available.

In summary, none of these proposals is complete with respect to the approach presented in this paper. In almost all these systems the internal WF specification is based on an "ad hoc model" which lacks a clear theoretical basis. This makes it impossible to formally reason about the correctness of WF specifications and about dynamic structural changes applied to them. A notable exception is the WF model used by ObjectFlow. This system is based on a constrained petri net model, which is comparable to the ADEPT model. But like the other proposals, ObjectFlow does not provide the required level of flexibility to users. Furthermore, none of the presented approaches deals with issues regarding the undoing of changes or the support for different types of changes with respect to their durability.

The same holds for *transactional WFs* whose emphasis and strength lie in different areas such as reliability or (forward) recovery of individual processes in the presence of failures [AAE96, Att93, DHL90, Hsu93, Hsu95, WoSh97]. Transactional WFs apply some concepts of advanced transaction models (ATM) [Elm92] and are therefore pretty good in handling task failures or abnormally terminated WFs [e.g., EdLi95]. Extensions of these models like "spheres of compensation" [Ley95, Dav78] will further contribute to simplify and to fasten application development and to make the resulting applications more reliable. However, transactional WFs do only meet a small part of the flexibility issues addressed in this paper [AAE96, KaRa95, WoSh97], especially regarding dynamic structural changes. To support sophisticated exception handling policies and ad hoc modifications requires the involvement of humans with the runtime system, as the WF engine will generally not have the knowledge to detect and to handle all possible failures and exceptions alone [AAE96, StMi95, Mey96]. Besides this, advanced transactions models must be closely integrated with facilities needed for the management and the undoing of changes.

Integration of WF Technology with Groupware Approaches

Several proposals have been made to combine formal and well-structured processes with informal group processes. Communication-oriented models are based on a speech act conversation model [WiFl86], which reduces organizational processes to networks of commitment loops between process participants. Other approaches follow goal-based models [BlNu95] and use circulation folders [KRW90]. All these approaches share the dis-advantage that the achieved flexibility is paid by a harder formalization of even simple, repetitive processes.

Other research groups try to combine the advantages offered by WF technology with those of groupware systems by supporting unstructured work at specific points of a WF [e.g., AGS95, BlNu95, WPS97]. The conditions under which an unstructured (group) task is executed may therefore be tightly specified. Details of the work to be done, however, are only described in terms of goals or guidelines. This approach can be used in combination with our model. Addressed research issues include the integration of WFMSs with groupware technology, the formal specification of unstructured activities and the management of the contextual information associated with them [BlNu95, WPS97]. Although these proposals offer an important contribution towards more flexible systems, they only address a small part of the flexibility requirements discussed in this paper. To be broadly applicable, process-centered WFMSs cannot afford to restrict their support to well-structured processes, including some unstructured tasks [SGJ96]. Several authors doubt the suitability of this integration approach at all [Hei96, Sie96]. As a potential disadvantage they consider the "break" between structured and unstructured parts of work, which results from the combined use of workflow with groupware technology. Important features such as auditing, rollback, security or consistency may be lost when (unstructured group) tasks are not controlled by the WFMS.

WF Schema Evolution

There are only few approaches which address correctness issues regarding dynamic structural changes. Notable exceptions come from Ellis et al. [EKR95], Casati et al. [CCP95, CCP96], and maybe others. In contrast to our proposal which concentrates on ad hoc changes applied to specific WF instances, these approaches deal with *changes at the schema level* and their propagation to WFs whose execution started with the old schema [CCP96, EKR95]. Although the operational support for both types of changes is a complex and yet unsolved problem and many related issues can be identified, in some respects ad hoc modifications are much more intricate and problematic, as they may have to be performed by non-computer experts.

As ADEPT$_{flex}$, both approaches are based on a conceptual WF model. However they restrict their considerations to dynamic changes of the control flow while other relevant aspects are left aside. Ellis et al. propose a mathematical model for the formal reasoning about certain classes of dynamic changes [EKR95]. This model is based on constrained petri nets. Simplistically a change corresponds to the replacement of a subnet ("old change region") of the process graph by a new subnet ("new change region"), and is said to be correct if after its application corresponding WF instances can either be executed according to the old schema or to the new one. The emphasis and strength of this approach lies in its formal foundation. Implementation issues and issues related to the operational support for dynamic changes are not addressed. Furthermore no statement is made about changes that cannot meet the defined correcntess criteria. Casati et al. address the problem of schema evolution from a static as well as a dynamic point of view [CCP96]. In contrast to Ellis et al. they go in line with our approach. Dynamic structural changes are based on a set of modification primitives, whose application does not violate the given correctness criteria. The proposed change primitives, however, offer only a limited semantics when compared to our approach. The strength rather lies in the variety of policies offered for managing the evolution of running WF instances (including support for version management). Formal criteria are introduced in order to determine which processes can be transparently migrated to the new version.

How to integrate dynamic structural changes at the schema level with changes at the instance level is an outstanding research issue. When looking at the proposals made by Ellis et al., for example, it is implicitly assumed that the execution of all instances of a specific process type is based on the same net. This assumption, however, cannot be maintained when ad hoc changes to individual processes must be considered, too. The proposals made in section 4 are a first step towards a solution of this problem.

In summary, today's WFMSs offer a promising perspective, but are not yet able to fully support the flexibility requirements of their users. The role of application developers as well as of end users in handling exceptions and changing the structure of processes during runtime is not well-understood and therefore poorly integrated with today's WFMSs.

6 Summary and Outlook

In this paper we have concentrated on issues regarding dynamic structural changes of WFs during their execution. We have argued that such changes are rather the norm in computerized processes and that their operational support will form a key part of process flexibility in future WFMSs. We have shown that the dynamic change problem has many facets and is therefore a worthwhile area of study.

We have introduced the basic concepts of the ADEPT workflow model and demonstrated its suitability for the (precise) specification of WFs, the verification and testing of the correctness of WF specifications, and the enactment of WFs. We have argued that the ADEPT model offers a good compromise for the trade-off between the expressive power of a WF model and the complexity of the needed algorithms for model checking, especially when contrasting it with general-purpose models such as petri nets. We believe that this is crucial for the efficient support of complex dynamic structural changes.

The ADEPT$_{flex}$ model which is based upon ADEPT has been presented and its *adequacy* with respect to dynamic structural changes has been demonstrated. ADEPT$_{flex}$ comprises a *complete* and *minimal* set of change operations, which ensure the *correctness* and *consistency* of the resulting process graph *by construction*. Taking the dynamic addition of tasks as an example, we have demonstrated that the underlying correctness properties of the ADEPT model and the set of preconditions defined for each type of change operation constitute a good basis for this. We discussed how to deal with changes that cannot meet the correctness criteria. We believe that neither hard-wired mechanisms nor hand-made solutions would be satisfactory in practice. Instead we have proposed a more flexible approach, offering several policies for dealing with the exceptions resulting from a change (e.g., missing or incomplete input data of tasks). We compared our model with other WF models and we did show that the semantics offered by the change facilities of ADEPT$_{flex}$ captures those of other models by far. Finally, we addressed issues regarding the management of temporary and permanent changes and the undoing of temporary changes when a backward operation is applied. This has turned out to be quite important in our experience. Several prototypical implementations have been built to demonstrate the feasability of our approach [Bla96, Gri97, Hen97, Kir96].

The work presented in this paper has been well-motivated by a variety of organizational studies and analyses of processes from the clinical domain [Mey96, RSD97, KRN94] where ad hoc changes as well as dynamically evolving WFs are rather the norm and exceptions do frequently occur. We also implemented complex processes from the university's women's hospital by applying current WF technology [RSD97]. As a result, today's WFMSs offer perspectives, but are far away from providing the flexibility needed by clinical users.

For the future, however, we believe that WF technology has the potential to lead to a completely different kind of application programming. The development of even complex distributed application systems may reduce to the reuse of premodelled process templates from a template repository, the customization of these templates and the insertion of the application components in the style of plug-and-play. To be broadly applicable, however, future WF technology must provide a high flexibility in user assistance and more human-centric approaches that include an integral support for exception handling and dynamic structural changes.

Although some progress has been achieved, a lot has to be done. Besides the topics addressed in this paper, some specific areas that warrant further attention are

- the support of simultaneous changes on individual processes
- the application of dynamic changes to WFs whose schema is decomposed into several parts that may be kept resp. controlled by different WF servers [e.g., BaDa97, WoWe97]
- the "intelligent" support of WF ensembles, i.e., dynamically evolving collections of more or less loosely coupled WFs. The requirements which can be identified here are far more challenging than those faced by concurrency control in standard database technology [HeDa97].
- the development of general concepts for the integration of dynamic structural changes at the schema level [e.g., CCP96, EKR96] with changes at the instance level (as proposed in this paper)
- the provision of "intelligent" interfaces for application programmers as well as end users; adding only functionality to current models and systems without understanding how the programmer resp. the end user will be able to utilise it will certainly not be very helpful!

We believe that dynamic WFs are a field that would benefit by more intense study by the research community.

References

[AAE96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, C. Mohan. *Advanced Transaction Models in Workflow Contexts*. Proc. 12th Int'l Conf on Data Engineering, New Orleans, February 1996.

[AGS95] P. Antunes, N. Guimaraes, J. Segovia, J. Cardenosa: *Beyond Formal Processes: Augmenting Workflow with Group Interaction Techniques*. Proc. Conf. on Organizational Comp Sys, (COOCS'95), 1995.

[Att93] P. C. Attie, M. P. Singh, A. Sheth, M. Rusinkiewicz: *Specifying and Enforcing Intertask Dependencies.* Proc. 19th Int'l Conf on Very Large Databases (VLDB'93), Dublin, August 1993, pp. 134-145.

[BaWa95] P. Barthelmess, J. Wainer: *Workflow Systems: a few Definitions and a few Suggestions*. Proc. Conf on Organizational Computing Systems, (COOCS'95), 1995, pp. 138 - 147.

[BaDa97] T. Bauer, P. Dadam: *A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration*. Technical Report No. 97–03, Department for Computer Science, University of Ulm, Germany, 1997.

[Bla96] R. Blaser: *Composing Processes by the Reuse of Application Components*. (in German). Master's thesis, University of Ulm, Germany, October 1996.

[BlNu95] R. Blumenthal, G. J. Nutt: *Supporting Unstructured Workflow Activities in the Bramble ICN System*. Proc. Conf on Organizational Computing Systems (COOCS'95), 1995, pp. 130 - 137

[CCP95] F.Casati, S. Ceri, B. Pernici, G. Pozzi: *Conceptual Modeling of WorkFlows*. Proc. 14th Int'l Conf Object-Oriented and Entity-Relationship Approach, GoldCoast, Australia, 1995, pp. 341 - 354.

[CCP96] F. Casati, S. Ceri, B. Pernici, G. Pozzi: *Workflow Evolution*. Proc. 15th Int'l Conf. on Conceptual Modeling (ER '96), Cottbus, Germany, 1996, pp. 438-455

[CGP97] G. Casati, P. Grefen, B. Pernici, G. Pozzi, G. Sánchez: *WIDE Workflow Model and Architecture.* Technical Report, University of Milano, Italy, 1997.

[CKO92] B. Curtis, M. Kellner, J. Over: *Process Management*. Comm. of the ACM, Vol. 35, No. 9, 1992.

[DHL90] U. Dayal, M. Hsu, R. Ladin: *Organizing Long-Running Activities With Triggers and Transactions,* Proc ACM SIGMOD Int'l Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 204-214.

[Dav78] C. T. Davis Jr.: *Data Processing Spheres of Control*. IBM Sys Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[DGS94] G. Dinkhoff, V. Gruhn, A. Saalmann, M. Zielanko: *Business Process Modelling in the Workflow Management Environment LEU*. Proc. 13th Int'l Conf. on the Entity-Relationship Approach, Manchester, 1994 (LNCS 881, Springer), pp. 46-63

[EdLi95] J. Eder, W. Liebhart: *The Workflow Activity Model WAMO*. Proc. 3rd Int'l Conference on Cooperative Information Systems, Vienna, Austria, 1995, pp. 87 - 98

[ElNu93] C. A. Ellis, G. J. Nutt: *Modeling and Enactment of Workflow Systems*, Proc. 14th Int'l Conf. on Application and Theory of Petri Nets, Chicago, 1993, (LNCS 691, Springer), pp. 1-16

[EKR95] C. A. Ellis, K. Keddara, G. Rozenberg: *Dynamic Change Within Workflow Systems*. Proc. Conf. on Organizational Computing Systems (COOCS'95), 1995, pp. 10 - 21

[Elm92] A. K. Elmargarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992

[GHS95] D. Georgakopoulos, M. Hornick, A. Sheth: *An Overview of Workflow Management.* Distributed and Parallel Databases, Vol. 3, 1995, pp. 119-153

[Gri97] M. Grimm: *ADEPT$_{time}$ – Dealing With Temporal Dependencies in Flexible WFMSs (in German).* Master's thesis, University of Ulm, Germany, To appear in 1997.

[Hei96] P. Heinl, H. Schuster, K. Stein: *Behandlung von Ad-hoc-Workflows im MOBILE Workflow-Modell.* In: Proc. Softwaretechnik in Automation und Kommunikation - Rechnergestützte Teamarbeit (STAK'96), Munich, March 1996, pp. 229 - 242.

[HeDa97] C. Heinlein, P. Dadam: *Interaction Expressions – A Powerful Formalism for Describing Inter-Workflow Dependencies.* Technical Report No. 97–04, Department for Computer Science, University of Ulm, Germany, 1997.

[Henn89] M. Hennessy: *Algebraic Theory of Processes*. The MIT Press, Cambridge, MA, 1989.

[Hen97] C. Hensinger: ADEPT$_{flex}$ – *Dynamic Modification of Workflows and Exception Handling in WFMSs (in German).* Master's thesis, University of Ulm, Germany, January 1997.

[HOR96] A. Hofstede, M. Orlowska, J. Rajapaks: *Verification Problems in Conceptual Workflow Specifications*. Proc. 15th Int'l Conf. on Conceptual Modeling, Cottbus, Germany, 1996, pp. 73-88.

[HsKl96] M. Hsu, C. Kleissner: *ObjectFlow: Towards a Process Management Infrastructure*. Distributed and Parallel Databases, Vol. 4*,* 1996, Kluwer Academic Publishers, pp. 169-194

[Hsu93] M. Hsu (ed.): *Special Issue on Workflow and Extended Transaction Systems*. IEEE Bulletin of the Technical Commitee on Data Engineering, Vol. 16, No. 2, 1993

[Hsu95] M. Hsu (ed.): *Special Issue on Workflow Systems*. IEEE Bulletin of the Technical Commitee on Data Engineering, Vol. 18, No. 1, 1995

[KaRa95] M. Kamath, K. Ramamritham: *Bridging the Gap Between Transaction Management and Workflow Management*. Technical Report, University of Massachusetts, 1995

[KRW90] B. Karbe, N. Ramsperger, P. Weiss*: Support of Cooperative Work by Electronic Circulation Folders*, Conf. on Office Information Systems, Cambridge, Mass., 1990, SIGOIS Bulletin, Vol. 11, No. 2,3, pp. 109-117

[Kir96] M. Kirsch: *Design and Implementation of a Graphical Tool for the Modeling and Animation of Flexible Workflows (in German).* Master's thesis, University of Ulm, Germany, 1996.

[KoRe96] I. Konyen, M. Reichert: *Organizational Aspects of Computerized Clinical Processes - Required Concepts, Integrity Rules and Transformation in WFMSs* (in German), Technical Report, Department of Databases and Information Systems, University of Ulm, Germany, 1996.

[KHK91] T. Kreifelts, E. Hinrichs, K.-H. Klein, P. Seuffert, G. Woetzel: *Experiences with the DOMINO Office Procedure System*. Proc. 2nd European Conf on CSCW (ECSCW'91), Amsterdam, The Netherlands, September 1991, pp. 117 - 130.

[KRN94] K. Kuhn, M. Reichert, M. Nathe, T. Beuter, P. Dadam: *An Infrastructure for Cooperation and Communication in an Advanced Clinical Information System*. Proc. 18th Symp on Comp in Med Care, Washington, 1994, 519 - 523.

[LeAl94] F. Leymann, W. Altenhuber: *Managing Business Processes as an Information Resource*. IBM Systems Journal, Vol. 33, No. 2, 1994, pp. 326-348

[Ley95] F. Leymann: *Supporting Business Transactions via Partial Recovery in Workflow Management Systems*, Proc. Datenbanksysteme in Büro, Technik und Wissenschaft , Dresden, Germany, 1995, pp. 51-70

[MaPn92] Z. Manna; A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems-Specification*. Springer, 1992.

[Mey96] J. Meyer: *Requirements for Future WFMSs: Flexibility, Exception Handling and Dynamic Changes in Clinical Processes*. Master's thesis, University of Ulm, Germany, January 1996.

[NaOt96] L. Nastansky, M. Ott : *Teambasiertes Workflowmanagement und Analyse prozeßorientierter Teamarbeit im Bereich zwischen kooperativer und strukturierter Vorgangsbearbeitung.*Technical Report, Workgroup Computing Competence Center Paderborn, University of Paderborn, Germany, 1996

[Rein93] B. Reinwald: *Workflow-Management in Verteilten Systemen.* Teubner, 1993.

[Rei97] M. Reichert: *A Conceptual and Operational Framework for Supporting Dynamic Structural Changes of Workflows in WFMSs (in German).* Phd thesis in preparation, University of Ulm, Germany, 1997.

[RSD97] M. Reichert, B. Schultheiß, P. Dadam: *Experiences with the Development of Process-centered, Clinical Application Systems Using Process-oriented Workflow Technology (in German),* submitted for publication, 1997.

[ReSc95] A. Reuter, F. Schwenkreis: *ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems*. In: [Hsu95], 1995, pp. 4-10.

[SGJ96] A. Sheth, D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, A. Wolf: *Report from the NSF Workshop on Workflow and Process Automation in Information Systems*, Technical Report UGA-CS-TR-96-003, University of Georgia, October 1996.

[Sie96] R. Siebert: *Adaptive Workflow for the German Public Administration*. Proc. 1st Int'l Conf on Practical Aspects of Knowledge Management (PAKM'96) – Workshop on Adaptive Workflow, Basel, Switzerland, 1996.

[StMi95] D. M. Strong, S. M. Miller: *Exceptions and Exception Handling in Computerized Information Processes*. In: ACM Transactions on Information Systems*, Vol. 13, No. 2, April 1995,* ACM Press, 1995, pp. 206 - 233.

[VoEr92] P. Vogel, R. Erfle: *Backtracking Office Procedures*. Proc. 15th Int'l Conf. on Database and Expert Systems (DEXA '92), Valencia, Spain, 1992, pp. 506-511

[WPS97] M. Weber, G. Partsch, A. Scheller-Huoy, J. Schweitzer, G. Schneider: *Flexible Real-time Meeting Support for Workflow Management Systems*, Proc. 30th Hawaii Int'l Conf. on System Sciences HICSS, Maui, Hawaii, 1997

[WiFl86] T. Winograd, F. Flores: *Understanding Computers and Cognition: A New Foundation For Design*, Ablex Publishing Corporation, Norwood, NJ, 1986

[WoWe97] D. Wodtke, G. Weikum: *A Formal Foundation for Distributed Workflow Execution Based on State Charts*. Proc. Int'l Conf. on Database Theory, Delphi, Greece, January 1997

[WoSh97] D. Worah, A. Sheth: *Transactions in Transactional Workflows*. In: S. Jajodia, L. Kerschberg (eds.): Advanced Transaction Models and Architecturres, Kluwer Publ, to appear in 1997