An Approach for Supporting Ad-hoc Modifications in Distributed Workflow Management Systems

Thomas Bauer
Dept. GR/EPD
Daimler AG
thomas.tb.bauer@daimler.com

Manfred Reichert
Information Systems Group
University of Twente
m.u.reichert@utwente.nl

Supporting enterprise-wide or even cross-organizational business processes is a characteristic challenge for any workflow management system (WfMS). Scalability at the presence of high loads as well as the capability to dynamically modify running workflow (WF) instances (e.g., to cope with exceptional situations) are essential requirements in this context. Should the latter one, in particular, not be met, the WfMS will not have the necessary flexibility to cover the wide range of process-oriented applications deployed in many organizations. Scalability and flexibility have, for the most part, been treated separately in the relevant literature thus far. Even though they are basic needs for a WfMS, the requirements related with them are totally different. To achieve satisfactory scalability, on the one hand, the system needs to be designed such that a workflow instance can be controlled by several WF servers that are as independent from each other as possible. Yet dynamic WF modifications, on the other hand, necessitate a (logical) central control instance which knows the current and global state of a WF instance. For the first time, this paper presents methods which allow ad-hoc modifications (e.g., to insert, delete, or shift steps) to be performed in a distributed WfMS; i.e., in a WfMS with partitioned WF execution graphs and distributed WF control. It is especially noteworthy that the system succeeds in realizing the full functionality as given in the central case while, at the same time, achieving extremely favorable behavior with respect to communication costs.

Keywords: Workflow Management, Dynamic Workflow Modification, Scalability, Distributed Workflow Execution

1 Introduction

For a variety of reasons, companies are developing a growing interest in changing their information systems such that they behave "process-oriented". That means to offer the right tasks, at the right point in time, to the right persons along with the information and the

application functions needed to perform these tasks. Workflow management systems (WfMS) offer a promising technology to achieve this goal [AH02, Fis00], since they allow computerized business processes to be run in a distributed system environment [LR00, SGW01]. Thus, workflow (WF) technology provides a powerful platform for implementing enterprise-wide as well as cross-organizational, process-oriented application systems (including e-services, like supply chain management, e-procurement, or customer relationship management).

The bottom line for any effective WfMS is that it must help to make large process-oriented application systems easy to develop and maintain. For this purpose, the application-specific code of a workflow-based application is separated from the process logic of the related business processes [LR00]. So instead of a large, monolithic program package, the result is a set of individual activities which represent the application programs. These (activity) programs can be implemented as isolated components that can expect that their input parameters are provided upon invocation by the run-time environment of the WfMS and which only have to worry about producing correct values for their output parameters. The process logic between activity programs is specified in a separate control flow and data flow definition. It sets out the order (sequence, branching, parallelism, loops) in which the individual activities are to be executed and it defines the data flow between them. For WF modeling, WfMS offer graphical process description languages, like e.g., Petri Nets [AH00], Statecharts [LS97], UML Activity Diagrams [DH01], or block-structured process graphs [MR00, RD98].

At run-time, new WF instances can be created from a process definition and then be executed according to the defined process logic. If a certain activity is to be executed, it is assigned to the worklists of the authorized users. Exactly which users are authorized to handle this activity is generally determined by the user role assigned to it [BFA99, Bu94].

1.1 Problem Description

Very often, a centralized WfMS shows deficits when it is confronted with high loads [KAGM96] or when the business processes to be supported span multiple organizations [DR99, LR07]. As in several other approaches (e.g. [CGP+96, MWW+98]), in the ADEPT project, we have met this particular demand by realizing a distributed WfMS made up of several WF servers. WF schemes may be divided into several partitions such that related WF instance may be controlled "piecewise" by different WF servers in order to obtain a favorable communication behavior [Bau01, BD97, BD00a].

A further common weakness of current WfMS is their lack of flexibility [HS98, MR00, RD98, SMO00, RRD04a, SO00, Wes98, WRR07, LR07]. Today's WfMS often do not adequately support ad-hoc modifications of in-progress WF instances, which become necessary, for example, to deal with exceptional situations [CFM99, DRK00, SM95, WRR07]. Therefore, ADEPT allows users (or agents) to dynamically modify a running WF instance based on high-level change patterns (e.g., to insert, delete, or move activities; for an overview see [WRR07]). As opposed to numerous other approaches, for the first time ADEPT has ensured that the WF instance remains consistent even after modification; i.e., there are no run-time errors and inconsistencies (e.g., deadlocks due to cyclic order relationships or program crashes due to activity invocations with missing input parameter data) [Rei00, DRK00, RRD04b, RD98].

In our previous work we considered the (distributed) execution of partitioned WF schemes and ad-hoc modifications as separate issues. In fact, we neglected to systematically examine how these two vital aspects of a WfMS interact. Typically such an investigation is not trivial as the requirements related to each of these two aspects are different: The performance of ad-hoc modifications and the correct processing of the workflow afterwards prescribe a logically central control instance to ensure correctness and consistency [RD98]. The existence of such a central instance, however, frustrates the accomplishments achieved by distributed WF execution. The reason for this is that a central component decreases the availability of the WfMS and increases the necessary communication effort between WF clients and the WF server. One reason for this lies in the fact that the central control instance must be informed of each and every change in the state of any WF instance. This state of the instance is needed to decide whether an intended modification is executable at all [RD98].

1.2 Contribution

The objective of this work is to enable ad-hoc modifications of single WF instances in a distributed WfMS; i.e., a WfMS with WF schema partitioning and distributed WF control. As a necessary prerequisite, distributed WF control must not affect the applicability of ad-hoc modifications; i.e., each modification, which is allowed in the central case, must be applicable in case of distributed WF execution as well. And the support of such ad-hoc modifications, in turn, must not impact distributed WF control. In particular, normal WF execution should not necessitate a great deal of additional communication effort due to the application of WF instance modifications. Finally, in the system to be developed, ad-hoc modifications should be correctly performed and as efficiently as possible.

In order to deal with these requirements, it is essential to examine which servers of the WfMS must be involved in the synchronization of an ad-hoc modification. Most likely we will have to consider those servers currently involved in the control of the respective WF instance. These active servers require the resulting *execution schema* of the WF instance (i.e., the schema and state resulting from the ad-hoc modification) in order to correctly control it after the modification. Thus we first need an efficient approach to determine the set of active servers for a given WF instance. This must be possible without a substantial expense of effort for communication. In addition, we must clarify how the new execution schema of the WF instance, generated as a result of the ad-hoc modification, may be transmitted to the relevant servers. An essential requirement is, thereby, that the amount of communication may not exceed acceptable limits.

The following section furnishes basic information on distributed WF execution and ad-hoc WF modifications in ADEPT – background information which is necessary for a further understanding of this paper. Section 3 describes how dynamic modifications are performed in a distributed WfMS, while Section 4 sets out how modified WF instances can be efficiently controlled in such a system. In Section 5 we discuss how the presented concepts have been implemented in the ADEPT WfMS prototype. We discuss related work in Section 6 and end with a summary and an outlook on future work.

2 Background Information

Within the ADEPT project [DRK00, RD98], we have investigated the requirements of enterprise-wide and cross-organizational workflow-based applications [DR99]. This section provides a brief summary of some of the concepts we developed for distributed WF control and ad-hoc modifications of single WF instances.

2.1 Distributed Workflow Execution in ADEPT

Usually, WfMS with one central WF server are unsuitable if the WF participants (i.e., the actors of the WF activities) are distributed across multiple enterprises or organizational units [DR99]. In such a case, the use of one central WF server would restrict the autonomy of the involved partners and might be disadvantageous with respect to respones times. Particularly, if the organizations are widespread, response times will significantly increase due to the long distance communication between WF clients and the WF server. In addition, owing to the

large number of users and co-active WF instances typical for enterprise-wide applications, the WfMS is generally subjected to an extremely heavy load [KAGM96, SK97]. This may lead to certain components of the system becoming overloaded. For all these reasons, in ADEPT, a WF instance may not be controlled by only one WF server. Instead, its related WF schema may be partitioned at buildtime (if favorable), and the resulting partitions be controlled "piecewise" by multiple WF servers during runtime[1] [Bau01, BD97] (see Figure 1). As soon as the end of a partition is reached at run-time, control over the respective WF instance is handed over to the next WF server (in the following we call this *migration*).

When performing such a migration, a description of the state of the WF instance has to be transmitted to the target server before this WF server can take over control. This includes, for example, information about the state of WF activities as well as values for WF relevant data; i.e., data elements connected with output parameters of activities. (To simplify matters, in this paper we assume that the WF templates (incl. their WF schemes) have been replicated and stored on all (relevant) WF servers of the distributed WfMS.)

To avoid unnecessary communication between WF servers, ADEPT allows to control parallel branches of a WF instance independently from each other – at least as no synchronization due to other reasons, e.g. a dynamic WF modification, becomes necessary. In the example given in Figure 1b, WF server s_3, which currently controls activity d, normally does not know how far execution has progressed in the upper branch (activities b and c). This has the advantage that the WF servers responsible for controlling the activities of parallel branches do not need to be synchronized.

The partitioning of WF schemes and distributed WF control have been successfully utilized in other approaches as well (e.g. [CGP+96, MWW+98]).In ADEPT, we have targeted an additional goal, namely the minimization of communication costs. Concrete experiences we gained in working with commercial WfMS have shown that there is a great deal of communication between the WF server and its WF clients, oftentimes necessitating the exchange of large amounts of data [End98]. This may lead to the communication system becoming overloaded. Hence, the WF servers responsible for controlling activities in ADEPT are defined in such a way that communication in the overall system is reduced: Typically, the WF server for the control of a specific activity is selected in a way such that it is located in the subnet to which most of the potential actors belong (i.e., the users whose role would allow them to

[1]To achieve a better scalability the ADEPT approach allows the same partition of different WF instances (of a particular WF type) being be controlled by multiple WF servers. Related concepts, however, are outside the scope of this paper and are presented in [BRD03].

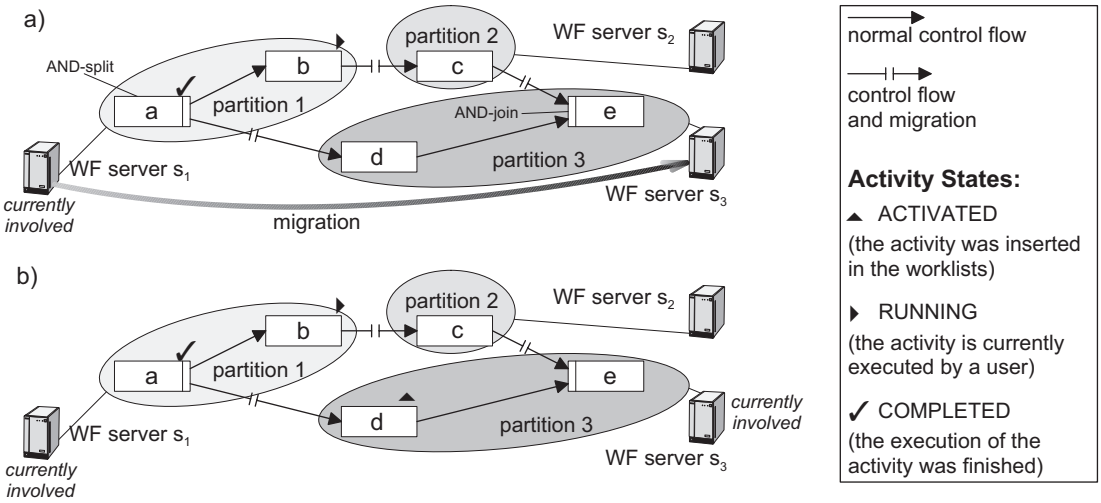

Figure 1: *a) Migration of a WF instance (from s_1 to s_3) and b) the resulting state of the WF instance.*

handle the activity). This way of selecting the server contributes to avoid cross-subnet communication between the WF server and its clients. Further benefits are improved response times and increased availability. This is achieved due to the fact that neither a gateway nor a WAN (Wide Area Network) is interposed when executing activities. Finally, the efficiency of the described approach – with respect to WF server load and communication costs – has been proven by means of comprehensive simulations (see [Bau01, BD99a, BD00b] for details).

Usually, WF servers are assigned to the activities of a WF schema already at build-time. However, in some cases this static approach does not suffice to achieve the desired results. This may be the case, for example, if *dependent actor assignments* become necessary. Such assignments indicate, for example, that an activity n has to be performed by the same actor as a preceding activity m. Consequently, the set of potential actors of activity n is dependent on the concrete actor assigned to activity m. Since this set of prospective actors can only be determined at run-time, it would be beneficial to wait with WF server assignment until run-time as well. Then, a server in a suitable subnet can be selected; i.e., one that is most favorable for the actors defined. For this purpose, ADEPT supports so-called *variable server assignments* [BD99b, BD00a]. Here, server assignment expressions like "server in subnet of the actor performing activity m" are assigned to activities and then evaluated at run-time. This allows the WF server, which is to control the related activity instance, to be determined dynamically.

2.2 Ad-hoc Workflow Modifications in ADEPT

To allow users to flexibly react in exceptional situations or to dynamically evolve the structure of in-progress WF instances, a WfMS must provide support for ad-hoc modifications of WF instances at run-time. With the ADEPT*flex* calculus, we developed in the ADEPT project, activities may be dynamically inserted, deleted, or shifted as desired and in a consistent manner [Rei00, RD98, RRD04b]. In fact, even very complex modifications may be carried out at a high semantic level. As an example consider the insertion of an activity which has to be executed after completing an arbitrary set of activities, and which must be finished before some other activities may be started [RD98]. However, we do not discuss the graph and state transformation formalism developed for dynamic WF modifications in this paper, as this is not relevant for its further understanding (for details see [Rei00, RD98]).

A simple example of an ad-hoc WF modification is shown in Figure 2. The depicted WF instance is modified by inserting a new activity x parallel to an existing one. Taking the (user) specification of the modification to be made, first of all, ADEPT checks whether the modification can be correctly performed or not; i.e., whether all correctness guarantees achieved by formal checks at build-time can be further ensured. If this is the case, ADEPT automatically calculates the set of *base operations* (e.g., insert activity, insert control edge) to be applied to the execution schema of the given WF instance. In addition, it automatically determines the new state of the WF instance in order to correctly proceed with the flow of control. In our example the state of the newly inserted activity x is automatically set to ACTIVATED; i.e., the corresponding task is immediately inserted into the worklists of potential actors.

As illustrated in Figure 2c, the calculated base operations, together with the change specification, are recorded in the *modification history* of the WF instance. This history will be required, for example, if the WF instance has to be partially rolled back [DRK00]. In ADEPT, the occurrence of modification events (and a reference to the corresponding modification history entry) is recorded in the *execution history* of the WF instance as well. As an example take the entry *DynModif(1)* in Figure 2b for the modification 1. Furthermore, the execution history contains other essential instance data, e.g., start / completion times of activities and information about the corresponding actors.

Ad-hoc modifications of WF instances during run-time may result in inconsistencies or errors if no further precautions are undertaken. First of all, any ad-hoc modification must result in a correct WF schema for the resepctive WF instance. For example, deleting an

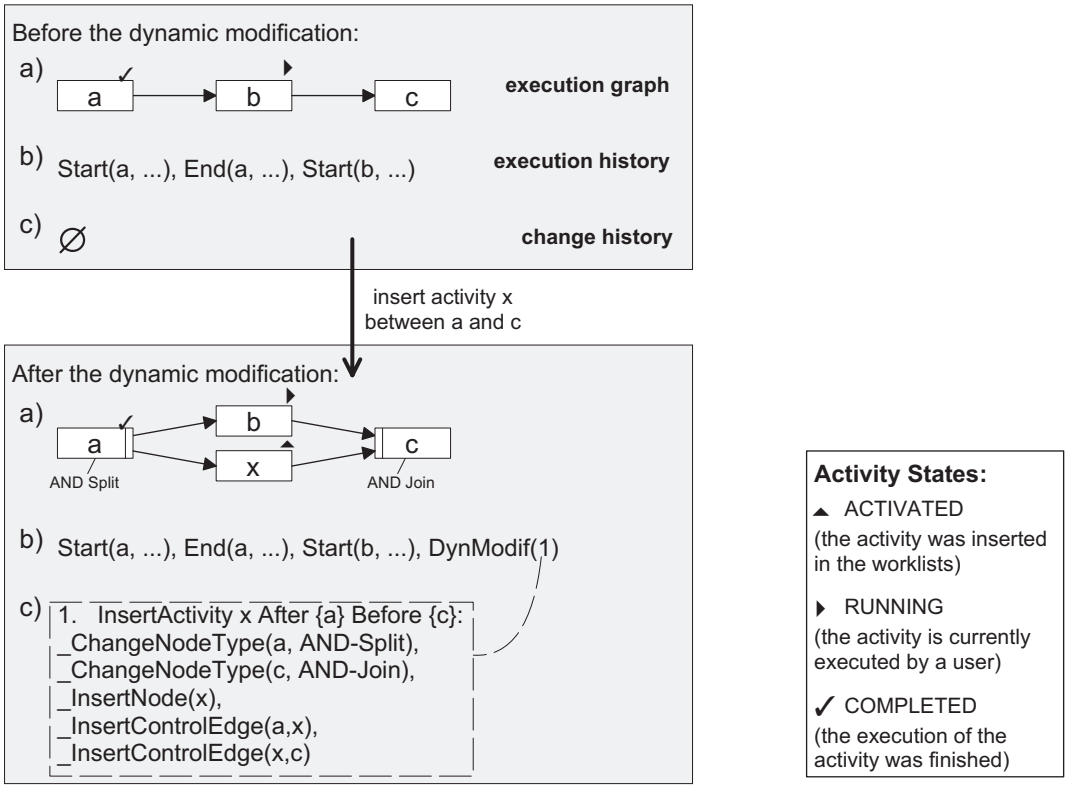

Figure 2: *(Simplified) example of an ad-hoc modification with a) WF execution schema, b) execution history, and c) modification history.*

activity may lead to incompleteness of the input data needed for subsequent activities. This, in turn, may cause activity program crashes or malfunctions when the associated application component is invoked. Or, if a control dependency is dynamically added, this may lead to "undesired" cyclic dependencies between activities, potentially causing a deadlock in the sequel [Rei00]. Besides such structural correctness properties, we have to ensure that the concerned WF instance is *compliant* with the new WF schema [RRD04a, RRD04b]; i.e., its previous execution could have been based on the new WF schema as well. This will not be the case, for example, if an activity is inserted into or deleted from an already processed region of the related WF schema. Generally, compliance is required to avoid inconsistent WF states (e.g., deadlocks, livelocks). ADEPT precludes such errors and ensures compliance. For this reason, formal pre- and post-conditions are defined for each change operation. They concern the state as well as the structure of the WF instance. Before introducing a modification, ADEPT analyzes whether it is permissible on the basis of the current state and structure of the WF instance; i.e., whether the (formally) defined pre- and post-conditions of the applied

change operations are fulfilled. Only if this is the case the structure and state of the WF execution graph are modified accordingly.

3 Ad-hoc Modifications in a Distributed WfMS

In principle, in a distributed WfMS ad-hoc modifications of single WF instances have to be performed just as in a central system: The WfMS has to check whether or not the desired modification is allowed on basis of the structure and state of the concerned WF instance. If the modification is permissible, the related base operations have to be determined and the WF schema belonging to the WF instance be modified accordingly (incl. adaptations of the state of WF activities if required).

To investigate whether an intended ad-hoc modification is permissible or not, the system first needs to know the current global state of the (distributed) WF instance (or at least relevant parts of it). As discussed in Section 2.1, in case of parallel executions, this state information may have to be retrieved from other WF servers as well. (For a description of how state data, i.e. WF control and WF relevant data [WMC99], may be efficiently transferred in a distributed WfMS we refer to [BRD01].)

This section describes a method for determining the set of WF servers on which the state information relevant for the applicability of a modification is located. In contrast to a central WfMS, in distributed WfMS it is generally not sufficient to modify the execution schema of the WF instance solely on the WF server responsible for controlling the modification. Otherwise, errors or inconsistencies may occur in the following, since other WF servers would use "out-of-date" schema and state information when controlling the WF instance. Therefore, in the following, we show which WF servers have to be involved in the modification procedure and how corresponding protocols have to look like.

3.1 Synchronizing Workflow Servers During Ad-hoc Modifications

An authorized user may invoke an ad-hoc modification on any WF server which controls the WF instance in question. Yet as a rule, this WF server alone may not always be able to correctly perform the modification. If other WF servers currently control parallel branches of the corresponding WF instance, state information from these WF servers may be needed as well. In addition, the WF server initiating the change process must also ensure that the corresponding modifications are taken over into the execution schemes of the respective WF

instance, which are being managed by these other WF servers. This becomes necessary to enable them to correctly proceed with the flow in the sequel (see below).

A naive solution would be to involve all WF servers of the WfMS by a broadcast. However, this approach is impractical in most cases as it is excessively expensive. In addition, all server machines of the WfMS must be available before an ad-hoc modification can be performed. Thus we have come up with three alternative approaches, which we explain and discuss below.

Approach 1: Synchronize all Servers Concerned by the WF Instance
This approach considers those WF servers which either have been or are currently active in controlling activities of the WF instance or which will be involved in the execution of future activities. Although the effort involved in communication is greatly reduced as compared to the naive solution mentioned above, it may still be unduly large. For example, communication with those WF servers which were involved in controlling the WF instance solely in the past (i.e., they will not participate again in the future) is superfluous. They do not need to be synchronized any more and the state information managed by these WF servers has already been migrated.

Approach 2: Synchronize all Current and Future Servers of the WF Instance
To be able to control a WF instance, a WF server needs to know its current WF execution schema. This, in turn, requires knowledge of all ad-hoc modifications performed so far. For this reason, a modification is relevant for those WF servers which either are currently active in controlling the WF instance or will be involved in controlling it in the future. Thus it seems to make sense to synchronize exactly these WF servers in the modification procedure.

However, with this approach, problems may arise in connection with conditional branches. For XOR-splits, which will be performed in the future, it cannot always be determined in advance which execution branch will be chosen. As different execution branches may be controlled by different WF servers, the set of relevant WF servers cannot be calculated immediately. Generally, it is only possible to calculate the set of the WF servers that will be potentially involved in this WF instance in the future.

The situation becomes even worse if variable server assignments (cf. Section 2.1) are used. Then, generally, for a given WF instance it is not possible to determine the WF servers that will be potentially involved in the execution of future activities. The reason for this is that the run-time data of the WF instance, which is required to evaluate the WF server assignment expressions, may not even exist at this point in time. For example, in Figure 3, during execution of activity g, the WF server of activity j cannot be determined since the

actor responsible for activity i has not been fixed yet. Thus the system will not always be able to synchronize future servers of the WF instance when an ad-hoc modification takes place. As these WF servers do not need to be informed about the modification at this time (since they do not yet control the WF instance), we suggest another approach.

Approach 3: Synchronize all Current Servers of the WF Instance

The only workable solution is to synchronize exclusively those WF servers currently involved in controlling the WF instance, i.e. the active WF servers. Generally, it is not trivial at all to determine which WF servers these in fact are. The reason is that in case of distributed WF control, for an active WF server of a WF instance the execution state of the activities being executed in parallel (by other WF servers) is not known. As depicted in Figure 3, for example, WF server s_4, which controls activity g, does not know whether migration $M_{c,d}$ has already been executed and, as a result, whether the parallel branch is being controlled by WF server s_2 or by WF server s_3. In addition, it is not possible to determine which WF server controls a parallel branch, without further effort, if variable server assignments are used. In Figure 3, for example, the WF server assignment of activity e refers to the actor of activity c, which is not known by WF server s_4.

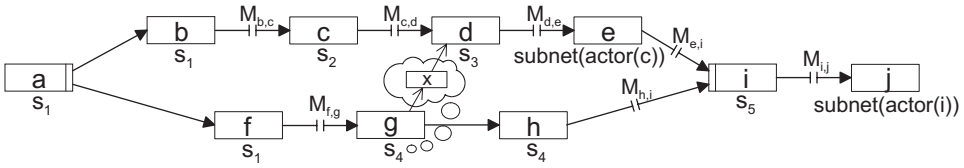

Figure 3: *Insertion of activity x between the activities g and d by the server s_4.*

In the following, we restrict our considerations to Approach 3.

3.2 Determining the Set of Active Servers of a Workflow Instance

As explained above, generally, a WF server is not always able to determine from its local state information which other WF servers are currently executing activities of a specific WF instance. And it is no good idea to use a broadcast call to search for these WF servers, as this would result in exactly the same drawbacks as described for the naive solution at the beginning of Section 3.1. We, therefore, require an approach for explicitly managing the active WF servers of a WF instance. The administration of these WF servers, however, should not be carried out by a fixed (and therefore central) WF server since this might lead to bottlenecks, thus negatively impacting the availability of the whole WfMS.

For this reason, in ADEPT, the set of active WF servers (*ActiveServers*) is managed by a *ServerManager* specific to the WF instance. For this purpose, for example, the start server of the WF instance can be used as the *ServerManager*. Normally, this WF server varies for each of the WF instances (even if they are of the same WF type), thus avoiding bottlenecks.[2] The start WF server can be easily determined from the (local) execution history by any WF server involved in the control of the WF instance. The following section shows how the set of active WF servers of a specific WF instance is managed by the *ServerManager*. Section 3.2.2 explains how this set is determined and how it is used to efficiently synchronize ad-hoc modifications.

3.2.1 Managing Active WF Servers of a WF Instance

As mentioned above, for the ad-hoc modification of a WF instance we require the set *ActiveServers*, which comprises all WF servers currently involved in the control of the WF instance. This set, which may be changed due to migrations, is explicitly managed by the *ServerManager*. Thereby, the following two rules have to be considered:

1. Multiple migrations of the same WF instance must not overlap arbitrarily, since this would lead to inconsistencies when changing the set of active WF servers.
2. For a given WF instance, the set *ActiveServers* must not change due to migrations during the execution of an ad-hoc modification. Otherwise, wrong WF servers would be involved in the ad-hoc modification or necessary WF servers would be left out.

As we will see in the following, we prevent these two cases by the use of several locks.[3]

In the following, we describe the algorithms necessary to satisfy these requirements. Algorithm 1 shows the way migrations are performed in ADEPT. It interacts with Algorithm 2 by calling the procedure *UpdateActiveServers* (remotely), which is defined by this algorithm. This procedure manages the set of active WF servers currently involved in the WF instance; i.e., it updates this set consistently in case of WF server changes.

[2]Using this policy, there may be scenarios where the same WF server would be always used, as all the WF instances in the WfMS are created on the same WF server. (An excellent example is the server that manages the terminals used by the tellers in a bank.) In this case, the *ServerManager* should be selected arbitrarily when a WF instance is generated.

[3]A secure behavior of the distributed WfMS could also be achieved by performing each ad-hoc modification and each migration (incl. the adaptation of the set *ActiveServers*) within a distributed transaction (with 2-phase-commit) [Dad96]. But this approach would be very restrictive since during the execution of such an operation, "normal WF execution" would be prevented. That means, while performing a migration, the whole WF instance would be locked and, therefore, even the execution of activities actually not concerned would not be possible. Such a restrictive approach is not acceptable for any WfMS. However, it is not required in our approach and we realize a higher degree of parallel execution while achieving the same security.

Algorithm 1 illustrates how a migration is carried out in our approach. It is initiated and executed by a source WF server that hands over control to a target WF server. First, the *SourceServer* requests a non-exclusive lock from the *ServerManager*, which prevents that the migration is performed during an ad-hoc modification.[4] Then an exclusive, short-term lock is requested. This lock ensures that the *ActiveServers* set of a given WF instance is not changed simultaneously by several migrations within parallel branches. (Both lock requests may be incorporated into a single call to save a communication cycle.)

The *SourceServer* reports the change of the *ActiveServers* set to the *ServerManager*, specifying whether it remains active for the concerned WF instance (*Stay*), or whether it will not be involved any longer (*LogOff*). If, for example, in Figure 3 the migration $M_{b,c}$ is executed before $M_{f,g}$, the option *Stay* will be used for the migration $M_{b,c}$ since WF server s_1 remains active for this WF instance. Thus, the option *LogOff* is used for the subsequent migration $M_{f,g}$ as it ends the last branch controlled by s_1. The (exclusive) short-term lock prevents that these two migrations may be executed simultaneously. This ensures that it is always clear whether or not a WF server remains active for a WF instance when a migration has ended. Next, the WF instance data (e.g., the current state of the WF instance, for details see [BRD01]) is transmitted to the target WF server of the migration. Since this is done after the exclusive short-term lock has been released (by *UpdateActiveServers*), several migrations of the same WF instance may be executed simultaneously. The algorithm ends with the release of the non-exclusive lock.

Algorithm 2 is used by the *ServerManager* to manage the WF servers currently involved in controlling a given WF instance. To fulfill this task, the *ServerManager* also has to manage the locks mentioned above. If the procedure *UpdateActiveServers* is called with the option *LogOff*, the source WF server of the migration is deleted from the set *ActiveServers(Inst)*; i.e., the set of active WF servers with respect to the given WF instance. The reason for this is that this WF server is no longer involved in controlling this WF instance. The target WF server for the migration, however, is always inserted into this set independently of whether it is already contained or not because this operation is idempotent.

The short-term lock requested by Algorithm 1 before the invocation of *UpdateActiveServers* prevents Algorithm 2 from being run in parallel more than once for a given WF in-

[4]For details see Algorithm 3. The lock does not prevent several migrations of one and the same WF instance from being performed simultaneously.

[5]$p() \rightarrow s$ means that procedure p is called and then executed by server s.

Algorithm 1 (Performing a Migration)
input
Inst: ID of the WF instance to be migrated
SourceServer: source server of the migration (it performs this algorithm)
TargetServer: target server of the migration
begin
// calculate the *ServerManager* for this WF instance by the use of its execution history
ServerManager = StartServer(Inst);
// request a non-exclusive lock and an exclusive short-term lock from the *ServerManager*
RequestSharedLock(Inst) → *ServerManager*;[5]
RequestShortTermLock(Inst) → *ServerManager*;
// change the set of active servers (cf. Algorithm 2)
if *LastBranch(Inst)* **then**
// the migration is performed for the last execution branch of the WF instance, that is active at the
// *SourceServer*
UpdateActiveServers(Inst, SourceServer, LogOff, TargetServer) → *ServerManager*;
else // another execution path is active at *SourceServer*
UpdateActiveServers(Inst, SourceServer, Stay, TargetServer) → *ServerManager*;
// perform the actual migration and release the non-exclusive lock
MigrateWorkflowInstance(Inst) → *TargetServer*;
ReleaseSharedLock(Inst) → *ServerManager*;
end.

stance. This helps to avoid an error due to overlapping changes of the set *ActiveServers(Inst)*. When this set has been adapted, the short-term lock is released.

Algorithm 2 (*UpdateActiveServers*: Managing the Active WF Servers)
input
Inst: ID of the affected WF instance
SourceServer: source server of the migration
Option: shows, if the source server will be involved in the WF instance furthermore (*Stay*), or not (*LogOff*)
TargetServer: target server of the migration
begin
// update the set of the current WF servers of the WF instance *Inst*
if *Option = LogOff* **then**
ActiveServers(Inst) = ActiveServers(Inst) − {SourceServer};
ActiveServers(Inst) = ActiveServers(Inst) ∪ {TargetServer};
// release the short-term lock
ReleaseShortTermLock(Inst);
end.

3.2.2 Performing Ad-hoc Modifications

Where the previous section has described how the *ServerManager* handles the set of currently active WF servers for a particular WF instance, this section sets out how this set is utilized when ad-hoc modifications are performed.

First of all, if no parallel branches are currently executed, trivially, the set of active WF servers contains exactly one element, namely the current WF server. This case may be detected by making use of the state and structure information (locally) available at the current WF server. The same applies to the special case that currently all parallel branches are controlled by the same WF server. In both cases, the method described in the following is not needed and therefore not applied. Instead, the WF server currently controlling the WF instance performs the ad-hoc modification without consulting any other WF server. Consequently, this WF server must not communicate with the *ServerManager* as well. For this special case, therefore, no additional synchronization effort occurs (when compared to the central case).

We now consider the case that parallel branches exist; i.e., an ad-hoc modification of the WF instance may have to be synchronized between multiple WF servers. The WF server which coordinates the ad-hoc modification then requests the set *ActiveServers* from the *ServerManager*. When performing the ad-hoc modification, it is essential that this set is not changed due to concurrent migrations. Otherwise, wrong WF servers would be involved in the modification procedure. In addition, it is vital that the WF execution schema of the WF instance is not restructured due to concurrent modifications, since this may result in the generation of an incorrect schema.

To prevent either of these faults we introduce Algorithm 3. It requests an exclusive lock from the *ServerManager* to avoid the mentioned conflicts. This lock corresponds to a write lock [GR93] in a database system and is incompatible with read locks (*RequestSharedLock* in Algorithm 1) and other write locks of the same WF instance. Thus, it prevents that migrations are performed simultaneously to an ad-hoc modification of the WF instance.

As soon as the lock has been granted, a query is sent to acquire the set of active WF servers of this WF instance.[6] Then a lock is requested at all WF servers belonging to the set *ActiveServers* in order to prevent local changes to the state of the WF instance. Any activities already started, however, may be finished normally since this does not affect the applicability of an ad-hoc modification. Next the (locked) state information is retrieved from all active WF servers. Remember that the resulting global and current state of the WF instance is required to check whether the ad-hoc modification to be performed is permissible (cf. Section 2.2). In Figure 3, for example, WF server s_4, which is currently controlling activity g and which wants to insert activity x after activity g and before activity d, normally does not know

[6]This query may be combined with the lock request into a single call to save a communication cycle.

Algorithm 3 (Performing an Ad-hoc Modification)
input
Inst: ID of the WF instance to be modified
Modification: specification of the ad-hoc modification
begin
// calculate the *ServerManager* for this WF instance
ServerManager = StartServer(Inst);
// request an exclusive lock from the *ServerManager* and calculate the set of active WF servers
RequestExclusiveLock(Inst) $\rightarrow$ *ServerManager*;
ActiveServers = GetActiveServers(Inst) $\rightarrow$ *ServerManager*;
// request a lock from all servers, calculate the current WF state, and perform the change (if possible)
for each Server $s \in$ *ActiveServers* **do**
RequestStateLock(Inst) $\rightarrow s$;
GlobalState = GetLocalState(Inst);
for each Server $s \in$ *ActiveServers* **do**
LocalState = GetLocalState(Inst) $\rightarrow s$;
GlobalState = GlobalState $\cup$ *LocalState*;
if *DynamicModificationPossible(Inst, GlobalState, Modification)* **then**
for each Server $s \in$ *ActiveServers* **do**
PerformDynamicModification(Inst, GlobalState, Modification) $\rightarrow s$;
// release all locks
for each Server $s \in$ *ActiveServers* **do**
ReleaseStateLock(Inst) $\rightarrow s$;
ReleaseExclusiveLock(Inst) $\rightarrow$ *ServerManager*;
end.

the current state of activity d (from the parallel branch). Yet the ad-hoc modification is permissible only if activity d has not been started at the time the modification is initiated [RD98]. If this is the case, the modification is performed at all active WF servers of the WF instance (*PerformDynamicModification*). Afterwards, the locks are released and any blocked migrations or modification procedures may then be carried out.

3.3 Illustrating Example

How migrations and ad-hoc modifications work together is explained by means of an example. Figure 4a shows a WF instance, which is currently controlled by only one WF server, namely the WF server s_1. Figure 4b shows the same WF instance after it migrated to a second WF server (s_2). In Figure 4c the execution was continued. One can also see that each of the two WF servers must not always possess complete information about the global state of the WF instance.

Assume now that an ad-hoc modification has to be performed, which is coordinated by the WF server s_1. Afterwards, both WF servers shall possess the current schema of the WF instance to correctly proceed with the flow of control. With respect to the (complete) current

state of the WF instance, it is sufficient that it is known by the coordinator s_1 (since only this WF server has to decide on the applicability of the desired modification). The other WF server only carries out the modification (as specified by WF server s_1).

4 Distributed Execution of a Modified Workflow Instance

If a migration of a WF instance has to be performed, its current state has to be transmitted to the target WF server. In ADEPT, this is done by transmitting the relevant parts of the execution history of the WF instance together with the values of WF relevant data (i.e., data elements used as input and output data of WF activities or as input data for branching and loop conditions) [BRD01].

If an ad-hoc modification was previously performed, the target WF server of a migration also needs to know the modified execution schema of the WF instance in order to be able to control the WF instance correctly. In the approach introduced in the previous section, only the active WF servers of the WF instance to be modified have been involved in the modification. As a consequence, the WF servers of subsequent activities, however, still have to be informed about the modification. In our approach, the necessary information is transmitted upon migration of the WF instance to the WF servers in question. Since migrations are rather frequently performed in distributed WfMS, this communication needs to be performed efficiently. Therefore, in Section 4.1 we introduce a technique which fulfills this requirement to a satisfactory degree. Section 4.2 presents an enhancement of the technique that precludes redundant data transfer.

4.1 Efficient Transmission of Information About Ad-hoc Modifications

In the following, we examine how a modified WF execution schema can be communicated to the target WF server of a migration. The key objective of this investigation is the development of an efficient technique that reduces communication-related costs as far as possible.

Of course, the simplest way to communicate the current execution schema of the respective WF instance to the migration target server is to transmit this schema in whole. Yet this technique burdens the communication system unnecessarily because related WF graph of this WF schema may comprise a large number of nodes and edges. This results in an enormous amount of data to be transferred – an inefficient and cost-intensive approach.

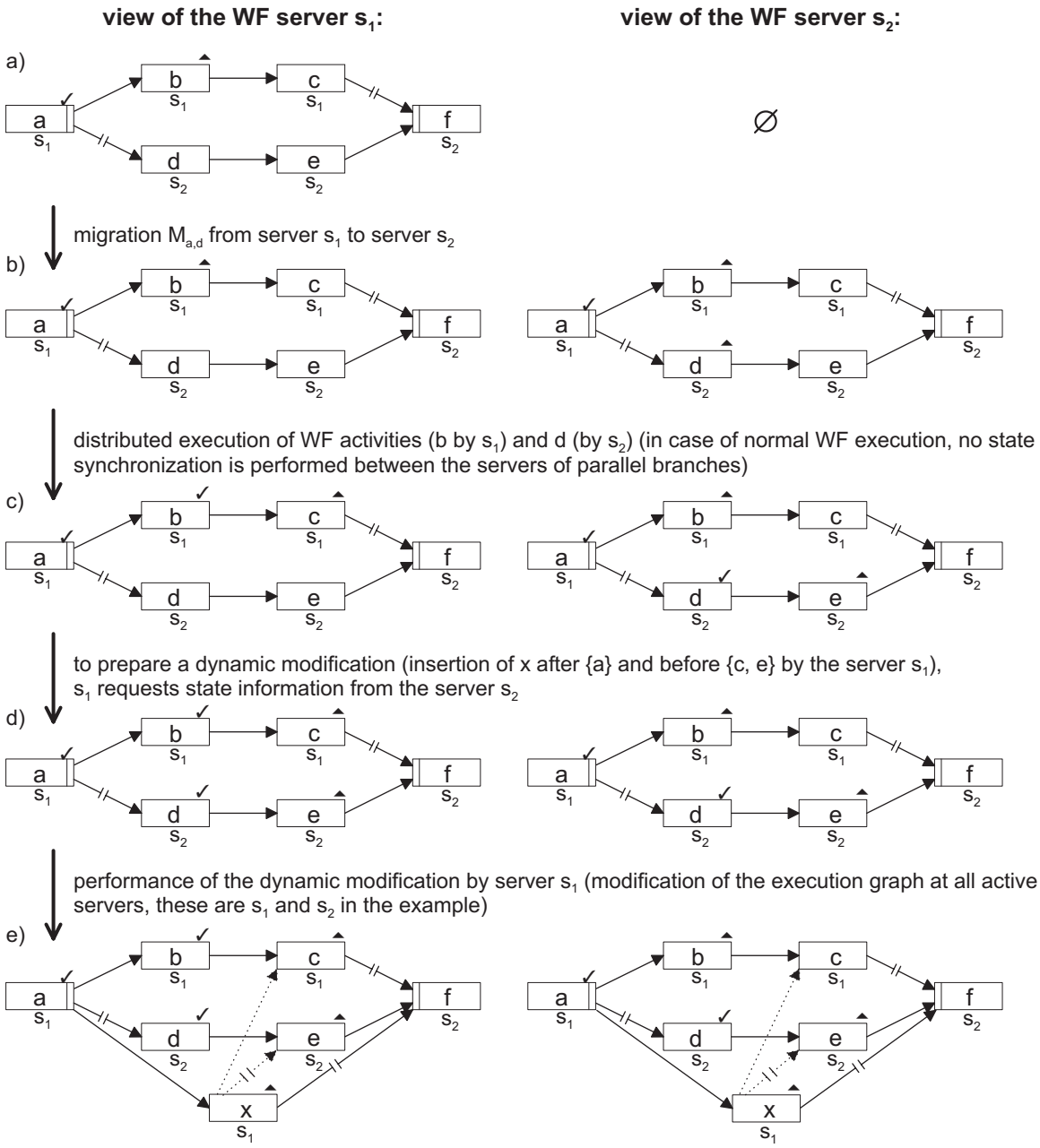

Figure 4: *Effects of migrations and ad-hoc modifications on the (distributed) execution schema of a WF instance (local view of the WF servers).*

Apart from this, the entire execution schema does not need to be transmitted to the migration target server as the related WF template has been already located there. (Note that a WF template is being deployed to all relevant WF servers before any WF instance may be created from it.) In fact, in most cases the current WF schema of the WF instance is almost identical to the WF schema associated with the WF template. Thus it is more efficient to transfer solely the relatively small amount of data which specifies the modification operation(s) applied to the WF instance. It would therefore seem practical to use the modification history (cf. Section 2.2) for this purpose. In the ADEPT*flex* model, the migration target server needs this history anyway [RD98], so that its transmission does not lead to an additional effort. When the base operations recorded in the modification history are applied to the original WF schema of the WF template, the result is the current WF schema of the given WF instance. This simple technique dramatically reduces the effort necessary for communication. In addition, as typically only very few modifications are performed on any individual WF instance, computation time is kept to a minimum.

4.2 Enhancing the Method Used to Transmit Modification Histories

Generally, one and the same WF server may be involved more than once in the execution of a WF instance – especially in conjunction with loops. In the example from Figure 5, for instance, WF server s_1 hands over control to WF server s_2 after completion of activity b but will receive control again later in the flow to execute activity d. Since each WF server stores the modification history until being informed that the given WF instance has been completed, such a WF server s already knows the history entries for the modifications it has performed itself. In addition, s knows any modifications that had been effected by other WF servers before s handed over the control of the WF instance to another WF server for the last time. Hence the data related to this part of the modification history need not be transmitted to the WF server. This further reduces the amount of data required for the migration of the "current execution schema".

4.2.1 Transmitting Modification History Entries

An obvious solution for avoiding redundant transfer of modification history entries would be as follows: The migration source server determines from the existing execution history exactly which modification the target WF server must already know. The related entries are then simply not transmitted when migrating the WF instance. In the example given in Figure 5,

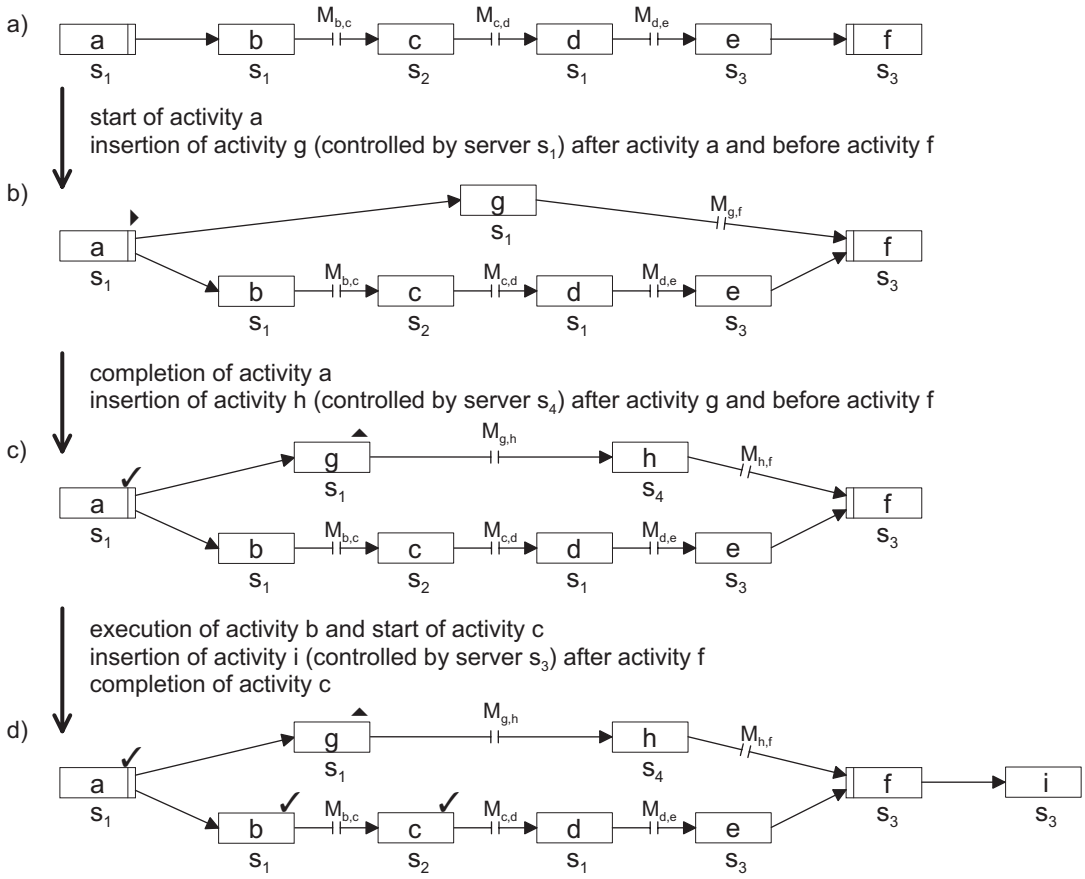

Figure 5: *a-d) WF instance and e) Execution history of WF server s_2 after completion of activity c. – In case of distributed WF control, with each entry the execution history records the WF server responsible for the control of the corresponding activity (additionally to the data values set out in Section 2.2).*

WF server s_2 can determine, upon ending activity c, that the migration target server s_1 must already know the modifications 1 and 2. In the execution history (cf. Figure 5e), references to these modifications (*DynModif(1)* and *DynModif(2)*) have been recorded before the entry *End(b, s_1, ...)* (which was logged when completing activity b). As this activity was controlled by WF server s_1, this WF server does already know the modifications 1 and 2. Thus, for the migration $M_{c,d}$, only the modification history entry corresponding to modification 3 needs to be transmitted. The transmitted part of the modification history is concatenated with the part already present at the target server before this WF server generates the new execution schema and proceeds with the flow of control.

In some cases, however, redundant transfer of modification history data cannot be avoided with this approach: As an example take the migrations $M_{d,e}$ and $M_{h,f}$ to the WF server s_3. For both migrations, with the above approach, all entries corresponding to modifications 1, 2, and 3 must be transmitted because the WF server s_3 was not involved in executing the WF instance thus far. The problem is that the migration source servers s_1 and s_4 are not able, from their locally available history data, to derive whether the other migration from the parallel branch has already been effected or not. For this reason, the entire modification history must be transmitted. Yet with the more advanced approach set out in the next section, we can avoid such redundant data transfer.

4.2.2 Requesting Modification History Entries

To avoid redundant data transmissions as described in the previous section, we now introduce a more sophisticated method. With this method, the necessary modification history entries are explicitly requested by the migration target server. When a migration takes place, the target WF server informs the source WF server about the history entries it already knows. The source WF server then only transmits those modification history entries of the respective WF instance which are yet missing on the target server. In ADEPT, a similar method has been used for transmitting execution histories; i.e., necessary data is provided on basis of a request from the migration target server (see [BRD01]). Here, no additional effort is expended for communication, since both, the request for and the transmission of modification history entries may be carried out within the same communication cycle.

With the described method, requesting the missing part of a modification history is efficient and easy to implement in our approach. If the migration target server was previously involved in the control of the WF instance, it already possesses all entries of the modification history up to a certain point (i.e., it knows all ad-hoc modifications that had been performed before this server handed over control the last time). But from this point on, it does not know any further entries. It is thus sufficient to transfer the ID of the last known entry to the migration source server to specify the required modification history entries. The source WF server then transmits all modification history entries made after this point.

The method set out above is implemented by means of Algorithm 4, which is executed by the migration source server as part of the *MigrateWorkflowInstance* procedure (cf. Algorithm 1). This procedure also effects transmission of the execution history and of WF relevant data (cf. [BRD01]). Algorithm 4 triggers the transmission of the modification history by re-

questing the ID of the last known modification history entry from the target WF server. If no modification history for the given WF instance is known at the target WF server, it returns *NULL*. In this case, the entire modification history is relevant for the migration and is transmitted to the target WF server. Otherwise, the target WF server requires only that part of the modification history, which follows the specified entry. This part is copied into the history *RelevantModificationHistory* and transmitted to the target WF server. This data may be transmitted together with the above-mentioned WF instance related data to save a communication cycle.

Algorithm 4 (Transmission of Modification History Data)
input
Inst: ID of the WF instance to be modified
TargetServer: server, which receives the modification history
begin
// start the transmission of the modification history by asking for the ID of the last known entry
LastEntry = GetLastEntry(Inst) → *TargetServer*;
// calculate the relevant part of the modification history
if *LastEntry = NULL* **then** // modification history is totally unknown at the target WF server
Relevant = True;
else // all entries until *LastEntry* (incl.) are known by the target server
Relevant = False;
// initialize the position counters for the original and the new modification history
$i = 1$; $j = 1$;
// read the whole modification history of the WF instance *Inst*
while *ModificationHistory(Inst)[i]* ≠ *EOF* **do**
if *Relevant = True* **then** // put the entry in the result (if necessary)
RelevantModificationHistory[j] = ModificationHistory(Inst)[i];
$j = j + 1$;
// check, if the end of that part of the modification history, that is known by
// the target WF server, is reached
if *EntryID(ModificationHistory(Inst)[i]) = LastEntry* **then** *Relevant = True*;
$i = i + 1$;
// perform the transmission of the modification history
TransmitModification(Inst, RelevantModificationHistory) → *TargetServer*;
end.

Algorithm 4 is illustrated by means of the example given in Figure 5: Concerning the migration $M_{c,d}$ the target WF server s_1 already knows the ad-hoc modifications 1 and 2. Thus it responds to the source server's request with *LastEntry = 2*. The migration source server then ignores the modification history entries 1 and 2, transmitting only the entry 3 to the target WF server s_1. This result is identical to that achieved in the approach presented in Section 4.2.1.

For the migrations $M_{h,f}$ and $M_{d,e}$, without loss of generality, it is assumed that migration $M_{h,f}$ is executed before $M_{d,e}$.[7] Since there is no modification history of this WF instance located on WF server s_3 yet, the target WF server of the migration $M_{h,f}$ returns *LastEntry = NULL*. Therefore, the entire modification history is transmitted to s_3. In the subsequent migration $M_{d,e}$, the target WF server s_3 then already knows the modification history entries 1 - 3, so that *LastEntry = 3* is returned in response to the source server query. (When the *while* loop in Algorithm 4 is run, the variable *Relevant* is not set to *True* until entries 1 - 3 have been processed. Since there exist no further entries in the modification history, *RelevantModificationHistory* remains empty with the result that no modification history entries have to be transmitted.) The problem of redundant data transfer, as set out in Section 4.2.1, is thus avoided here.

To sum up, with our approach not only ad-hoc modifications can be performed efficiently in a distributed WfMS (see Section 3), transmission costs for migration of modified WF instances may also be kept very low.

5 Implementation

All of the methods presented in this paper have been implemented a powerful proof-of-concept WfMS prototype [Zei99]. The ADEPT prototype demonstrates the feasibility of ad-hoc modifications in a distributed WfMS and it shows how the related concepts work in conjunction with other important WF features (e.g., handling of temporal as well as security constraints). The prototype has been completely implemented in Java, for communication Java RMI has been used.

5.1 Build-Time Clients

The ADEPT prototype supports the WF designer by powerful build-time clients. They enable the definition of workflow and activity templates, the modeling of organizational entities (and their relationships), the specification of security constraints (e.g., authorizations with respect to WF modifications), and the plug-in of application components. All relevant information is stored in the ADEPT database. In addition, XML-based descriptions of model data may be

[7] A lock at the target WF server prevents the migrations from being carried out concurrently in an uncoordinated manner. This ensures that migrations for one and the same WF instance are serialized; i.e., the lock is maintained from start of migration, while modification history entries (and other WF-related data [BRD01]) are acquired and transmitted, until the entries have finally been integrated into the modification history at the target WF server. This lock prevents history entries from being requested redundantly due to the request being based on obsolete local information.

generated; e.g., to export WF models to other tools or to exchange them between different WF servers of the distributed WfMS.

For WF modeling, ADEPT offers a syntax-driven, graphical WF editor. A sample screen is depicted in Figure 6. It shows a clinical workflow as it is modeled in ADEPT. In the upper part of the screen the control flow is shown, whereas the lower part displays the input parameters of the currently selected activity "calculate dose" (incl. the mapping of these parameters to global data elements). Additional information about this activity is shown on the right side. Further down, a placemarker box is displayed, which helps the WF designer to navigate through the WF model.

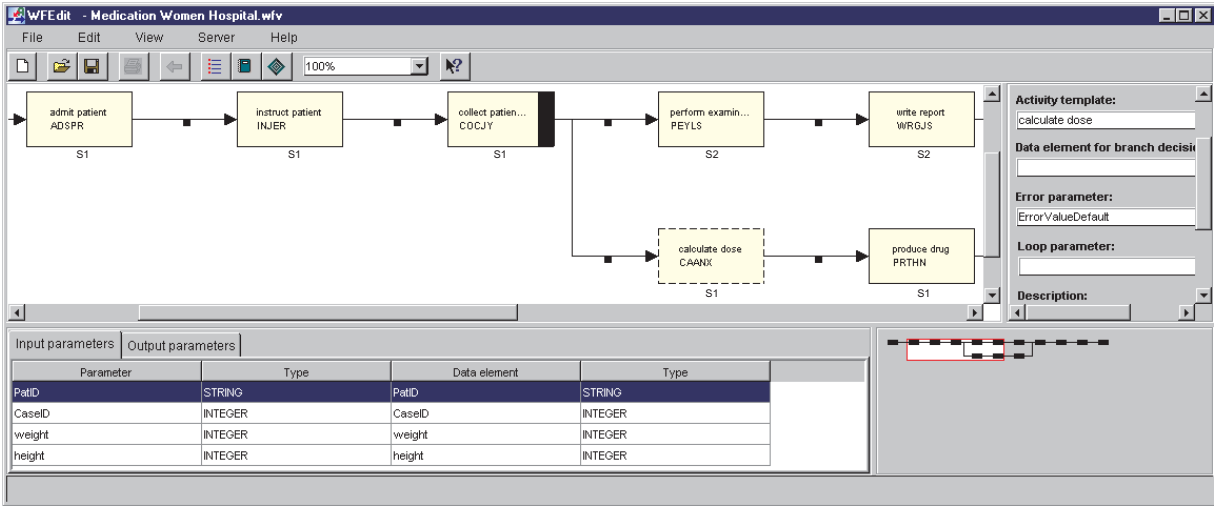

Figure 6: *Graphical WF editor.*

We explain the workflow example from Figure 6 in more detail, since we will refer to it in the following. The displayed WF model describes the medication of a patient during a treatment cycle in a hospital. A corresponding WF instance begins with the patient's admission to a ward (by a ward sister). It then proceeds with activities "instruct patient" (ward doctor) and "collect patient data" (ward sister). Afterwards, there is a split into two execution branches which may run parallel to each other. The upper branch sets out the activities of a medical examination in another department ("perform examination" and "write report", both with user role "radiology doctor"), whereas the lower branch defines preparatory steps performed by the ward (e.g., "calculate dose", "produce drug"). These two branches contain some other activities ("read report", "validate dose") which are not displayed in Figure 6. When both branches are finished, the produced drug will be given

to the patient, some aftercare will be provided, and the patient will be discharged (also not displayed in Figure 6).

ADEPT supports the WF designer in calculating optimal WF server assignments for the WF activities; i.e., in partitioning the WF graph such that the overall communication costs will be minimized at run-time (cf. Section 2.1). For this purpose, we have implemented sophisticated algorithms which make use of the information from the organizational database (e.g., the roles and locations of the users). Concerning our example from Figure 6, corresponding WF instances are controlled by the WF servers s_1 and s_2. (The calculated WF server assignments are displayed below the activity nodes. Accordingly, activities "perform examination" and "write report" are controlled by WF server s_2, whereas all other activities are carried out by WF server s_1.)

It is worth mentioning that the WF editor supports the WF designer in modeling error-free WF templates (e.g., exclusion of deadlocks, proper invocation of activity components, consistency of temporal constraints). To achieve this, it enables both, on-the-fly checks during WF editing and complete model checks initiated by the designer. In any case, a new WF template may only be released, if all correctness and consistency checks are successful. Note that this is very important in the context of ad-hoc modifications for which the WfMS can only guarantee consistency, if the WF instance was consistent before the modification as well. This, in turn, is crucial for the WfMS to guarantee a reliable and secure execution behavior of the (distributed) WF instances.

A new release of a WF template is introduced by deploying it to all relevant WF servers. For this, an XML-based description is sent to these servers, which is then imported into the corresponding run-time databases. – We omit descriptions of other build-time components (e.g. the ADEPT application integration tool), since they are not relevant in the context of this paper.

5.2 Run-Time Clients

We have implemented several run-time clients for end users as well as for system and process administrators. They provide support for the configuration of the distributed WfMS, the handling of WF instances, the handling of worklists, the run-time definition of ad-hoc modifications, and the monitoring of WF instances.

To monitor the progress of in-progress WF instances and to demonstrate the effects of ad-hoc modifications, ADEPT offers a special monitoring client. It allows authorized users

(e.g. the process administrator) to visualize the execution schema of a WF instance, together with the information related to that WF instance. A sample screen is depicted in Figure 7. It shows the execution schema of a WF instance which was created from the WF template as defined in Figure 6. The activities "admit patient", "instruct patient", and "collect patient data" have been already completed (this is indicated by the symbol $\surd$), whereas the activity "calculate dose" is currently activated (indicated by the symbol $\square$). The screen from Figure 7 also displays the data elements read and written by the currently selected activity ("calculate dose" in the example) as well as detailed information about this activity (e.g., activity state, actor assignment, execution mode, server assignment, earliest/latest starting times, etc.). All relevant information is managed by the WF server which controls this activity (s_1 in the example). A client program can access it by using the ADEPT API (application programming interface).

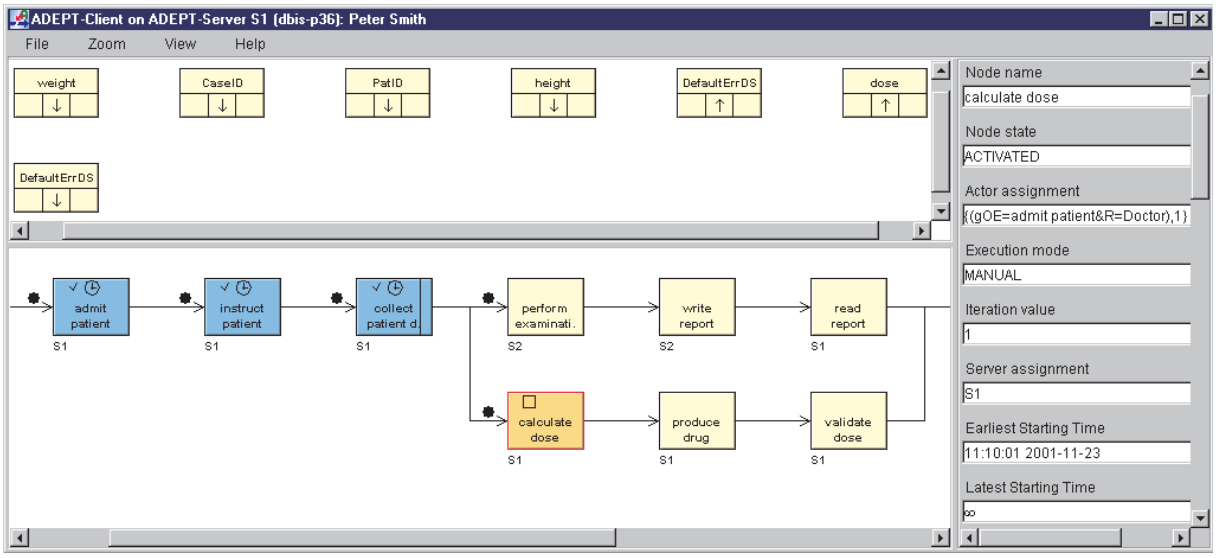

Figure 7: *ADEPT monitoring client (before the ad-hoc modification of the WF instance).*

Actually, the monitoring client depicted in Figure 7 only shows the WF execution schema from the viewpoint of WF server s_1 (to which it is connected). But this WF server does not know how far the execution has proceeded in the upper branch of the parallel branching (which is currently controlled by WF server s_2). For example, WF server s_1 does not know whether the activity "perform examination" has been already activated, started, or completed.

Let us now discuss how an ad-hoc modification can be realized in the given example. First of all, end users must be able to define such a modification at a high semantic level; i.e., without requiring that they are familiar with a WF modeling tool or that they have

knowledge about the distribution of the WF instance. For this, ADEPT offers advanced client programs to the end users, which are simple and easy to use.[8] We now return to our example from Figure 7. Assume that an authorized user (who is connected to WF server s_1) wants to insert the new activity "perform allergy test" after activity "instruct patient" and before the activities "write report" and "produce drug"; i.e., the user wishes that the allergy test shall be performed after the patient was instructed, but has to be completed before a report will be written and the drug will be produced. If this modification is applied to the WF instance from Figure 7, for example, the execution schema as shown in Figure 8 will result. (The node $n1$ depicted in Figure 8 represents an AND-split node resulting from the modifcation [RD98].)

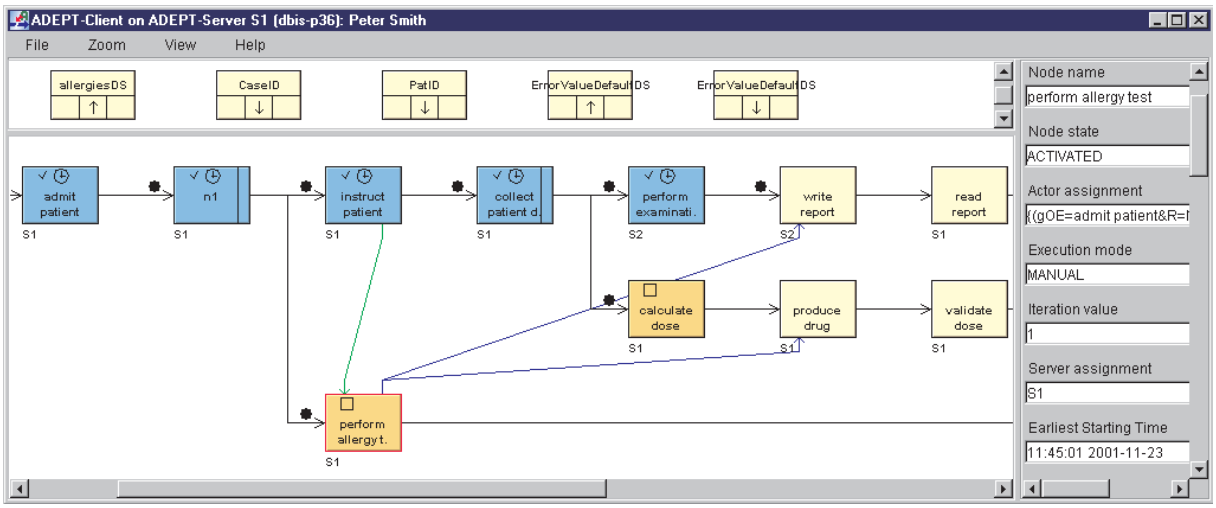

Figure 8: *ADEPT$_{workflow}$ monitoring client (after the ad-hoc modification).*

6 Discussion

In the WF literature, we find numerous approaches dealing with issues related to scalability and distributed WF execution. Besides central approaches, which include most commercial systems (e.g. MQSeries Workflow[9] [IBM99] and Staffware [Sta99]), several distributed WfMS, consisting of multiple WF servers, exist. Some of them assign a WF instance always to the same WF server (as a whole). Examples are Exotica/Cluster [AKA+94] and MOBILE [Jab97]

[8] To enable application developers to implement such programs, ADEPT provides a powerful programming interface to them. Its functionality goes far beyond the standards as defined by the Workflow Management Coalition [WMC98]. In particular, ADEPT comprises powerful modification operations, which hide much of the complexity of an ad-hoc modification from the client programmer.

[9] MQSeries Workflow uses multiple servers and it allows remote execution of sub-processes. Therefore, it is not a purely central approach.

(which was extended in [SNS99], see below). Comparable to ADEPT, MENTOR [MWW+98] and WIDE [CGP+96] select the WF server for a WF activity "next" to its potential actors. CodAlf, BPAFrame (both [SM96]), and METEOR$_2$ [SK97] allocate the WF server for a WF activity on the node where its corresponding application program is located. Completely distributed WfMS, like Exotica/FMQM [AMG+95] and INCAs [BMR96], use the machines of the actors as WF servers. Finally, there are approaches for distributed WF management, which do not have a special strategy for distributing the activities to the WF servers (e.g., EVE [GT98], METUFlow [DGA+97], MOKASSIN [GJS+99], WASA$_2$ [Wes99], and the Petri-net based approach presented in [GLO98]). A more comprehensive overview of distributed WfMS can be found in [Bau01, BD99a].

A great number of publications treat ad-hoc modifications. They concentrate on different issues arising in this context. Like ADEPT [RD98], Chautauqua [EM97], WASA$_2$ [Wes98], and WF nets [Aal01a] deal with issues related to the correctness and consistency of modified WF instances. This also applies to the approach described in [SMO00], which does also address temporal issues in connection with ad-hoc modifications. CoMo-Kit [DMP97] and AgentWork [MR00] use knowledge-based techniques to increase the flexibility in WfMS. In particular, AgentWork sets out an approach for automatically modifying WF instances at the occurrence of observable exceptions. The approach set out in [Aal01b] proposes generic WF models to deal with WF modifications. A generic WF model describes a family of WF models / variants of the same WF type. Consequently, an (ad-hoc) modification is handled by migrating a WF instance between different members of the same process family. This is supported by defining a minimal representative for each process family and by specifying rules for transferring from a variant to the minimal representative (and vice versa). An approach based on inheritance, which uses generic inheritance-preserving transformation and transfer rules, is suggested by the same author in [AB02]. With this approach, semantic errors in connection with ad-hoc modifications (e.g., an undesired swapping of activities) can be avoided by choosing appropriate inheritance notions. A powerful approach for the methodical restructuring of long-running transactions is offered by TAM [LP98]. Finally, there are several approaches aiming at the support of WF WF schema evolution and the propagation of the resulting modifications to already running WF instances (if compliant to the new scheme). Corresponding work has been done in BREEZE [SO00], MOKASSIN [JH98], WIDE [CCPP98], and TRAM [KG99]. An in-depth discussion of approaches on ad-hoc modifications is presented in [Rei00].

Nevertheless, we must point out that projects which consider ad-hoc modifications as well as distributed WF execution at the same time are very rare. In particular, the impact of these important features to each other has not yet been considered sufficiently. The major objective of the approaches cited was not to develop a scalable and flexible WfMS which is efficient with regard to communication costs. This issue has been systematically investigated for the first time in this paper.

We now consider some approaches which address WF modifications as well as distributed WF execution. WIDE allows WF schema modifications and their propagation to running WF instances (if compliant to the new schema) [CCPP98]. In addition, control of WF instances is distributed [CGP+96]. Thereby, the set of the potential actors of an activity determines the WF server which is to control this activity. In MOKASSIN [GJS+99, JH98] and WASA [Wes98, Wes99], distributed WF execution is realized through an underlying CORBA infrastructure. Both approaches do not discuss the criteria used to determine a concrete distribution of the tasks; i.e., the question which WF server has to control a specific activity remains open. Here, modifications may be made at both, the WF schema and the WF instance level under consideration of correctness issues. INCAs [BMR96] realizes WF instance control by means of rules. WF control is distributed, in INCAs, with a given WF instance controlled by that processing station that belongs to the actor of the current activity. The mentioned rules are used to calculate the processing station of the subsequent activity and, thereby, the actor of that activity. With this approach, it is possible to modify the rules, what results in an ad-hoc change of the WF instance behavior. As opposed to the approach presented in this paper, all these approaches do not explicitly address how ad-hoc modifications and distributed WF execution interact.

The approach proposed in [CGR00] enables some kind of flexibility in distributed WfMS as well, especially in the context of virtual enterprises. However, it does not allow to modifiy the structure of in-progress WF instances. Instead, the activities of a WF template represent placeholders for which the concrete implementations are selected at run-time.

In the WF literature, some approaches for distributed WF management are cited where a WF instance is controlled by one and the same WF server over its entire lifetime; e.g., Exotica [AKA+94] and MOBILE [Jab97]. (The latter approach was extended in [SNS99] that way that a sub-process may be controlled by a different WF server, which is determined at run-time.) Although migrations are not performed, different WF instances may be controlled by different WF servers. And, since a central control instance exists for each WF instance in

these approaches, ad-hoc modifications may be performed just as in a central WfMS. Yet there is a drawback with respect to communication costs (cf. [Bau01, BD99a, BD00b]): The distribution model does not allow to select the most favorable WF server for the individual activities. When developing ADEPT, we therefore did not follow such an approach since the additional costs incurred in standard WF execution are higher than the savings generated due to the (relatively seldom performed) ad-hoc modifications.

7 Summary and Outlook

In summary, both distributed WF execution and ad-hoc modification are essential functions for any WfMS to efficiently support the demanding process-oriented application systems deployed by many organizations today (e.g. [LR07]). However, each of these aspects is closely linked with a number of requirements and objectives that are, to some extent, opposing. The reason for this is that the central control instance necessary for ad-hoc modifications typically impacts the efficiency of distributed WF execution. Therefore, we can no longer afford to consider these two aspects separately. For the first time, an investigation of exactly how these functions interact has been presented in this work. And the results have shown that they are, in fact, compatible: We have realized ad-hoc modifications in a distributed WfMS efficiently. Our approach also allows extremely efficient distributed control of previously modified WF instances due to the fact that only a part of the relatively small modification history needs to be transmitted when transferring the modified execution schema. This is vital as migrations are frequently performed operations. To conclude, ADEPT succeeds in seamlessly integrating both distributed WF execution and ad-hoc WF modifications into a single system. The presented concepts have been implemented in a powerful proof-of-concept prototype. It shows that one can really build a WfMS which offers the described functionality within one system. It also shows, however, that such a high-end WfMS is a large software systems, easily reaching the code complexity of high-end database management systems.

There is room for further optimization of the system with respect to the selection of the WF servers which need to be synchronized in an ad-hoc modification: If a modification concerns only a part of the WF schema, the modification could be performed by only those active WF servers controlling that part of the WF instance. This reduces the effort necessary for synchronization and communication. In the extreme case, if only a single branch of a parallel execution has to be modified, only a single server must perform the modification. However, activities belonging to parallel execution branches may be impacted by the modifi-

cation performed (e.g. due to dependencies in the data flow or the temporal conditions set), thus necessitating synchronization of the related WF servers in these cases. Our investigations have shown that the opportunity to deploy such an enhancement is fairly rare so that a significant improvement in the behavior of the system cannot be expected. Nevertheless, this aspect offers a starting-point for future research.

In this paper we have shown how distributed WF control and ad-hoc modifications work in conjunction. Generally, many non-trivial interdependencies exist among the different features of a WfMS (e.g., distribution, temporal constraints, ad-hoc modifications, WF modeling), which must be carefully analyzed and understood. One cannot implement such a system by adding one balcony to another to solve situation-dependent problems. Instead a proper framework is needed which allows to argue about correctness and which covers all possible cases. The ADEPT project reflects this kind of thinking to a large degree. The work on inter-workflow dependencies [Hei01], temporal constraints [DRK00], and component-based application development is on its way.

References

[Aal01a] W.M.P. van der Aalst. Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. *Information Systems Frontiers*, 3(3):297–317, 2001.

[Aal01b] W.M.P. van der Aalst. How to Handle Dynamic Change and Capture Management Information: An Approach Based on Generic Workflow Models. *Int. Journal of Computer Systems, Science, and Engineering*, 16(5):295–318, 2001.

[AB02] W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 2002.

[AH00] W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 25(1):43–69, 2000.

[AH02] W.M.P. van der Aalst and K. van Hee. *Workflow Management*. MIT Press, 2002.

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Gnthr, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Technical Report RJ9913, IBM Almaden Research Center, 1994.

[AMG+95] G. Alonso, C. Mohan, R. Gnthr, D. Agrawal, A. El Abbadi, and M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proc. IFIP Working Conf. on Inf. Syst. for Decentralized Organisations*, Trondheim, 1995.

[Bau01] T. Bauer. *Efficient Realization of Enterprise-wide Workflow Management Systems*. PhD thesis, University of Ulm, Fakultt fr Informatik, 2001. (Tenea-Verlag, in German).

[BD97] T. Bauer and P. Dadam. A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In *Proc. CoopIS'97*, pages 99–108, Kiawah Island, SC, 1997.

[BD99a] T. Bauer and P. Dadam. Distribution Models for Workflow Management Systems. *Informatik Forschung und Entwicklung*, 14(4):203–217, 1999. (in German).

[BD99b] T. Bauer and P. Dadam. Efficient Distributed Control of Enterprise-Wide and Cross-Enterprise Workflows. In *Proc. Workshop Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications, 29. Jahrestagung der GI*, pages 25–32, Paderborn, 1999.

[BD00a] T. Bauer and P. Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proc. CAiSE'00*, pages 94–109, Stockholm, 2000.

[BD00b] T. Bauer and P. Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proc. CAiSE'00*, Stockholm, 2000.

[BFA99] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–10, 1999.

[BMR96] D. Barbará, S. Mehrotra, and M. Rusinkiewicz. INCAs: Managing Dynamic Workflows in Distributed Environments. *J of Database Management*, 7(1):5–15, 1996.

[BRD01] T. Bauer, M. Reichert, and P. Dadam. Efficient Transmission of Process Instance Data in Distributed Workflow Management Systems. *Informatik Forschung und Entwicklung*, 16(2):76–92, 2001. (in German).

[BRD03] T. Bauer, M. Reichert, and P. Dadam. Intra-Subnet Load Balancing for Distributed Workflow Management Systems. *Int. J Coop Inf Sys*, 12(3):295–323, 2003.

[Bu94] C.J. Buler. Policy resolution in workflow management systems. *Digital Technical Journal*, 6(4):26–49, 1994.

[CCPP98] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data & Knowledge Engineering*, 24(3):211–238, 1998.

[CFM99] F. Casati, M. Fugini, and I. Mirbel. An Environment for Designing Exceptions in Workflows. *Information Systems*, 24(3):255–273, 1999.

[CGP+96] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sánchez. WIDE: Workflow Model and Architecture. CTIT Technical Report 96-19, University of Twente, 1996.

[CGR00] A. Cichocki, D. Georgakopoulos, and M. Rusinkiewicz. Workflow Migration Supporting Virtual Enterprises. In *Proc. BIS'00*, pages 20–35, Poznań, 2000.

[Dad96] P. Dadam. *Distributed Databases and Client/Server Systems*. Springer-Verlag, 1996. (in German).

[DGA+97] A. Dogac, E. Gokkoca, S. Arpinar, P. Koksal, I. Cingil, B. Arpinar, N. Tatbul, P. Karagoz, U. Halici, and M. Altinel. Design and Implementation of a Distributed Workflow Management System: METUFlow. In *Proc. NATO Advanced Study Institute on Workflow Management Systems and Interoperability*, pages 61–91, Istanbul, 1997.

[DH01] M. Dumas and A.H.M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proc. Int. Conf. on the Unified Modeling Language*, Toronto, 2001.

[DMP97] B. Dellen, F. Maurer, and G. Pews. Knowledge Based Techniques to Increase the Flexibility of Workflow Management. *Data & Knowledge Engineering*, 23(3):269–296, 1997.

[DR99] P. Dadam and M. Reichert, editors. *Proc. Workshop Enterprise-wide and Cross-Enterprise Workflow Management, Concepts, Systems, Applications*. 29. Jahrestagung der GI, Paderborn, 1999.

[DRK00] P. Dadam, M. Reichert, and K. Kuhn. Clinical Workflows - The Killer Application for Process-oriented Information Systems? In *Proc. 4th Int. Conf. on Business Inf. Syst.*, pages 36–59, Posen, 2000.

[EM97] C.A. Ellis and C. Maltzahn. The Chautauqua Workflow System. In *Proc. 30th Hawaii Int. Conf. on System Sciences*, Maui, 1997.

[End98] H. Enderlin. Realization of a Distributed Workflow Execution Component Based on IBM FlowMark. Master's thesis, University of Ulm, Fakultt fr Informatik, 1998. (in German).

[Fis00] L. Fischer. *Workflow Handbook 2001*. Future Strategies Inc., 2000.

[GJS+99] B. Gronemann, G. Joeris, S. Scheil, M. Steinfort, and H. Wache. Supporting Cross-Organizational Engineering Processes by Distributed Collaborative Workflow Management - The MOKASSIN Approach. In *Proc. 2nd Symposium on Concurrent Multidisciplinary Engineering, 3rd Int. Conf. on Global Engineering Networking*, Bremen, 1999.

[GLO98] V. Guth, K. Lenz, and A. Oberweis. Distributed Workflow Execution Based on Fragmentation of Petri Nets. In *Proc. 15th IFIP World Computer Congress: Telecooperation - The Global Office, Teleworking and Communication Tool*, pages 114–125, 1998.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[GT98] A. Geppert and D. Tombros. Event-Based Distributed Workflow Execution with EVE. In *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing*, pages 427–442, Lake District, 1998.

[Hei01] C. Heinlein. Workflow and Process Synchronization with Interaction Expressions and Graphs. In *Proc. Int. Conf. on Data Engineering*, pages 243–252, Heidelberg, 2001.

[HS98] Y. Han and A. Sheth. On Adaptive Workflow Modeling. In *Proc. 4th Int. Conf. on Information Systems Analysis and Synthesisis*, Orlando, 1998.

[IBM99] IBM. *MQSeries Workflow Administrators Guide*, 1999.

[Jab97] S. Jablonski. Architecture of Workflow Management Systems. *Informatik Forschung und Entwicklung*, 12(2):72–81, 1997. (in German).

[JH98] G. Joeris and O. Herzog. Managing Evolving Workflow Specifications. In *Proc. CoopIS'98*, pages 310–321, New York, 1998.

[KAGM96] M. Kamath, G. Alonso, R. Gnthr, and C. Mohan. Providing High Availability in Very Large Workflow Management Systems. In *Proc. 5th Int. Conf. on Extending Database Technology*, pages 427–442, Avignon, 1996.

[KG99] M. Kradolfer and A. Geppert. Dynamic Workflow Schema Evolution Based on Workflow Type Versioning and Workflow Migration. In *Proc. 4rd IFCIS Int. Conf. on Cooperative Information Systems*, pages 104–114, Edinburgh, 1999.

[LP98] L. Liu and C. Pu. Methodical Restructuring of Complex Workflow Activities. In *Proc. 14th Int. Conf. on Data Engineering*, pages 342–350, Orlando, Florida, 1998.

[LR00] F. Leymann and D. Roller. *Production Workflow - Concepts and Techniques*. Prentice Hall, 2000.

[LR07] R. Lenz and M. Reichert. IT Support for Healthcare Processes - Premises, Challenges, Perspectives. *Data and Knowledge Engineering*, 61:82–111, 2007.

[LS97] Y. Lei and M.P. Singh. A Comparison of Workflow Metamodels. In *Proc. of the ER'97 Workshop on Behavioral Models and Design Transformations*, Los Angeles, CA, 1997.

[MR00] R. Mueller and E. Rahm. Dealing with Logical Failures for Collaborating Workflows. In *Proc. 5th Int. Conf. on Cooperative Information Systems*, pages 210–223, Eilat, 2000.

[MWW+98] P. Muth, D. Wodtke, J. Weißenfels, A. Kotz-Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *JIIS*, 10(2):159–184, 1998.

[RD98] M. Reichert and P. Dadam. ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control. *JIIS*, 10(2):93–129, 1998.

[Rei00] M. Reichert. *Dynamic Changes in Workflow Management Systems*. PhD thesis, University of Ulm, Fakultt fr Informatik, 2000. (in German).

[RRD04a] S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.

[RRD04b] S. Rinderle, M. Reichert, and P. Dadam. Flexible support of team processes by adaptive workflow systems. *Distributed and Parallel Databases*, 16(1):91–116, 2004.

[SGW01] G. Shegalov, M. Gillmann, and G. Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB Journal*, 10(1):91–103, 2001.

[SK97] A. Sheth and K.J. Kochut. Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems. In *Proc. NATO Advanced Study Institute on Workflow Management Systems and Interoperability*, pages 12–21, Istanbul, 1997.

[SM95] D.M. Strong and S.M. Miller. Exceptions and Exception Handling in Computerized Information Processes. *ACM ToIS*, 13(2):206–233, 1995.

[SM96] A. Schill and C. Mittasch. Workflow Management Systems on Top of OSF DCE and OMG CORBA. *Distributed Systems Engineering*, 3(4):250–262, 1996.

[SMO00] W. Sadiq, O. Marjanovic, and M. E. Orlowska. Managing Change and Time in Dynamic Workflow Processes. *Int. Journal Cooperative Information Systems*, 9(1-2):93–116, 2000.

[SNS99] H. Schuster, J. Neeb, and R. Schamburger. A Configuration Management Approach for Large Workflow Management Systems. In *Proc. Int. Conf. on Work Activities Coordination and Collaboration*, San Francisco, 1999.

[SO00] W. Sadiq and M. E. Orlowska. On Capturing Exceptions in Workflow Process Models. In *Proc. BIS'00*, pages 3–19, Poznan, 2000.

[Sta99] Staffware. *Server Administrators Guide*, 1999.

[Wes98] M. Weske. Flexible Modeling and Execution of Workflow Activities. In *Proc. 31st Hawaii Int. Conf. on Sys Sciences*, pages 713–722, Hawaii, 1998.

[Wes99] M. Weske. Workflow Management Through Distributed and Persistent CORBA Workflow Objects. In *Proc. CAiSE'99*, pages 446–450, Heidelberg, 1999.

[WMC98] Workflow Management Coalition. *Workflow Management Application Programming Interface (Interface 2 & 3), Document Number WFMC-TC-1009, Version 2.0*, 1998.

[WMC99] Workflow Management Coalition. *Terminology & Glossary, Document Number WFMC-TC-1011, Document Status - Issue 3.0*, 1999.

[WRR07] B. Weber, S. Rinderle, and M. Reichert. Change patterns and change support features in process-aware information systems. In *Proc. 19th Int'l Conf. on Advanced Information Systems Engineering (CAiSE'07)*, pages 574–588, 2007.

[Zei99] J. Zeitler. Integration of Distribution Concepts into an Adaptive Workflow Management System. Master's thesis, University of Ulm, Fakultt fr Informatik, 1999. (in German).