Journal of Intelligent Information Systems 10, 93–129 (1998)

ADEPT$_{flex}$—Supporting Dynamic Changes of Workflows Without Losing Control

MANFRED REICHERT reichert@informatik.uni-ulm.de
PETER DADAM dadam@informatik.uni-ulm.de
University of Ulm, Dept. Databases and Information Systems, D-89069 Ulm, Germany

Abstract. Today's workflow management systems (WFMS$_s$) are only applicable in a secure and safe manner if the business process (BP) to be supported is well-structured and there is no need for ad hoc deviations at run-time. As only few BPs are static in this sense, this significantly limits the applicability of current workflow (WF) technology. On the other hand, to support dynamic deviations from premodeled task sequences must not mean that the responsibility for the avoidance of consistency problems and run-time errors is now completely shifted to the (naive) end user. In this paper we present a formal foundation for the support of dynamic structural changes of running WF instances. Based upon a formal WF model (ADEPT), we define a complete and minimal set of change operations (ADEPT$_{flex}$) that support users in modifying the structure of a running WF, while maintaining its (structural) correctness and consistency. The correctness properties defined by ADEPT are used to determine whether a specific change can be applied to a given WF instance 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. We discuss basic issues with respect to the management of changes and the undoing of temporary changes at the instance level. Recently we have started the design and implementation of ADEPT$_{workflow}$, the ADEPT workflow engine, which will make use of the change facilities presented in this paper.

Keywords: workflow management, exception handling, dynamic change, adaptive workflows

1. Introduction

Process-oriented workflow management systems (WFMS$_s$) (Georgakopoulos et al., 1995; Hsu, 1995; Leymann and Altenhuber, 1994) offer a promising approach for the development of business applications that directly follow the execution logic of the underlying business process (BP). The separation of the applications control structures from the implementation of its task programs contributes to simplify and to speed up application development, and enables the run-time system to assist users in coordinating and scheduling the tasks of a BP.

Current process-oriented WFMSs are applicable in a reliable and secure manner only if the BP to be supported is well-structured and there is no need for ad hoc deviations or dynamic extensions at run-time (see, Barthelmess and Wainer, 1995; Ellis et al., 1995; Siebert, 1996; Reichert and Dadam, 1997a). As only few BPs are static in this sense, this significantly limits the benefit and the applicability of current workflow (WF) technology. As an example, consider BPs from the clinical domain (see, Reichert et al., 1996, 1997b), 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 run-time (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) (Strong and Miller, 1995; Meyer, 1996). 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, the WFMS does not always have sufficient knowledge to handle these situations alone. Instead, user involvement is required in order to resolve exceptions and to deal with unplanned events. Hence, the resulting requirements are far more challenging than those faced by standard transaction technology and advanced transaction models (Worah and Sheth, 1997; Elmargarmid, 1992).

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 as well as whole 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 reassignment 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 premodeled task sequences of a WF at run-time is a two-edge sword. On the one hand, it captures the natural freedom of process participants to work on a BP 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—make it difficult to have the system behave in a predictable and correct manner. For this reason, supporting dynamic WF changes must not mean that the responsibility for the avoidance of consistency problems or run-time errors is now completely shifted to the naive end user or to the application programmer. Instead, correctness and consistency criteria are required in order to enable the run-time system to adequately assist users in applying structural changes. That is, the system should guarantee that all consistency constraints that have been ensured prior to a dynamic change are also ensured after the the WF instance has been modified.

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 instance is restructured. Otherwise, changes such as the deletion or the addition of a task may cause severe inconsistencies (e.g., unintended lost updates) or even run-time errors (e.g., program

crashes due to the invocation of task modules with invalid or missing parameters). Changes must consider the state of the WF instance, too. For example, it should not be possible to delete a task or to change its attributes if it was already 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, or user roles, to specific WF types or regions of a WF graph (e.g., a single task), to certain states of a WF, or to any combination of them.

Normally, several instances of a specific WF type are active at the same time. As changes of different kinds may be applied to these instances during their execution, several issues must be addressed. First of all, WF instances of the same type (i.e., the same starting schema) may have to be represented by different *execution graphs*. Secondly, the run-time 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 WF is passed back to a previous point of control (e.g., when a new iteration of loop is entered). Consequently, a technical challenge is how to represent and manage these different types of changes, and how to undo temporary changes in a correct manner. This requires sophisticated mechanisms for change management and a close integration of change operations with other core services of the WFMS. Finally, changes should be made "on the fly" without loss of run-time *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 WF, which cannot be handled without extensive system support. In providing support for dynamic WF changes, whether for the process administrator or, in some form, for 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 formal foundation for the support of dynamic changes of running WF instances. We concentrate on structural changes and on related modification operations. Implementation issues, e.g., concerning the transactional execution of changes, are outside the scope of this paper. Fundamental to our approach is a conceptual, graph-based WF model (ADEPT[1]) which has a formal foundation in its syntax and (operational) semantics. Based on this model we develop a *complete and minimal set of change operations* which support users in modifying the structure of running WF instances, while preserving their *correctness and consistency* (ADEPT$_{flex}$). If a change leads to the violation of correctness properties, it is either rejected or the correctness of the WF graph (e.g., concerning the flow of data) must be restored by handling the exceptions resulting from the change (possibly leading to concomitant changes). Furthermore, we show how temporary and permanent structural changes of WF instances are managed and which precautions must be made to enable the run-time system to undo temporary changes in case of backward operations.

The contribution of this paper is demonstrating the principle feasibility of our approach and giving some insights into fundamental research issues related to dynamic WF changes. This includes the following three results:

- we demonstrate the suitability of our WF model for WF specification and for the support of dynamic structural changes,
- we show how even complex, dynamic structural changes can be applied to a WF instance during its execution and which precautions must be made to do this in a secure and correct 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 remainder of the paper we concentrate on ad hoc structural changes applied to individual WF instances. We do not explicitly consider *changes at the schema level* and their propagation to WFs whose execution started with the old schema (see Casati et al., 1996; Ellis et al., 1995). However, many of the presented concepts can also be applied to this type of change.

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.

2. Fundamentals of the ADEPT workflow model

A variety of WF description languages have been discussed in the literature. Some of them are based on formal models such as high level Petri nets (Ellis and Nutt, 1993; Ellis et al., 1995; Kreifelts et al., 1991; Leymann and Altenhuber, 1994), state- and activity-charts (Wodtke and Weikum, 1997), temporal logic (Manna and Pnueli, 1992; Attie et al., 1993), or process algebra (Hennessy, 1989). One strength of these formal approaches lies in the offered mechanisms for specifying, analyzing, and verifying the properties of static WF structures, e.g., regarding state transitions, deadlocks, or the reachability of states. Adequate mechanisms for modifying these structures at run-time, however, are missing for the most part (cf., Ellis et al., 1995). To support dynamic WF 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 the analysis of more complex WF models extremely costly (cf., Hofstede et al., 1996), which may cause a significant overhead when complex structural changes become necessary at run-time. Secondly, for the effective support of users—possibly non-computer experts—in performing dynamic changes, a WF model must allow an intuitive and structured representation of a BP, which is hard to achieve with these models.

The ADEPT model presented in this section follows a more structured approach. Essential for the specification and for the execution of WFs is the concept of *symmetrical control structures*, which is well-known from structured programming (cf., Reinwald, 1993): task sequences, branchings (with different split and join semantics), and loop backs are specified as symmetrical blocks with well-defined start and end nodes. These blocks may be arbitrarily nested, but they 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 of a WF graph. A detailed description of the ADEPT model is beyond the scope of this paper. We restrict our considerations to the basic concepts provided for the specification of the control and data flow of a WF. Other important aspects, e.g., the modeling of temporal and organizational aspects and mechanisms for their dynamic adaptation are described in (Grimm, 1997; Hensinger, 1997; and Kirsch, 1996).

2.1. Workflow modeling

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

Flow of control. We represent a WF's control flow as a directed, structured graph (N, E). 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 which we describe in the following. Each WF schema has a unique *start node* ($NT = STARTFLOW$), and it has a unique *end node* ($NT = ENDFLOW$). The start node has no predecessor, and the end node has no successor. All other nodes from N must be preceded and succeeded by at least one node. The *sequential execution* of two tasks is modeled 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 they are synchronized symmetrically at a unique join node. ADEPT supports three types of branching: *parallel processing* (AND-split/AND-join), *conditional routing* (OR-split/OR-join), and *parallel branching with final selection* (AND-split/OR-join). The routing decision of a *conditional branching* (see figure 1(b)) may either be value-based 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 these tasks

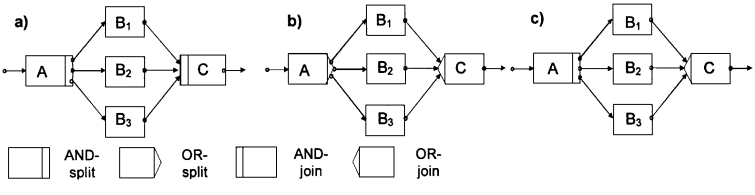

Figure 1. (a) Parallel processing, (b) conditional branching, and (c) parallel branching with final selection.

is selected for execution, the work items of the others are removed from the corresponding worklists. This allows us to model situations where several tasks are activated, but only one of them may be executed. When the split node of a *parallel branching with final selection* (see figure 1(c)) fires, all successor branches are triggered, and they may be worked on concurrently. In contrast to a parallel processing, the flow may proceed at the join node as soon as one of the branches is completed. 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. In any case, the corresponding tasks are reset in their state (see Section 2.2), and their effects on data elements of the WF (see below) are undone. As an important extension, more than one branch may be completed. In this case the "winner" must be selected by an authorized user before the flow can proceed.

Up to now we have only considered non-cyclic WF graphs. In ADEPT, the repetitive execution of a set of tasks can be modeled by the use of *loops*. Like a branching, 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 node is triggered. As we will see in Section 4, the use of loops raises some challenging issues in connection with dynamic changes. When inserting a new task into a loop's body, for instance, it must be clear whether this insertion should only be valid for the current iteration of the loop or for following iterations as well.

To take provisions for task failures already at the modeling level, ADEPT provides a second type of backward edge: a *failure edge* ($ET = FAILURE\_E$) connects a task n_{failure} with a preceding node n_{restart}. At run-time, the edge signals if the execution of the task n_{failure} fails. As a consequence, all nodes succeeding n_{restart} (incl. n_{restart}) and preceding n_{failure} (incl. n_{failure}) are reset in their state. In contrast to a loop iteration, the effects of the corresponding tasks on the data elements of the WF instance (see below) are undone. Afterwards the flow proceeds with the execution of n_{restart}. Note that the symmetrical structuring and the regular nesting do not apply to failure edges, as a task may have several outgoing failure edges, possibly linking it with nodes from different branches of a preceding parallel branching. Another restriction must be added: If the node n_{restart} is contained within a loop's body (or within a branch with OR-join), this body (branch) must also contain the node n_{failure}. As the use of failure edges is therefore not always possible in connection with these control structures, also we support the dynamic rollback of WFs. Generally, the state of a WF can be reset to an arbitrary previous state.

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

- A "soft" synchronization $n_1 \rightarrow n_2$ ($ET = SOFT\_SYNC\_E$) is used to specify a *delay dependency* between the two tasks n_1 and n_2, i.e., 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.

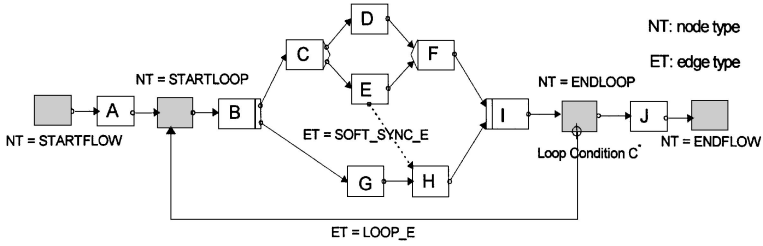

Figure 2. Example of a simple WF model.

- On the other hand, a "strict" synchronization $n_1 \rightarrow n_2$ ($ET = STRICT\_SYNC\_E$) 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 a conditional branching with tasks from a parallel branching with final selection.

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 AND-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.

Example 1. Figure 2 shows an example for the use of a soft synchronization: the task H is triggered when G is completed and E is either completed or skipped (i.e., the corresponding branch is not selected for execution).

Flow of data. The input and the output data of tasks and the flow of data between them are an important functional aspect of a WFMS. Nevertheless, the modeling of the data flow and the exchange of structured data between the tasks of a WF are often poorly supported in today's WFMSs (Sheth et al., 1996). This leaves significant complexity to application developers, and it makes it impossible to provide system support for verifying the correctness of a data flow schema or for adjusting it when structural changes are applied to a WF. In our model, the exchange of data between tasks is based on global WF variables: a WF schema 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 parameters with elements from D. For simplification, the input (output) parameters of the WF schema are logically treated as the output (input) parameters of its start (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. In order to avoid hard-wired adjustments within task modules, each task node $n \in N$ can be associated with a set of so-called auxiliary services S_n. The execution of these services is closely connected to the execution of the task. An auxiliary service $s \in S_n := S_n^{\text{prec}} \cup S_n^{\text{succ}}$ is either triggered when n is started ($s \in S_n^{\text{prec}}$) or when it is terminated ($s \in S_n^{\text{succ}}$), and it therefore does not appear as a separate work item in any worklist. Services from the set S_n^{prec} may also be used to

request incomplete or missing input data of a task from the user initiating it, which has turned out to be important in our context (see Section 3). Furthermore, a task n (also the application program associated with it) may only be executed after all services from S_n^{prec} have been successfully completed. On the other hand, if a task fails or if it is undone, the effects of its associated services on global data elements are undone as well.

Definition 1 (Data flow schema). Let (N, E) be the control flow graph of a WF schema P and let D denote a finite set of data elements associated with P. Let further $PARS(X)$ denote the set of parameters associated with the task or the service X [$PARS(X) := InPARS(X) \cup OutPARS(X)$]. A *data link df* between a parameter par^{df} and a data element d^{df} is then described by the 4-tuple:

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

with

$$d^{\text{df}} \in D, n^{\text{df}} \in N \cup S\left(S := \bigcup_{n \in N} S_n\right), par^{\text{df}}, \in PARS(n^{\text{df}}),$$

$$access\_mode^{\text{df}} \in \{read, write\}$$

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

The data links connecting service parameters with data elements from D form a key part of P's data flow schema. The intuitive meaning of a link $(d, n, p, read) \in DF$ is that the value of $p \in InPARS(n)$ is read from d when the task n is started. On the other hand, the data link $(d, n, p, write)$ expresses that the value of the output parameter $p \in OutPARS(n)$ is written into d after the successful completion of n. In Section 2.3 we introduce properties for the correctness of a data flow schema; 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.

With respect to data management we follow an approach similar to that described in (Reuter and Schwenkreis, 1995). When a task (or service) updates a data element d, its current value is not overwritten. Instead a new version is created, which may be accessed by succeeding tasks and services. This allows us to restore previous values of data elements in case of a partial rollback, and it makes it possible for tasks from different branches of a parallel processing (with OR-/AND-join) to work on different copies of the same data element d.

Example 2. An example for a simple data flow schema is depicted in figure 3 [Note, that the output parameter (input parameter) of the start node (end node) corresponds to the input parameter (output parameter) of the WF]. Assume that G has read access to the data element d_1 Although the task C may write d_1 before G is started, this value would not be visible to G. G may only access that 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 or by a service preceding it in the flow of control.

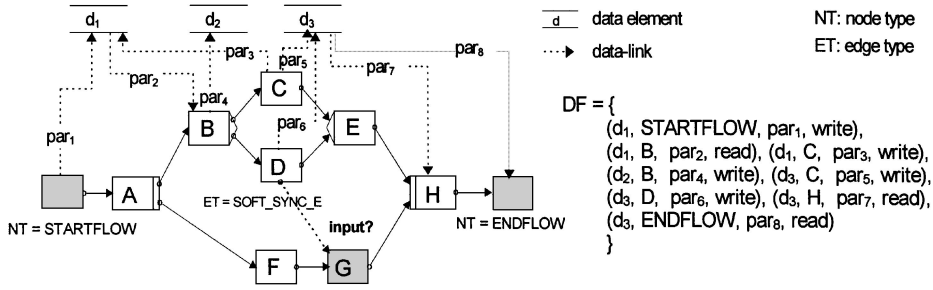

Figure 3. Example of a simple data flow schema.

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 or succeeding the execution of tasks. D denotes the set of data elements and DF defines the set of data links connecting task and service parameters with elements from D.

2.2. Workflow execution

The state of a WF instance is one of the major criteria for deciding whether a specific structural change can be applied to it or not. As an example, consider the deletion of a task which should not be allowed if the task was already completed. Furthermore, after applying structural changes to a WF graph, concomitant changes of the states of its nodes and edges may become necessary in order 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.

ADEPT is based on a well-defined operational semantics to support this. The state of a WF instance is defined by the current marking of its nodes and edges, by the values stored for its data elements (possibly in different versions), and by its execution history. The state of a single task n is described by the current marking 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 data about them. Finally, each edge e of a WF execution graph is in one of the states $ES^e \in \{NOT\_SIGNALED, FALSE\_SIGNALED, TRUE\_SIGNALED\}$. When a WF instance is created, the graph of its starting schema (N, E, S, D, DF) is initialized. The state of all nodes is set to *NOT_ACTIVATED*, and all edges are marked as *NOT_SIGNALED*. Furthermore, the WF's input data are stored in the corresponding data elements.

When the WF is started, the start node of its graph is marked as *COMPLETED*, and its outgoing control edge is set to *TRUE_SIGNALED*. Each time an edge $n_1 \rightarrow n_2$ (of arbitrary type) is marked, the state of its destination node n_2 is reevaluated according to the *execution rules* defined by ADEPT. Executions rules describe the conditions under which a node may be activated, i.e., routed to the corresponding worklists. If the

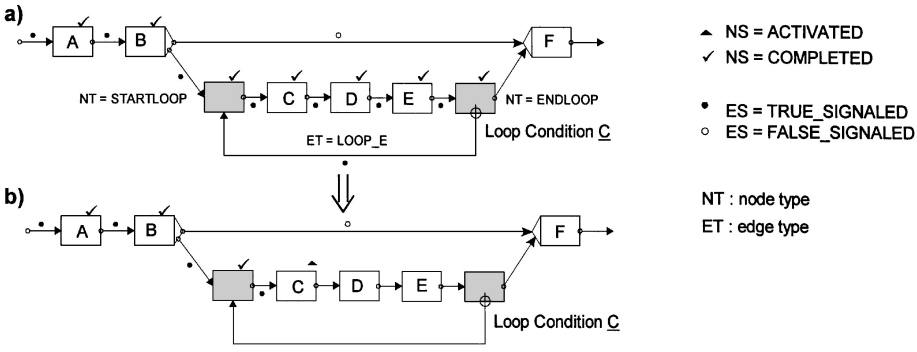

Figure 4. Application of execution and signaling rules in connection with a loop.

node n_2 corresponds to an AND-join, for instance, it is set to the state *ACTIVATED* if the following conditions are met: n_2 is marked as *NOT_ACTIVATED* and all ingoing control edges ($ET = CONTROL\_E$) are marked as *TRUE_SIGNALED*. Furthermore, all sync edges $n \rightarrow n_2, n \in N$ with $ET = STRICT\_SYNC\_E$ must be marked as *TRUE_SIGNALED*, and all sync edges $n \rightarrow n_2, n \in N$ with $ET = SOFT\_SYNC\_E$ must be marked as either *TRUE_SIGNALED* or *FALSE_SIGNALED* (see Section 2.1). Corresponding execution rules exist for all node types of a WF (incl. the start and the end nodes of loops).

The completion of a task leads to the signaling of its outgoing control as well as of its outgoing sync edges. The marking of edges follows well-defined *signaling rules*, which are based on the operational semantics of the different control structures. Upon successful completion of an AND-split node, for example, all outgoing edges are set to *TRUE_SIGNALED*. This, in turn, may trigger the activation of succeeding tasks, and so on. On the other hand, a task is skipped if it cannot be activated anymore. That is the case, for example, if the task belongs to a branch of a conditional branching that has not been chosen for execution, or if an ingoing sync edge of the task (with $ET = STRICT\_SYNC\_E$) has been marked as *FALSE_SIGNALED*. When a task node is marked as *SKIPPED*, its outgoing edges are set to *FALSE_SIGNALED*, which may lead to the skipping of succeeding nodes.

Finally, a WF instance terminates successfully when the ingoing control edge of its end node is set to *TRUE_SIGNALED*. We omit further details and present two examples instead.

Example 3. Figure 4 shows the use of the execution and signaling rules in connection with a loop. After E was completed and the loop condition $\underline{C}$ was evaluated to *TRUE*, the loop edge is set to *TRUE_SIGNALED* (see figure 4(a)). This, in turn, triggers the execution of the start node of the loop, whereupon the states of all nodes and edges of the loop's body (incl. the loop's end node and the loop edge) are reset and C is activated (see figure 4(b)).

Example 4. As a second example, consider figure 5(a). Assume that upon receiving a node termination event from B its outgoing control edge $B \rightarrow C$ signals *TRUE* and the edge $B \rightarrow D$ signals *FALSE*. This, in turn, leads to the reevaluation of the nodes C and D, which are activated respectively skipped. After skipping D, its outgoing control as well

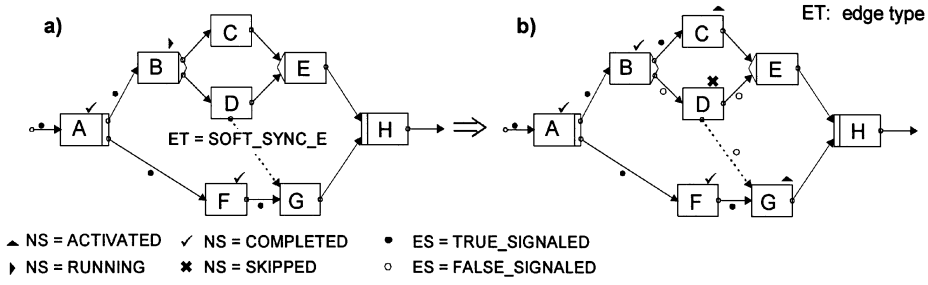

Figure 5. Synchronizing nodes from different branches of a parallel processing.

as its outgoing sync edges are set to *FALSE_SIGNALED*. Consequently, the state of *G* is reevaluated, and it is set to *ACTIVATED* (see figure 5(b)).

2.3. Correctness and consistency properties

As motivated in Section 1, formal criteria are needed to identify the possible exceptions resulting from a structural WF change and to provide support for handling them. In this section we give an overview of some of the correctness properties defined by ADEPT. We focus on the flow of data. Properties regarding the correctness of the control flow are only sketched at the beginning of this section.

Flow of control. A control flow graph (N, E) must meet certain *constraints* in order to ensure the correct execution of the WF at run-time. Each node $n \in N$ must be reachable from the WF's start node. That is, there is a valid sequence of signaling events leading from the initial marking of the WF graph to the activation of *n* (see Section 2.2). Furthermore, we require that from every reachable state of the WF a final state can be reached, i.e., there is a valid sequence of signaling events leading from the current marking of the WF graph to the activation of its end node. For non-cyclic WF graphs, which are based on task sequences and symmetrical branchings, these properties are satisfied *by construction*. This does not always apply to a WF graph whose control structures contain backward or sync edges. For example, the use of sync edges should not lead to cycles or termination problems of the flow. The presentation of conditions under which a graph (N, E) satisfies these properties and algorithms for their analysis are outside the scope of this paper.

Flow of data. In the following, we simplistically assume that for the correct execution of an action *A* (i.e., a task program or an auxiliary service assigned to 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. For each data link $df \in DF$ (cf., Definition 1) the domains of d^{df} and par^{df} must be type compatible. In addition, each parameter of an action must appear in exactly one data link $df \in DF$[2]. In order to avoid the invocation of actions with missing or incomplete input data the following constraint has to be added:

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

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

The predicate $Reads(n, d)$ ($Writes(n, d)$) expresses that an input parameter (an output parameter) of $n \in N \cup S$ is connected to d by a data link $df \in DF$.

This rule ensures that all input parameters of an action are supplied before it may be executed. Trivially, for a given task $n \in N, NT^n \neq STARTFLOW$ which reads a data element d, the rule DF-1 is satisfied if d is written by the start node of the WF, or if it is written by a preceding auxiliary service $s \in S_n^{\text{prec}}$. Furthermore, this rule guarantees that the output parameters of a WF (i.e., the input parameters of its end node) are completely supplied. In order to avoid unintended lost updates of data elements a second constraint has to be made, which we describe only informally here. For details the interested reader is refered to Appendix A.

Rule DF-2. Tasks from different branches of a parallel processing (with AND-join) are not allowed to have write access to the same data element, unless they are synchronized by a sync edge.

Write-after-write conflicts might also occur if two succeeding tasks have write access to the same data element and no read access occurs between them (see Appendix A). In (Hensinger, 1997) we present an algorithm for checking the correctness of a data flow schema with respect to the rules DF-1 and DF-2. The algorithm makes use of the symmetrical structuring of WF graphs, but it considers synchronizations between tasks from parallel branches as well. For a basic understanding, however, an example is more suitable.

Example 5. In the WF graph depicted in figure 3, G may read the data elements d_1 and d_2, but it is not allowed to read d_3; d_3 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 $V^H = \{\{STARTFLOW, A, F, G, B, C, E\}, \{STARTFLOW, A, F, G, B, D, E\}\}$ (cf., rule DF-1). G would not be allowed to write d_3 as this data element may be written by the concurrent task C (cf., rule DF-2).

Of course, the constraints upon which the definitions of the rules DF-1 and DF-2 are based must be relaxed in several respects. In our current implementation we follow a more flexible approach that distinguishes between optional and mandatory task parameters. Such extensions are important as not always all input parameters of a task are necessarily required for the correct processing of the task program. We further distinguish between parameters that can be supplied by a corresponding auxiliary service and those that cannot. We enrich interface descriptions with semantic information about parameters, and we provide support for referenced data (e.g., documents or database objects). Finally, concurrent write operations to the same data element must be allowed under certain conditions (e.g., in connection

with data elements of type *SET* or *LIST*). For simplification, we omit these extensions for the remainder of the paper.

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 new 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.

2.4. Adequacy of the ADEPT model

At first glance, the ADEPT model seems to be somewhat limited when compared to other WF models. These *structural limitations* are deliberated, as they offer advantages in several respects: The use of symmetrical control structures provides the basis for a syntax-driven design of WF's (cf., Kirsch, 1996) and for an efficient analysis of structural properties of a BP model (cf., Hensinger, 1997). We believe that this is crucial for the support of dynamic WF changes, especially if we want to ensure that applied changes are correct. In our experience, ADEPT offers a good compromise for the trade-off existing between the expressive power of a WF model on the one hand, and the complexity of model checking on the other hand. With respect to clinical BPs (by nature these processes are probably much more complex than the BPs found in many other application areas) it has proven that the modeling power of ADEPT is adequate. Note, that for the specification of more complex BPs, sync edges, failure edges, or null tasks (cf., Section 3) are very helpful. In addition, we are working on extensions of the ADEPT WF model (e.g., regarding concepts for the support of time and time dependencies) which will further increase its modeling power.

3. Dynamic structural changes of workflows

Based upon the ADEPT model we have developed a set of operations (ADEPT$_{flex}$) which serves as the framework for dynamic structural changes of WFs. The main emphasis in designing these operations was put on *correctness* and *consistency* issues: The application of a change operation to a specific WF instance must result in a WF with a syntactically correct schema and with a "legal" state, i.e., the change should not cause inconsistencies and run-time errors. Furthermore, the set of change operations should be *complete* and *minimal* in the sense of being able to realize each possible form of correct and consistent restructuring of a WF graph—with "minimal" we mean, that the number of change operations needed to achieve completeness should be kept as minimal as possible. Other design goals, which we do not discuss in detail in this paper, concern *efficiency* and *security* issues as well as *ease of use*.

In summary, ADEPT$_{flex}$ comprises operations for inserting tasks as well as whole task blocks into a WF graph, for deleting them, for fast forwarding the progress of a WF by skipping tasks, for jumping to currently inactive parts of a WF graph, for serializing

tasks that were previously allowed to run in parallel (and vice versa), and for the dynamic iteration and the dynamic rollback of a WF respectively of a WF 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 WF region by a new one. The *insert operation* shall serve as an illustrative example, and it will be discussed in more detail in Section 3.1. The other operations are sketched in Section 3.2.

3.1. Dynamic insertion of tasks

The addition of a new task to a WF during its execution may become necessary due to several reasons. The support of dynamically evolving WFs, unplanned events and missing or incomplete data name a few examples. The dynamic addition of a task to a WF is somewhat comparable to the addition of a new procedure to a program in the midst of its execution. When a task is inserted into a WF graph, new nodes and edges (including data links) must be added while maintaining the correctness and consistency of the WF. Current state-of-the-art systems do not provide a sufficient level of flexibility and consistency with respect to this operation. Typically, they allow the addition of an activity only upon completion of a task and before the activation of its successors (e.g., Hsu and Kleissner, 1996; Casati et al., 1996; Vogel and Erfle, 1992). Issues concerning data integrity are mostly ignored, leading to the problems mentioned in the introduction section. For the flexible support of BPs a more generic approach is required. Generally, it should be possible

- to add new tasks or even premodeled task blocks to a WF at any point of time during its execution
- to synchronize the execution of an inserted task with the execution of other tasks from the WF graph
- to insert tasks into WF regions which have not yet been entered
- to dynamically map the parameters of the added task to existing or to newly generated data elements

There is no problem to provide an operation for inserting a new task as a direct predecessor (or 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 would not yield to a satisfactory solution, as it does not reconcile with our design goals minimality and ease of use. Supporting the dynamic addition of tasks 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. Obviously, the addition of a task as a direct successor of another task is too weak to meet the requirements presented above. We therefore follow a more generic approach: a new task X, together with associated services S_X, data elements D_X, and data links DF_X, may be inserted into the graph of a WF instance by synchronizing its execution with two node sets M_{before} and M_{after}: The execution of X is triggered as soon as all tasks from the set M_{before} are either completed or cannot be worked on 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 branches of the WF graph. On the other hand, tasks from M_{after} may only be activated after completing X.

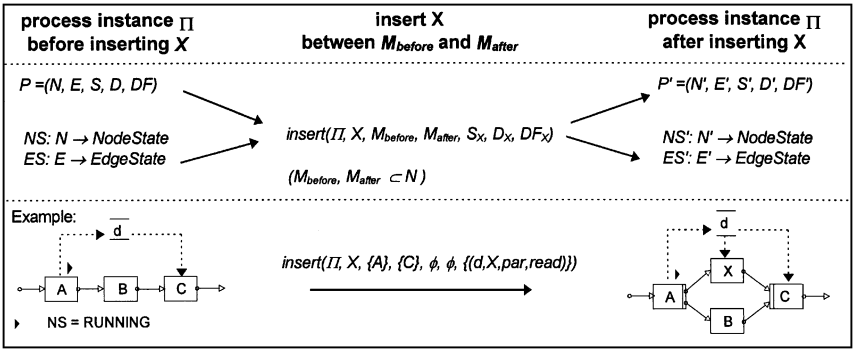

Figure 6. Dynamic insertion of a new task X (together with associated auxiliary services S_X, data elements D_X, and data links DF_X) between two task sets M_{before} and M_{after}.

The addition of a new task transforms the schema (N, E, S, D, DF) and the state (NS, ES) of the WF to a new schema (N', E', S', D', DF',) and a new state (NS', ES') (see figure 6). Such a graph transformation must result in a WF with a syntactically correct schema (incl. the flow of data) and with a legal state. In order to ensure 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 steps which become necessary when inserting a new task into a WF graph. First of all, we concentrate on the restructuring of the control flow. Afterwards we discuss relevant issues regarding the adjustment of the data flow.

Graph substitution.[3] In the following, let (N, E) be the syntactical correct control flow graph of a WF instance. The following steps must be carried out in order to insert a new task X between the two node sets M_{before} and M_{after}:

1. Find the minimal, closed subgraph $B \subseteq (N, E)$ that contains all nodes from $M_{\text{before}} \cup M_{\text{after}}$. Let n_{begin} denote the start node, and let n_{end} denote the end node of B.[4]
2. Insert an AND-split node n_1 as a direct predecessor of the node n_{begin}, and insert a corresponding AND-join node n_2 as a direct successor of the node n_{end}. Both, n_1 as well as n_2, are supposed to be null tasks[5] (NT = $NULL$), i.e., task nodes without associated actions. When embedding the node n_1 (n_2) into the WF graph, it takes over the input (output) firing behavior and the ingoing (outgoing) control edges of the node n_{begin} (n_{end}).
3. Insert a new node, representing X, as a branch between the nodes n_1 and n_2, and synchronize X with the tasks from M_{before} and M_{after}. That is, for each $B \in M_{\text{before}} \neg \{STARTFLOW\}$ add a sync edge $B \rightarrow X$, and for each $A \in M_{\text{after}} \neg \{ENDFLOW\}$ add a sync edge $X \rightarrow A$ (with $ET = SOFT\_SYNC\_E$).
4. Apply reduction rules and reevaluate the state of nodes and edges (see below).

As already mentioned, the application of these steps must lead to a syntactically correct WF graph. To ensure this, the following constraints must be made: Firstly, for all

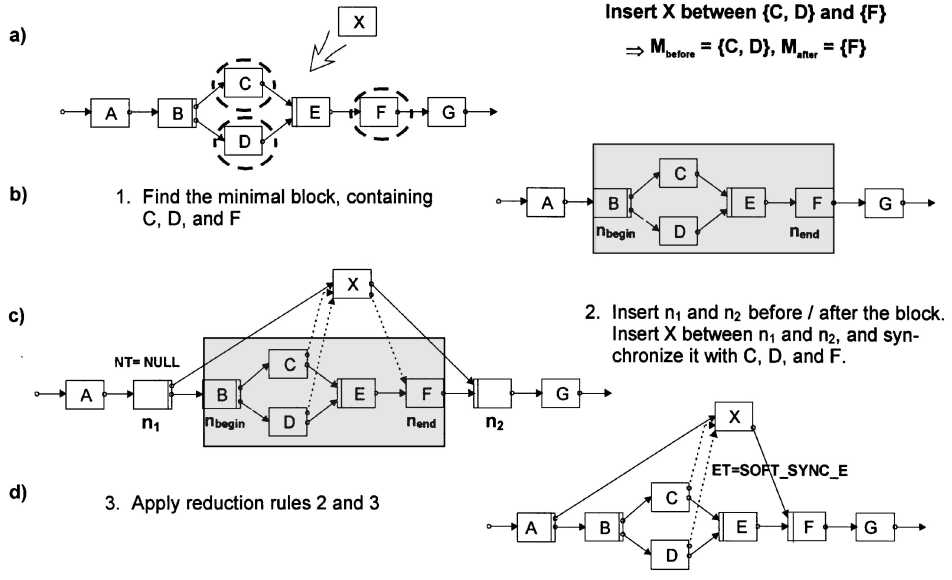

Figure 7. Insertion of a new task between two sets of nodes.

$n_a \in M_{\text{before}}$, $n_b \in M_{\text{after}}$ the node n_a must precede n_b in the flow of control. Secondly, the region covered by the nodes between M_{before} and M_{after} (incl. nodes from these sets) may only contain complete loop control structures. Finally, to avoid the insertion of unnecessary synchronization edges, nodes from M_{before} (M_{after}) should not succeed each other in the flow of control. One can show, that the insertion of a new task does not violate the syntactical correctness of the graph (N, E) and does not lead to termination problems if these conditions are satisfied. For further details the interested reader is refered to Appendix B. We omit them here and present an example instead.

Example 6. The example depicted in figure 7 shows how a task X is inserted between two sets of nodes. First of all, the minimal block that contains all nodes from the set $\{C, D, F\}$ is determined (see figure 7(b)). In the next step, a split node n_1, representing a null task, 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 it is synchronized with the nodes C, D, and F by adding the soft sync edges $C \rightarrow X$, $D \rightarrow X$, and $X \rightarrow F$ (see figure 7(c)). One can easily see that the symmetrical structuring of the WF graph is preserved and that the insertion of the sync edges does not influence the termination behavior of the WF (cf., Section 2.3).

The example further shows that null tasks and sync edges might be added to the WF graph which are not necessarily required to achieve the desired execution semantics. These nodes and 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. Reduction rules may be

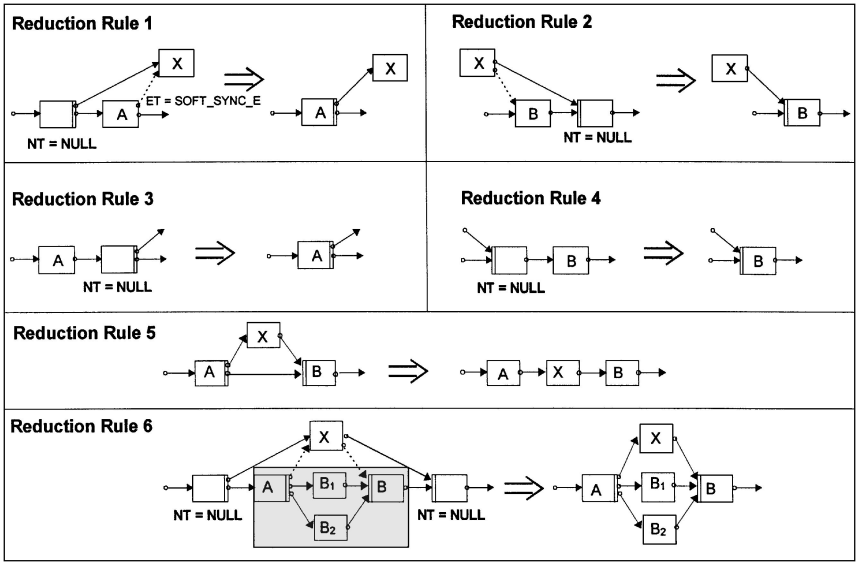

Figure 8. Examples of reduction rules.

applied to the null tasks originating from the insertion of a task and to their direct successors and predecessors. Their application does not change the WF's execution behavior, i.e., the set of valid task sequences remains unchanged. The effect of their application to the WF from figure 7(c) is shown in figure 7(d).

State constraints. The applicability of the insert operation depends on the state of the WF graph, too. In order to avoid the insertion of a new task as a predecessor of an already running or terminated task, we require that all elements from M_{after} must be in one of the states *NOT_ACTIVATED* or *ACTIVATED*. If a task $n \in M_{after}$ has already been activated, i.e., routed to worklists, the corresponding work items are removed from these worklists before the insertion takes place. The nodes from M_{before} may be in an arbitrary state.

After adding new nodes and edges to a WF graph its state must be reevaluated. This reevaluation is based on the execution and signaling rules presented in Section 2.2. Whether a newly inserted task is activated immediately or not depends on the current state of the WF graph. The former is the case if at insertion time all nodes from M_{before} are in a final state (i.e., *COMPLETED* or *SKIPPED*). Note, that the insertion of a new task does not necessarily mean that it will be activated for sure. If the task is inserted into a region of the WF graph that has not yet been entered, its execution may depend on future routing decisions.

Example 7. As a simple example, consider the graph shown in figure 9, and assume that a new task X shall be inserted between the AND-split D and its corresponding AND-join G. Using the presented graph substitution steps, applying reduction rule 6 (cf., figure 8),

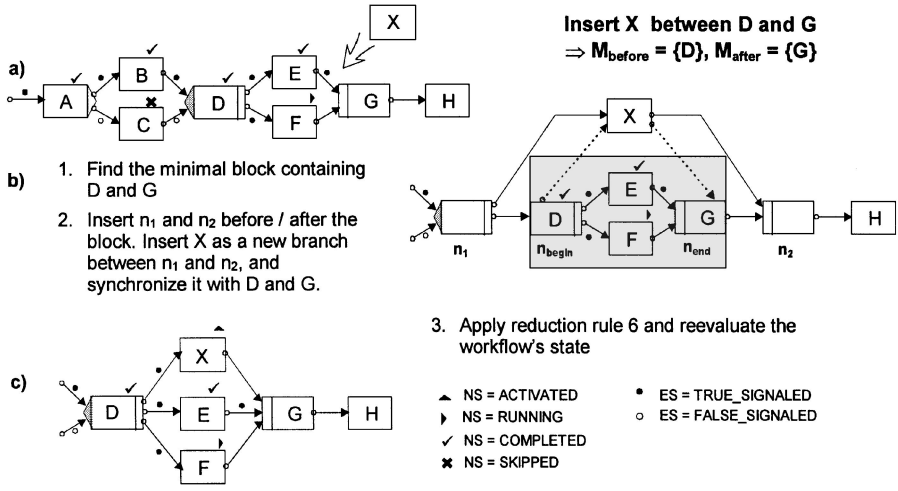

Figure 9. Adding a new task X between the AND-split D and its corresponding AND-join G.

and reevaluating the WF's state, the expected result is obtained (see figure 9(c)). Note, that it is possible to add X as a new branch between D and G, although the successors of the AND-split D, the nodes E and F, have already been completed respectively started. Furthermore, looking at the WF graph from figure 9(a), a new task X may not be inserted between the nodes D and E. In order to insert a new task between D and F, first of all, the execution of F would have to be aborted by the user.

Adjusting the data flow schema. As already mentioned, a new task X may be "plugged" into a WF graph, together with associated data elements D_X, auxiliary services S_X, and data links DF_X. So when a task X 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 of its state, but also it generally requires extensions of the sets D, S and DF. In any case, it must be ensured that the resulting WF schema (N', E', S', D', DF') meets the correctness properties defined in Section 2.3.

All input parameters of the newly inserted task X must be supplied before it may be executed (cf., rule DF-1). A simple approach to achieve this would be to request the necessary input data from the user initiating X. For this, X has to be connected with a preceding provider service s (see Section 2.3), whose output parameters logically correspond to X's input parameters. In our current prototype implementation such a service is supported by the dynamic generation and the dynamic processing of an electronic form, which makes use of the interface description of X. The procedure depicted in Table 1 shows how the sets D_X, S_X, and DF_X might be adapted in order to obtain a syntactically correct data flow schema satisfying rule DF-1.

Obviously, if the original WF schema (N, E, S, D, DF) satisfies the 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

Table 1. Adjusting a data flow schema by adding new data links.

$D_X := \emptyset$; $DF_X := \emptyset$;
create a provider service s with $OutPARS(s) := \emptyset$;
for all $par \in InPARS(X)$ **do**
  create data element dp with $(Id^{dp} \neq Id^d \forall\, d \in D \cup D_X) \wedge (dom^{dp} = dom^{par})$:
    $D_X := D_X \cup \{dp\}$
  create parameter p with $(Id^p = Id^{par} \wedge dom^p = dom^{par} \wedge dir^p =$ "OUT"):
    $OutPARS(s) := OutPARS(s) \cup \{p\}$
  $DF_X := DF_X \cup \{(dp, s, p, write), (dp, X, par, read)\}$
end
$S_X = S_X^{prec} := \{s\}$

satisfactory solution, since unnecessary and redundant data entries may result in the course of a WF execution, potentially leading to data inconsistencies. For a more intelligent support, it must also be possible to dynamically map parameters of the inserted task to already existing data elements from D. This raises a variety of challenging issues with respect to dynamic parameter mapping, which can only be sketched here. First of all, the data elements $C_X \subseteq D$ to which X's input parameters may potentially be mapped must be identified. According to rule DF-1 (cf., Section 2.3), we obtain

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

Example 8. As an example, consider the WF graph depicted in figure 3. 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 the definition of the set C_X is independent from the state of the WF. This ensures that all data elements of C_X are supplied when X is activated, independently from previously made routing decisions. On the other hand, there are scenarios in which it would be useful to relax this assumption and to consider the state of the WF as well; that is, to extend the set of data elements to which input parameters from X may be linked to

$$C_X^* = C_X \cup \{d \in D \mid \exists n^* \in \overline{pred}\ (X) :\ NS^{n^*} = COMPLETED \wedge Writes(n^*, d)\}^6$$

Example 9. As an example, take the insertion shown in figure 9, and assume that B is the only task that writes the 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 potentially mapped to d_1, although this data element is not contained in the set C_X (see figure 10). Following this approach, it might become necessary to undo the insertion of X in case of a backward operation. As we will see in Section 4, in this context it makes a big difference whether X should be executed at most once (*temporary insertion*), or whether the insertion should be valid until completion of the WF (*permanent insertion*).

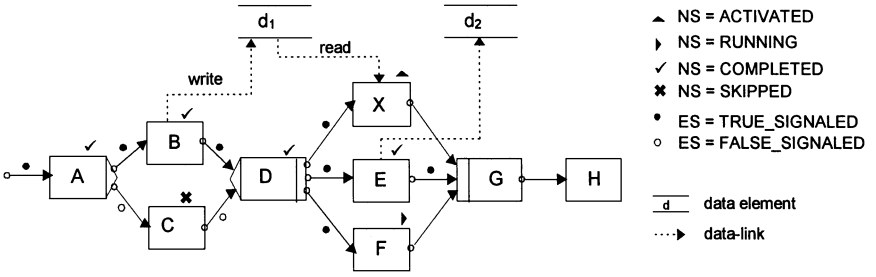

Figure 10. Linking an inserted task to existing data elements.

The set C_X (or C_X^*) only describes which data elements may be considered when input parameters of X are mapped to elements from D. A specific input parameter $p \in InPARS(X)$ may be linked to a data element $d \in C_X$ (or $d \in C_X^*$), only 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 presented in (Blaser, 1996). Basic to it is a controlled vocabulary which is used for the naming of data elements and task parameters (respectively the data structures they are built upon). The vocabulary is organized as a semantic network and considers semantic relationships between the concepts, upon which data elements and parameters are built. In (Blaser, 1996) 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 or newly inserted data elements. In order to avoid unintended lost updates, an output parameter may be linked to a data element, only if the rule DF-2 is further satisfied. In the WF 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. So far we have concentrated on correctness and consistency issues regarding the dynamic addition of a task to the graph of a WF instance. 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 important to allow process participants to fix a date or a deadline for the execution of the inserted task. The necessary extensions for this are described in (Grimm, 1997).

Secondly, for security reasons, $\text{ADEPT}_{\text{flex}}$ allows WF designers (as well as selected process participants) to restrict the use of the insert operation to specific WF types or WF categories, to selected users or user roles, to specific regions of a WF graph (e.g., a task block), to selected WF states, to specific activity types or categories, or to any combination of them. Generally, also we do not require that the user who adds a task to a WF must subsequently work on it. This provides additional flexibility to process participants, as they are allowed to add tasks to a WF of which the execution may be explicitly or implicitly delegated to other process participants. This requires a powerful meta model for capturing organizational entities and relationships between them.

Table 2. Examples for the use of the insert operation. $succ(A)$ ($pred(B)$) denotes the set of direct successors (predecessors) of the task A.

Insertion	Choice
An intermediate step between a node A and its successors (A may be a split node of arbitrary type)	$M_{before} = \{A\},\ M_{after} = succ(A)$
An intermediate step preceding the execution of a task A	$M_{before} = pred(A),\ M_{after} = \{A\}$
A new branch of a parallel branching with split node Sp and join node J	$M_{before} = \{Sp\},\ M_{after} = \{J\}$
A new task without any additional synchronization	$M_{before} = \{\text{STARTFLOW}\}$, $M_{after} = \{\text{ENDFLOW}\}$

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

Application. The insert operation described covers a broad spectrum of applications, and it allows a variety of user-friendly operations. Some of them are summarized in Table 2.

The insert operation also serves as the basis for composing higher-level operations. For example, several instantiations of the same task type (*dynamic task*) can be realized by the repetitive use of this change operation. Its generality 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 WF graph, and it may be dynamically extended by the repetitive application of the insert operation presented. As a last interesting aspect, we use the insert operation for internal exception handling as well. 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 (see Section 3.2)

These examples demonstrate that our approach is able to support a large variety of different application scenarios. In the next section we sketch other change operations and some interesting issues related to 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, to jump forward to WF 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 WF graph (incl. the undoing of temporary changes). Due to space limitations we must omit a presentation of the whole set of operations here. In the following, we therefore only deal with some interesting issues related to the deletion of tasks and to the dynamic modification of premodeled task sequences.

Dynamic deletion of tasks. Individual tasks or task sequences may have to be skipped or removed when the conditions for their execution become unnecessary. Of course, the

deletion of tasks should not always be allowed. Firstly, nodes which are an integral part of the WF structure (e.g., the start node of the WF) must not be deleted at all. Secondly, WF designers may customize a WF schema in order to disallow the deletion of individual tasks or tasks from specific WF regions.

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

Concerning the adjustment of the control flow graph, the delete operation is realized by substituting a null task (see Section 3.1) for the task to be deleted. This approach can be handled in a simple and effective manner, as the node of the deleted task and its associated (control) 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, its associated auxiliary services and data links must be removed from the set S and from the set DF. This might lead to missing or incomplete input data of succeeding data-dependent steps and therefore to a violation of the rule DF-1 (cf., Section 2.3).

Let $N^* \subset \overline{succ}\ (X)$[7] denote the set of tasks whose input parameters are not completely supplied due to the deletion of X. The following exception handling policies can be applied in $\text{ADEPT}_{\text{flex}}$ to deal with such cases and to regain a correct and consistent WF graph:

- 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 it 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 $S_n^{\text{prec}},\ n \in N^*$ (see Section 3.1)—this must not lead to the violation of the rule DF-2!
- Abortion of the delete operation.

Of course, these policies may be used in combination with each other. In order to relieve users from performing the necessary adjustments of the data flow schema "manually", ADEPT supports the specification of *success dependencies* between succeeding tasks. If a task X is deleted from the WF graph at run-time, all succeeding tasks which are success-dependent on X are deleted as well. This, in turn, may lead to the cascading deletion of other tasks. 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 (Davis Jr., 1978; Leymann, 1995), but it is applied here to the structure of the WF.

Changing task sequences at run-time. As mentioned in the introduction section, changes of premodeled task sequences frequently become necessary in exceptional situations. Since WF designers are generally not capable to predict all possible deviations in advance,

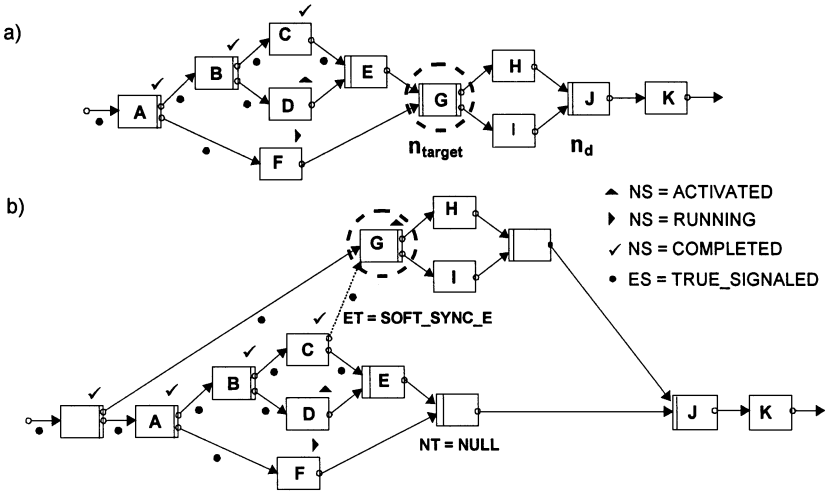

Figure 11. Parallelization of tasks, that were previously constrained to be executed serially, due to a *jump forward* operation.

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.

Example 10. As an example, take the WF graph depicted in figure 11(a), 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 marked. Assume further, that the steps D, E, and F have to be finished or worked on concurrently, but they must be completed before task J may be triggered. In order to achieve this, the WF graph must be restructured as shown in figure 11(b). 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 or 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:

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

Tasks from this set may be aborted, omitted, or as in our example be further worked on. For the latter case, their execution must 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 activated.

Finally, changes of premodeled task sequences may lead to an incorrect data flow schema if no further precautions are taken. The rules presented in Section 2.3 contribute to identify

such cases and to provide adequate mechanisms for exception handling. Due to lack of space this aspect cannot be discussed here. We conclude that more user-friendly operations, providing elegant and efficient support for dynamic structural changes, can be based on the presented operations.

4. Change management

Several instances of a specific WF type may be active at the same time. As changes of different kinds may have been applied to them, several aspects must be considered:

- WF instances of the same type, i.e., the same starting schema, must be (logically) represented by different WF graphs.
- Changes, that are applied to an individual WF instance, may depend on previous changes of that WF.
- Structural WF changes may require concomitant modifications in order to preserve the correctness and the consistency of the WF graph (see Section 3); the necessary graph adaptations must be carried out within the same transaction in order to allow forward recovery in the presence of failures.
- It must be possible to undo structural changes of a WF under certain conditions.

For the management of structural WF changes, it makes a big difference whether an applied change must be preserved until the completion of the WF (*permanent change*), or whether it is only of temporary nature (*temporary change*). This division is particularly important for the support of long-running BPs, where changes may affect WF regions that are entered several times, e.g., due to loop iterations or due to the partial rollback of a WF. If a task is inserted into the body of a loop, for instance, it must be specified whether this insertion should only be valid for the current iteration of the loop or for following iterations as well. In the first case, the added task is executed at most once, and the (structural) change must be undone before the next iteration of the loop is entered (i.e., the inserted task must be removed together with its associated data links and services). For the remainder of this section, we simplistically assume that the durability of a change—temporary vs. permanent—can be specified at the time it is applied to the WF instance.

4.1. Change history

Ideally, the undoing of temporary changes and the necessary adjustments of the WF graph should be completely handled at the system level without costly user interactions. In order to achieve this, the run-time system must have precise information about previously made changes. In our approach, we maintain the following information for each WF instance Π:

- a WF graph P_{all} reflecting the current *structure* and the current *state* of Π. This graph considers all changes that have been applied to the WF instance, temporary as well as permanent ones.

- a WF graph P_{perm} which has resulted from applying permanent changes to the starting schema of Π. Temporary changes as well as the state of Π are not considered by this graph.
- a *change history* C, which is used analogously to the WF's execution history: it records data on all changes, that have been applied to Π, in a chronologically ordered list. These data may later be used to undo changes. Each history entry contains the following information: (1) the *type* of the change operation, e.g., insertion or deletion of a task (incl. its call parameters); (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 is used by the run-time 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 (see Section 3); (6) the list of *change primitives* (incl. their call parameters) that 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, or data links.[8]

The execution of a WF is based on the graph P_{all}. Logically, this graph must be kept for each WF instance, as different kinds of ad hoc changes may be applied to WF instances of the same type (i.e., with the same starting schema). We require the additional graph P_{perm} in order to ensure that permanent changes remain correct when temporary modifications are undone. For example, a permanently inserted task must not be data-dependent on a temporarily inserted one. Otherwise, the undoing of the temporary insertion leads to an incorrect data flow schema, which may cause severe run-time errors. In order to avoid such dependencies, the application of a permanent change to a WF instance requires additional checks, which can be based upon the graph P_{perm} (see below).

4.2. Applying temporary and permanent changes

The introduction of temporary and permanent changes to a WF instance Π requires different procedures. In order to perform a *temporary change* c_t, we must check whether it can be applied to P_{all} while maintaining the correctness and consistency of this WF graph (cf., Section 3). If unresolvable exceptions occur, the change operation is aborted. Otherwise, c_t is applied to P_{all}, and a corresponding entry is added to the change history C. Note, that a temporary change may be based on previously made temporary changes as well as on permanent changes. In addition, it may consider the state of the WF (cf., 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 P_{perm} does not violate the correctness of this graph. In contrast to temporary changes, this verification is performed independently from the state of Π as well as from temporarily applied changes. Otherwise c_p may be based on wrong assumptions, which might cause severe problems when the state of Π is reset or when a temporary change is undone due to a backward operation. As an example, take the insertion of the task X in figure 10 (cf., Example 9); as this change makes use of a previously made routing decision, it can only be applied temporarily. Generally, if the change c_p is correct

with respect to P_{perm}, we must also check its applicability to the graph P_{all}. 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 the change history C.

4.3. Undoing temporary changes

Up to now we have described how changes are managed and how they are put into effect. In the following, we sketch the necessary steps for undoing temporary changes (i.e., for removing them from the WF 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 (cf., Section 3) and to their undoing.

Important points for the decision which changes must be undone and which not are the *durability* of the change (temporary vs. permanent) and their *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} (cf., Section 3.1), whereas the start region of a delete operation consists of the null task replacing the removed task (cf., Section 3.2). For simplification, we require that a temporary change must be undone if each node of its start region is in a final state ($NS \in \{COMPLETED, SKIPPED, FAILED\}$), and if it 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 (see 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* (cf., 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; other temporary changes may exist which have been based on these modifications and which 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 quite usual and may be explicitly desired by users. They therefore must be taken into account 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 entry c_1 in C that must be undone due to the backward operation; that is, the oldest entry of the change history whose start region is covered by the backward region. If no such entry exists in C, 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 WF 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}.[9] In case of a temporary change, first of all, we check whether its start region is covered by the backward region. If that

is the case, the change is not redone, and e is removed from C. 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), the redo will not be performed, and the initiator of the change will be informed.

Example 11. Taking the graph depicted in figure 12(a), we illustrate the principle feasibility of this approach. Figure 12(b) shows the same WF graph after the flow has proceeded, the two nodes N^* and N^{**} were temporarily inserted into the graph (together with the data element d and corresponding data links), and the node F was permanently deleted. Figure 12(d) shows the resulting graph after applying a backward operation and after undoing the temporary changes. Although its start region $\{H\}$ is not contained within the backward region, the change $c_t^{(2)}$ is undone, too, since it is dependent on $c_t^{(1)}$.

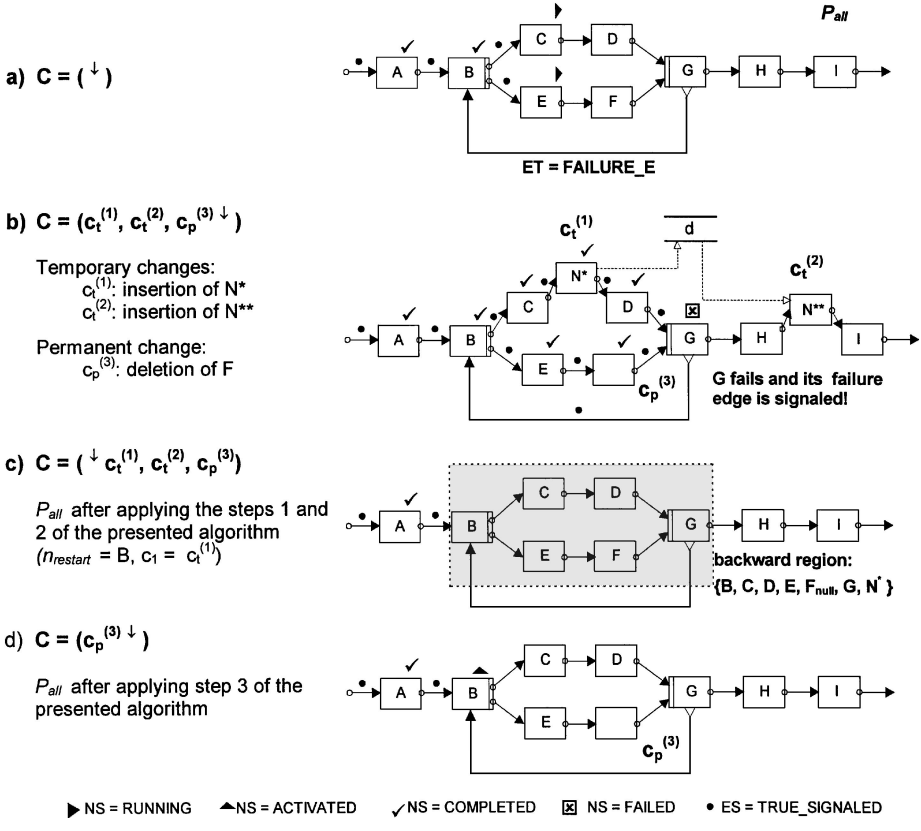

Figure 12. Undoing temporary changes after a failure edge has been signaled. (The entries of the change history C that precede the arc ↓ correspond to the changes currently applied to the graph P_{all}.)

The assumptions made in this section may be relaxed. For example, in some cases it is desirable to preserve a temporary change when a rollback operation is applied, but to undo it when a new iteration of a loop is entered. As a last interesting aspect, the use of a change history contributes to increase the user friendliness of the system, since structural changes can be undone (*UNDO of structural changes*) by the user (e.g., the initiator of the change), as long as they have not yet influenced the execution of the WF, and as long as no further changes have been based upon them.

In summary, the support of dynamic structural changes make great demands on the WF engine and on change management. In this paper we have concentrated on conceptual issues related to dynamic WF changes. There are a variety of important implementation issues, that we have not addressed, but which are of high importance for the management of dynamic changes at the workflow enactment level: the internal representation and the persistent storage of WF instances (i.e., the underlying data structures), the transactional execution of (possibly long-running) change operations, WF recovery, the synchronization of concurrent change transactions, the resolution of conflicting implementation goals (e.g., performance vs. flexibility), or dynamic changes of WFs which are controlled by different distributed WF servers name a few examples.

5. Related work

It is widely recognized that state-of-the-art WF technology does only provide rudimentary support for exception handling and for dynamic structural changes of running WF instances (see, Barthelmess and Wainer, 1995; Ellis et al., 1995; Reichert et al., 1997a; Sheth and Kochut, 1997; Siebert, 1996). Both, in research and in commercial WFMSs, several directions can be made out that try to overcome these limitations. These approaches focus on

- the provision of services for exception handling and for ad hoc structural changes
- the support of WF designers in modifying the schema of a WF and in propagating the applied changes to already running WF instances that started with the old schema (WF schema evolution)
- the integration of WFMSs with groupware technology to combine formal and well-structured processes with informal group processes.

5.1. Exception handling and ad hoc structural changes in WFMSs

We are mainly interested in process-oriented WF technology as opposed to e.g., Lotus Notes or Groupflow (Nastansky and Ott, 1996). Current process-oriented WFMSs like FlowMark (Leymann and Altenhuber, 1994) or DOMINO (Kreifelts et al., 1991), however, do only address a small part of the issues discussed in this paper. Although most of them allow online modification of task/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 address these issues. The proposals

made by HOON (Han et al., 1996), ProMInanD (Vogel and Erfle, 1992), ObjectFlow (Hsu and Kleissner, 1996), WIDE (Casati et al., 1997), MOBILE (Heinl et al., 1996), and DYNAMITE (Heimann et al., 1996) are worth mentioning.

Han et al. (1996) suggest a Petri net based model (HOON) for adaptive WFs. The basic idea is to use mechanisms for later binding of software components and WF models, which may be dynamically and hierarchically combined at run-time. HOON does not support dynamic changes in the narrower sense. Structural changes of a WF model are not possible after it has been bound to a net's transition. The authors do also not talk about correctness issues.

ProMInanD is a representative of WFMSs based on the object migration model (see, Karbe et al., 1990). A WF, together with its definition, is regarded as an object ("electronic circulation folder") which is sent from user to user according to the modeled control flow. Only the user who is currently in charge of the folder may change the flow, e.g., by adding an intermediate task. A potential weakness is the simplicity of the used WF model—parallel and iterative executions are not explicitly supported—and the lack of a clear theoretical basis. The offered change operations consider only the control flow, but they ignore other structural components of the WF specification. The data flow is limited to the exchange of files between tasks, so that the WFMS has minimal control over it. This leaves significant complexity to application programmers, who themselves must ensure the correctness the data flow when the WF is restructured.

A comparable functionality is offered by ObjectFlow (Hsu and Kleissner, 1996) which uses a constrained Petri net based model. Users may temporarily change the course of the flow or add intermediate tasks. In addition, ObjectFlow supports dynamic tasks, i.e., the multiple concurrent instantiation of the same task type at a specific point of the WF. A limited mechanism for exception handling is offered: the actions which are necessary to handle abnormal events have to be explicitly modeled as additional paths in the WF 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. This approach may lead to complex WF models as the offered modeling constructs are not high-level (Ellis and Nutt, 1993).

The WIDE WF model offers a trigger-based approach for exception handling (Casati et al., 1997). Exception handlers (EHs) can be installed to handle events such as the cancellation of a task or the break of the normal flow. In contrast to ObjectFlow and ProMInanD, the WF may proceed while the exception is handled. For each type of exception WIDE provides a default EH (e.g., for user notification), which may be overwritten by the WF programmer. However, it lies in the responsibility of programmers to avoid inconsistencies and errors, which complicates application development and may introduce new errors and exceptions into the model.

The MOBILE WF model (Heinl et al., 1996) allows the use of incomplete sub-process models at predefined points (i.e., nodes) of the WF. Incomplete models are described in terms of goals as well as partially defined process patterns, and they must be completed at run-time. The authors do not indicate how users are supported in changing an incomplete model and which operations are available. Correctness issues are also not addressed.

A more competitive approach is offered by DYNAMITE (Heimann et al., 1996). DYNAMITE aims at the support of the software development process, which is often highly

dynamic and for which the planning and the execution of tasks may overlap. DYNAMITE uses dynamic task nets, which are built and modified incrementally during process execution. Formally, task nets are based on a graph rewriting system. The tasks which shall be dynamically added to a task net must be predefined in a process schema. This significantly limits the dynamics of this approach. Operations for changing task sequences and for deleting tasks are not available, and correctness issues are not discussed by the authors.

The same holds for *transactional WFs*, whose emphasis and strength lie in different areas such as reliability or forward recovery in the presence of failures (Alonso et al., 1996; Attie et al., 1993; Kamath and Ramamritham, 1995; Hsu, 1993, 1995; Worah and Sheth, 1997). Transactional WFs apply concepts of advanced transaction models (Elmargarmid, 1992); they are, therefore, pretty good in handling task failures or abnormally terminated WFs (e.g., Eder and Liebhart, 1995). Concepts like "spheres of compensation" (Leymann, 1995; Davis Jr., 1978) will further contribute to simplify and to speed up application development and to make WF applications more reliable. However, transactional WFs do only meet a small part of the issues discussed in this paper. As the WF engine will generally not have the knowledge to detect and to handle all possible failures and exceptions alone, dynamic changes typically require user involvement. Besides this, transactional WFs must address issues concerning the transactional execution of structural changes and their synchronization.

5.2. WF schema evolution

There are few approaches which address correctness issues in connection with dynamic structural changes. Notable exceptions come from (Ellis et al., 1995; Casati et al., 1996). In contrast to $\text{ADEPT}_{\text{flex}}$, which concentrates on ad hoc changes of individual WF instances, these approaches deal with *changes of the WF schema* and their propagation to running WF instances whose execution started with the old schema. Although the 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 end users. Like $\text{ADEPT}_{\text{flex}}$, both approaches are based on a conceptual WF model. However, they restrict their considerations to dynamic changes of the control flow; other relevant aspects are left aside.

Ellis et al. (1995) propose a mathematical model, which is based on constrained Petri nets. A change corresponds to the replacement of a subnet of the WF graph by a new subnet; it is said to be correct if afterwards the corresponding WF instances can either be executed according to the old schema or to the new one. The emphasis and strength of this approach lie in its formal foundation. Casati et al. (1996) address the problem of WF schema evolution from a static as well as from a dynamic point of view. In contrast to Ellis et al. (1995) 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 WF instances can be transparently migrated to the new version.

How to integrate dynamic structural changes at the schema level with ad hoc changes at the instance level is an outstanding research issue. When looking at the proposal made by Ellis et al. (1995), for example, it is implicitly assumed that the execution of all instances of a specific WF type is based on the same net. This assumption cannot be maintained when ad hoc structural changes at the instance level must be considered, too. The proposals made in Section 4 can be considered as a first step towards a solution of this problem.

5.3. 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 (Winograd and Flores, 1986) which reduces organizational processes to networks of commitment loops between process participants. Other approaches follow goal-based models (e.g., Blumenthal and Nutt, 1995) or use circulation folders (Karbe et al., 1990). All these approaches share the disadvantage 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 (GW) systems by supporting unstructured activities at specific points of a WF (e.g., Antunes et al., 1995; Blumenthal and Nutt, 1995; Sheth and Kochut, 1997; Weber et al., 1997). A group task corresponds to a node in the WF graph. 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 issues include the integration of WFMSs with GW technology, the exchange of data between them, and the management of contextual information (Blumenthal and Nutt, 1995; Weber et al., 1997). Several authors doubt the suitability of this approach (Heinl et al., 1996; Siebert, 1996). As a potential disadvantage they consider the “break” between structured and unstructured parts of work resulting from the combined use of WF with GW technology. Important features such as auditing, rollback, security or consistency may be lost when unstructured group tasks are not controlled by the WFMS.

6. Summary and outlook

In this paper we have concentrated on issues regarding dynamic structural changes of WF instances during their execution. We have argued that such changes are rather the norm in computerized processes and that their adequate 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 WF model. We demonstrated its suitability for the (precise) specification of WFs, the verification, and testing of the correctness of WF specifications, and the execution 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 algorithms needed 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 $\text{ADEPT}_{\text{flex}}$ model which is based upon ADEPT has been presented and its *adequacy* with respect to dynamic structural changes has been demonstrated. $\text{ADEPT}_{\text{flex}}$ comprises a *complete* and *minimal* set of change operations which ensure the *correctness* and *consistency* of the resulting WF graph by *construction*. Taking the dynamic addition of tasks as an example, we have demonstrated that the 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 have 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. We have compared our model with other WF models, and we have shown that the semantics offered by the change facilities of $\text{ADEPT}_{\text{flex}}$ captures those of other models by far. Finally, we have addressed issues regarding the management of temporary as well as permanent changes and the undoing of temporary changes when backward operations are applied.

The work presented in this paper has been well-motivated by a variety of organizational studies and analyses of processes from the clinical domain (Kuhn et al., 1994; Meyer, 1996; Reichert et al., 1996) where ad hoc changes and dynamically evolving WFs are rather usual and exceptions do frequently occur. We also implemented complex processes from the University's Women Hospital by applying current WF technology (Reichert et al., 1997b). As a result, today's WFMSs offer perspectives, but they are far away from providing the flexibility needed by clinical users. The role of application developers and end users in handling exceptions and in changing the structure of WFs is not well-understood and therefore poorly integrated with today's WFMSs.

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 premodeled process templates from a 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 (and on which we are currently working on) are

- the support of simultaneous changes on individual WF instances
- the application of dynamic changes to WFs whose schema is decomposed into several parts that may be kept and controlled by different WF servers (e.g., Bauer and Dadam, 1997; Wodtke and Weikum, 1997)
- 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 (Heinlein and Dadam, 1997)

- the development of general concepts for the integration of dynamic structural changes at the schema level (e.g., Casati et al., 1996; Ellis et al., 1995) with changes at the instance level (as proposed in this paper)
- the provision of "intelligent" interfaces for application programmers and for end users; adding only functionality to current WF technology without understanding how the programmer or the end user will be able to utilize it will certainly not be helpful. In any case, dynamic changes should be possible at the minimum cost to application programmers as well as end users.

We believe that dynamic WFs are a field that would benefit by more intense study by the research community. During the last years, we have developed a series of small prototypes each of which concentrating on a single aspect like the modeling component, support of temporal constraints, and support of dynamic changes in order to better understand end-user related issues as well as implementation aspects (Blaser, 1996; Grimm, 1997; Hensinger, 1997; Kirsch, 1996). Recently we have started the design and implementation of ADEPT$_{workflow}$, the ADEPT workflow engine, which will integrate the features described above within one system.

Appendix A

Correctness of a data flow schema

Definition A.1 (*Successor funtion*). We define

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

with

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

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

with

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

$succ(n)$ comprises the set of all direct successors of the node $n \in N$, i.e., the set of nodes which are the destination of a control or of a sync edge with source n. $\overline{succ}$ denotes the transitive closure of this function. $\overline{succ}(n)$ comprises those tasks of the WF 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 those nodes from N which are reachable from n by following only control edges. As the meaning of the corresponding predecessor functions and their transitive closures is intuitive, we omit their definition here.

Example A.1. In the WF graph shown in figure 3 we have

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

Rule DF-2. Let $P = (N, E, S, D, DF)$ be the schema of a WF. For $n_1,\ n_2 \in N$ with $Writes(n_1, d) \wedge Writes(n_2, d)$ we require

(1) $(n_1 \in \overline{succ}(n_2) \vee n_2 \in \overline{succ}(n_1))$ **or**
$(\exists n_s \in N: n_s \textit{ is OR-join} \wedge n_s \in M := (\overline{succ_c}(n_1) \cap \overline{succ_c}(n_2)) \wedge \forall n \in M, n \neq n_s : n \in \overline{succ_c}(n_s))$
(2) $n_2 \in \overline{succ}(n_1) \Rightarrow \exists n_3 \in (\overline{succ}(n_1) \cap \overline{pred}(n_2)) \cup \{n_2\}$ with $Reads(n_3, d)$

Simplistically, we have omitted write operations of elements from S in the presentation of this rule. If n_1 and n_2, which have write access to the same data element, do not succeed each other in the flow of control, they must belong to different branches of a branching with an (inner) OR-join n_s (cf., Rule(1)). Therefore, tasks from different branches of a parallel processing (with AND-join) may not have write access to the same data element, unless they are serialized by the use of a sync edge. Rule (2) aims at avoiding write-after-write conflicts of succeeding tasks.

Appendix B

Correctness of a graph substitution when adding new tasks

Theorem B.1 (Syntactical correctness and termination behavior after adding a new task). *Let (N, E) be the syntactically correct control flow graph of a WF schema P, for which* (1) *every node $n \in N$ is reachable from P's start node and for which* (2) *from every reachable state a final state can be reached. Furthermore let $M_{before},\ M_{after} \subset N$ be two disjoint sets with*

(I_1) $\forall n_b \in M_{after},\ \forall n_a \in M_{before} : n_b \in \overline{succ}(n_a)$ *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*

(I_2) *The region covered by the nodes from the set $M_{before} \cup M_{after} \cup (\overline{succ}(M_{before}) \cap \overline{pred}(M_{after}))$ may only contain complete loop control structures*

Then, the application of the presented insert algorithm to add a new task X between the sets M_{before} and M_{after} (cf., Section 3.1*) results in a syntactically correct control flow graph (N', E') again, which also satisfies the properties* (1) *and* (2).

Proof sketch: We sketch the idea for the proof of this theorem 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 null tasks does not influence the termination behavior of the WF. The restrictions for the use of sync edges (see 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 synchronize a node contained within a loop body with the inserted task X (because of condition (I_2)) and do not lead to cycles or termination problems. The latter is guaranteed by the ordering of tasks from the sets M_{before} and M_{after}

(because of condition (I_1)). 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') also satisfies these properties. Note that only sync edges of the type $ET = SOFT\_SYNC\_E$ are used. □

Acknowledgments

We are grateful to Clemens Hensinger, Thomas Bauer, Christian Heinlein, and Birgit Schultheiß for numerous interesting discussions on topics related to our research.

Notes

1. ADEPT stands for *A*pplication *D*evelopment Based on *E*ncapsulated Premodeled *P*rocess *T*emplates.
2. This does not necessarily mean that the value of an input parameter cannot be aggregated from the values of several data elements (cf., Blaser, 1996).
3. This algorithm must be extended if a user wants to insert a new task between the start (or end) node of a loop and an arbitrary node contained within the loop's body.
4. B is defined as follows: It contains all nodes from $M_{before} \cup M_{after}$ (excl. the start node and the end node of the WF graph), and it has a unique start/end node. Furthermore, if any node—except the end node—from B corresponds to the start node of a loop, the loop's end node must also be contained within B, (and vice versa). The same constraints apply to branchings.
5. We have adopted this notion from (Casati et al., 1996). A null task does not correspond to any action in the real world. After a null task has been triggered, its outgoing edges are marked immediately.
6. The set $\overline{pred}(X)$ corresponds to the transitive closure of nodes preceding X in the flow of control (cf., Appendix A). Simplistically, we have omitted write operations from elements of S in the definition of this set.
7. $\overline{succ}(X) \subseteq N$ comprises those nodes that are reachable from X by following control as well as sync edges (cf., Appendix A).
8. Note, that these primitives modify the sets N, E, S, D and DF. Generally, their individual application to a WF graph does not preserve its syntactical correctness and consistency.
9. 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.

References

Alonso, G., Agrawal, D., Abbadi, A.El., Kamath, M., Günthör, R., and Mohan, C. (1996). Advanced transaction models in workflow contexts. *Proc. 12th Int. Conf. on Data Engineering*. New Orleans, Louisiana: IEEE Computer Society Press.

Antunes, P., Guimaraes, N., Segovia, J., and Cardenosa, J. (1995). Beyond formal processes: Augmenting workflow with group interaction techniques. *Proc. Conf. on Organizational Computing Systems*. Milpitas, CA: ACM Press.

Attie, P.C., Singh, M.P., Sheth, A., and Rusinkiewicz, M. (1993). Specifying and enforcing intertask dependencies. *Proc. 19th Int. Conf. on Very Large Databases*. (pp. 134–145). Dublin, Ireland: Morgan Kaufmann Publishers.

Barthelmess, P. and Wainer, J. (1995). Workflow systems: A few definitions and a few suggestions. *Proc. Conf. on Organizational Computing Systems*. (pp. 138–147). Milpitas, CA: ACM Press.

Bauer, Th. and Dadam, P. (1997). A distributed execution environment for large-scale workflow management systems with subnets and server migration. *Proc. 2nd IFCIS Conf. on Cooperative Inf. Sys.* (pp. 99–108). Kiawah Island, South Carolina, USA: IEEE Computer Society Press.

Blaser, R. (1996). Composing Processes by the Reuse of Application Components (in German). Masters Thesis, University of Ulm, Germany.

Blumenthal, R. and Nutt, G.J. (1995). Supporting unstructured workflow activities in the bramble ICN system. *Proc. Conf. on Organizational Computing Systems*. (pp. 130–137). Milpitas, CA: ACM Press.

Casati, F., Ceri, S., Pernici, B., and Pozzi, G. (1996). Workflow evolution. *Proc. 15th Int. Conf. on Conceptual Modeling*. (pp. 438–455). Cottbus, Germany: Springer.

Casati, F., Grefen, P., Pernici, B., Pozzi, G., and Sánchez, G. (1997). WIDE Workflow Model and Architecture. Technical Report, University of Milano, Italy.

Davis Jr., C.T. (1978). Data Processing Spheres of Control, *IBM Systems Journal*, 17, 179–198.

Eder, J. and Liebhart, W. (1995). The workflow activity model WAMO. *Proc. 3rd Int. Conf. on Cooperative Inf. Sys.* (pp. 87–98). Vienna, Austria.

Ellis, C.A. and Nutt, G.J. (1993). Modeling and enactment of workflow systems. *Proc. 14th Int. Conf. on Application and Theory of Petri Nets*. (pp. 1–16). Chicago, WA: Springer.

Ellis, C.A., Keddara, K., and Rozenberg, G. (1995). Dynamic change within workflow systems. *Proc. Conf. on Organizational Computing Systems*. (pp. 10–21). Milpitas, CA: ACM Press.

Elmargarmid, A.K. (Ed.) (1992). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers.

Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An Overview of Workflow Management, *Distributed and Parallel Databases*, 3, 119–153.

Grimm, M. (1997). $ADEPT_{time}$—Dealing With Temporal Dependencies in Flexible WFMSs (in German). Masters Thesis, University of Ulm, Germany.

Han, Y., Himminghofer, J., Schaaf, T., and Wikarski, D. (1996). Management of workflow resources to support runtime adaptability and system evolution. *Proc. Int. Conf. on Practical Aspects of Knowledge Management*. Basel, Switzerland.

Heimann, P., Joeris, G., Krapp, C., and Westfechtel, B. (1996). DYNAMITE: Dynamic task nets for software process management. *Proc. 18th Int. Conf. Software Engineering*. (pp. 331–341). Berlin, Germany.

Heinl, P., Schuster, H., and Stein, K. (1996). Behandlung von Ad-hoc-Workflows im MOBILE workflow-modell. *Proc. Softwaretechnik in Automation und Kommunikation—Rechnergesttzte Teamarbeit*. (pp. 229–242). Munich, Germany.

Heinlein, C. and Dadam, P. (1997). Interaction Expressions—A Powerful Formalism for Describing Inter-Workflow Dependencies. Technical Report No. 97–04, Department for Computer Science, University of Ulm, Germany.

Hennessy, M. (1989). *Algebraic Theory of Processes*, Cambridge: The MIT Press.

Hensinger, C. (1997). $ADEPT_{flex}$—Dynamic Modification of Workflows and Exception Handling in WFMSs (in German). Masters Thesis, University of Ulm, Germany.

Hofstede, A., Orlowska, M., and Rajapaks, J. (1996). Verification problems in conceptual workflow specifications. *Proc. 15th Int. Conf. on Conceptual Modeling*. (pp. 73–88). Cottbus, Germany: Springer.

Hsu, M. (Ed.) (1993). Special Issue on Workflow and Extended Transaction Systems, *IEEE Bulletin of the Technical Commitee on Data Engineering*, 16(2).

Hsu, M. (Ed.) (1995). Special Issue on Workflow Systems, *IEEE Bulletin of the Technical Commitee on Data Engineering*, 18(1).

Hsu, M. and Kleissner, C. (1996). ObjectFlow: Towards a Process Management Infrastructure, *Distributed and Parallel Databases*, 4, 169–194.

Kamath, M. and Ramamritham, K. (1996). Bridging the gap between transaction management and workflow management. *Proc. NSF Workshop on Workflow and Process Automation Inf. Sys.* Athens, Georgia.

Karbe, B., Ramsperger, N., and Weiss, P. (1990). Support of Cooperative Work by Electronic Circulation Folders, *SIGOIS Bulletin*, 11, 109–117.

Kirsch, M. (1996). Design and Implementation of a Graphical Tool for the Modeling and Animation of Flexible Workflows (in German). Masters Thesis, University of Ulm, Germany.

Kreifelts, T., Hinrichs, E., Klein, K.-H., Seuffert, P., and Woetzel, G. (1991). Experiences with the DOMINO office procedure system. *Proc. 2nd European Conf. on CSCW*. (pp. 117–130). Amsterdam, The Netherlands.

Kuhn, K., Reichert, M., Nathe, M., Beuter, T., and Dadam, P. (1994). An infrastructure for cooperation and communication in an advanced clinical information system. *Proc. 18th Symp. on Comp. in Med. Care*. (pp. 519–523). Washington: Hanley & Belfus, Medical Publisher.

Leymann, F. (1995). Supporting business transactions via partial recovery in workflow management systems. *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft*. (pp. 51–70). Dresden, Germany: Springer.

Leymann, F. and Altenhuber, W. (1994). Managing Business Processes as an Information Resource, *IBM Systems Journal*, 33, 326–348.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems-Specification*, Springer.

Meyer, J. (1996). Requirements for Future WFMSs: Flexibility, Exception Handling and Dynamic Changes in Clinical Processes (in German). Masters Thesis, University of Ulm, Germany.

Nastansky, L. and Ott, M. (1996). Teambasiertes Workflowmanagement und Analyse Prozeorientierter Teamarbeit im Bereich Zwischen Kooperativer und Strukturierter Vorgangsbearbeitung.Technical Report, Workgroup Computing Competence Center Paderborn, University of Paderborn, Germany.

Reichert, M., Kuhn, K., and Dadam, P. (1996). Process reengineering and process automation in clinical application environments (in German). *Proc. GMDS'96*. (pp. 219–223). Bonn, Germany: MMV Medizin Verlag.

Reichert, M. and Dadam, P. (1997a). A framework for dynamic changes in workflow-management systems. *Proc. 8th Int. Workshop on Database and Expert Systems Applications*. (pp. 42–48). Toulouse, France: IEEE Computer Society Press.

Reichert, M., Schultheiß, B., and Dadam, P. (1997b). Experiences with the development of process-oriented clinical application systems based on process-oriented workflow technology (in German). *Proc. GMDS'97*. (pp. 181–187). Ulm, Germany: MMV Medizin Verlag.

Reinwald, B. (1993). *Workflow-Management in Verteilten Systemen*, Stuttgart: Teubner.

Reuter, A. and Schwenkreis, F. (1995). ConTracts—A Low-Level Mechanism for Building General-Purpose Workflow Management Systems, *IEEE Bulletin of the Technical Committee on Data Engineering*, 18, 4–10.

Sheth, A., Georgakopoulos, D., Joosten, S., Rusinkiewicz, M., Scacchi, W., Wileden, J., and Wolf, A. (1996). Report from the NSF Workshop on Workflow and Process Automation in Information Systems, Technical Report No. UGA-CS-TR-96-003, University of Georgia.

Sheth, A. and Kochut, K. (1997). Workflow applications to research agenda: Scalable and dynamic work coordination and collaboration systems. *Proc. of the NATO Advanced Study Institute on WFMSs and Interoperability*. Istanbul, Turkey.

Siebert, R. (1996). Adaptive workflow for the german public administration. *Proc. 1st Int. Conf. on Practical Aspects of Knowledge Management—Workshop on Adaptive Workflow*. Basel, Switzerland.

Strong, D.M. and Miller, S.M. (1995). Exceptions and Exception Handling in Computerized Information Processes, *ACM Transactions on Inf. Sys.*, 13, 206–233.

Vogel, P. and Erfle, R. (1992). Backtracking office procedures. *Proc. 15th Int. Conf. on Database and Expert Systems*. (pp. 506–511). Valencia, Spain: Springer.

Weber, M., Partsch, G., Scheller-Huoy, A., Schweitzer, J., and Schneider, G. (1997). Flexible real-time meeting support for workflow management systems. *Proc. 30th Int. Conf. on System Sciences*. Maui, Hawaii: IEEE Computer Society Press.

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

Wodtke, D. and Weikum, G. (1997). A formal foundation for distributed workflow execution based on state charts. *Proc. Int. Conf. on Database Theory*. Delphi, Greece.

Worah, D. and Sheth, A. (1997). Transactions in Transactional Workflows. In S. Jajodia and L. Kerschberg (Eds.), *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers.