Supporting Ad-Hoc Changes in Distributed Workflow Management Systems

Manfred Reichert[1] and Thomas Bauer[2]

[1]Informaton Systems Group, University of Twente, The Netherlands
m.u.reichert@cs.utwente.nl
[2]Dept. GR/EPD, DaimlerChrysler AG Group Research, Germany
thomas.tb.bauer@daimlerchrysler.com

Abstract. Flexible support of distributed 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 adapt running process instances are essential requirements. Should the latter one be not 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 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 (WF) instance can be controlled by several WF servers that are as independent from each other as possible. Yet dynamic WF changes, on the other hand, necessitate a (logical) central control instance which knows the current and global state of a WF instance. This paper presents methods which allow ad-hoc modifications (e.g., to insert, delete, or shift steps) to be correctly 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 favorable behavior with respect to communication costs.

1 Introduction

Workflow management systems (WfMS) enable the definition, execution, and monitoring of computerized business processes. Very often, a centralized WfMS shows deficits when it is confronted with high loads or when the business processes to be supported span multiple organizations. As in several other approaches (e.g. [9,15]), in the ADEPT project, we have met this particular demand by realizing a distributed WfMS made up of several workflow (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 [3,5]. Such a distributed WF execution is also needed, for example, for the WF-based support of ubiquitous applications and their integration with backend systems.

R. Meersman and Z. Tari et al. (Eds.): OTM 2007, Part I, LNCS 4803, pp. 150–168, 2007.

Comparable to centralized WfMS (e.g., Staffware), a distributed WfMS must meet high flexibility requirements in order to cover the broad spectrum of processes we can find in large organizations [16,20,14]. In particular, at the WF instance level it must be possible to deviate from the pre-defined WF schema during runtime if required (e.g., by adding, deleting or moving process activities in the flow of control). As reported in literature (e.g., [14,19]), such ad-hoc WF changes become necessary to deal with exceptional or changing situations. Within the ADEPT project we have developed an advanced technology for the support of adaptive workflows. In paticular, ADEPT allows users (or agents) to dynamically modify a running WF instance without causing any run-time error or inconsistency in the sequel (e.g., deadlocks or program crashes due to activity invocations with missing input parameter data) [16,17].

In our previous work we considered distributed execution of partitioned WF schemes and ad-hoc modifications as separate issues. In fact, we have not systematically investigated 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: Ad-hoc WF instance modifications and the correct processing of the WF instance afterwards prescribe a logically central control instance to ensure correctness and consistency [16]. The existence of such a central instance, however, contradicts to the accomplishments achieved by distributed WF execution. The reason for this is that a central component decreases the availability of the WfMS and increases communication efforts 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 [16].

The objective of this paper is to introduce an approach for enabling 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, 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. 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 communication efforts. 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 relevant servers. An essential requirement is, thereby, that the amount of communication may not exceed acceptable limits.

Section 2 gives background information about distributed WfMS, which which is needed for the understanding of this paper. Section 3 describes how ad-hoc modifications are performed in a distributed WfMS, while Section 4 sets out how modified WF instances can be efficiently controlled in such a system. We discuss related work in Section 5 and end with a summary in Section 6.

2 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. 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. This may lead to certain components of the system becoming overloaded. For all these and other reasons, in the distributed variant of 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] [3] (cf. Fig. 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 WF templates (i.e., respective 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.

[1] To achieve a better scalability we allow the same partition of different WF instances to be controlled by multiple WF servers (for details see [6]).

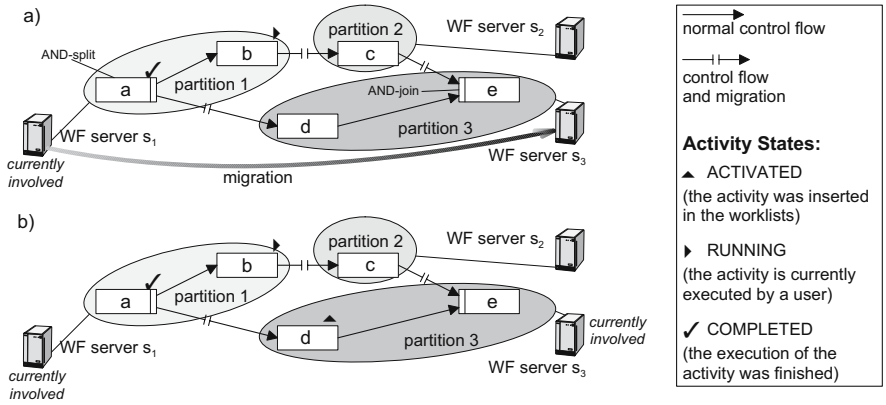

Fig. 1. a) Migration of a WF instance (from s_1 to s_3) and b) the resulting state of the WF instance

The partitioning of WF schemes and distributed WF control have been successfully utilized in other approaches as well (e.g. [9,15]).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. 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 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. The efficiency of the described approach – with respect to WF server load and communication costs – has been proven by means of comprehensive simulations and is outside the scope of this paper (see [4]).

Usually, servers are assigned to the activities of a WF schema already at build-time. However, in some cases this 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* [5]. 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 shall control the related activity instance, to be determined dynamically.

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 (for an example see Fig. 2). The WfMS has to check whether or not the desired modification is allowed on basis of the current structure and state of the concerned WF instance. If the modification is permissible (e.g., if the instance has not progressed too far in its execution), the related change operations will have to be determined and the WF schema belonging to the WF instance will be modified accordingly (incl. adaptations of the WF instance state if required).

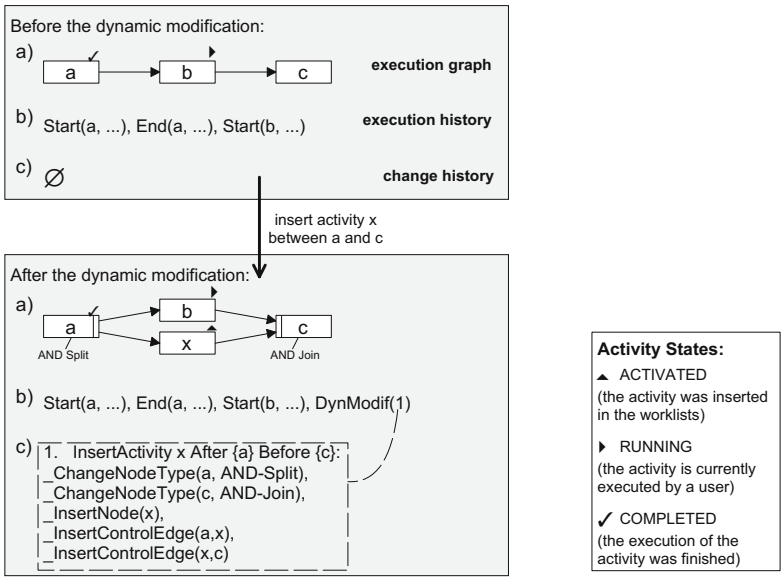

Fig. 2. (Simplified) example of an ad-hoc modification in a centralized WfMS with a) WF execution schema, b) execution history, and c) modification history

To investigate whether an ad-hoc modification is permissible in a distributed WfMS, first, the system needs to know the current global state of the (distributed) WF instance (or at least relevant parts of it). In case of parallel execution branches this state information may be distributed over several WF servers and therefore may have to be retrieved from these WF servers when a change becomes necessary.

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 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 (currently) 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. Note that this becomes necessary to enable them to correctly proceed with the control 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 With 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 in the past and which will not participate again in future is superfluous. They do not need to be synchronized any more and the state information managed by them has already been migrated.

Approach 2: Synchronize 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. Sect. 2) 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. – In the following, we restrict our considerations to Approach 3.

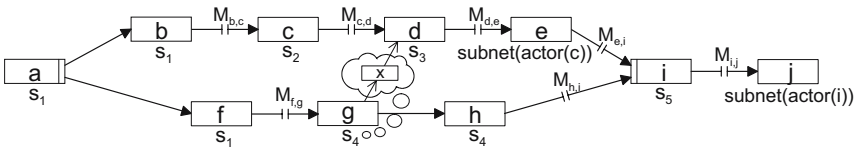

Fig. 3. Insertion of activity x between the activities g and d by the server s_4

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 not a 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.

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 subsections show how the set of active WF servers of a specific WF instance is managed by the *ServerManager*, how this set is determined, and how ad-hoc modifications can be efficiently synchronized.

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.[2] We now 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] 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). 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. 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 the migration from being performed during an ad-hoc modification (cf. Algorithm 3). 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) 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 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*;[3]
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*;

[3] *p()* → *s* means that procedure *p* is called and then executed by server *s*.

```
   // perform the actual migration and release the non-exclusive lock
   MigrateWorkflowInstance(Inst) → TargetServer;
   ReleaseSharedLock(Inst) → ServerManager;
end.
```

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 instance. 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: source server 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};
   end if
   ActiveServers(Inst) = ActiveServers(Inst) ∪ {TargetServer};
   ReleaseShortTermLock(Inst); // release the short-term lock
end.
```

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 being 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 [11] 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.

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) → *s*;
ReleaseExclusiveLock(Inst) → *ServerManager*;
end.

As soon as the lock has been granted, a query is sent to acquire the set of active WF servers of this WF instance.[4] 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. Note 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 or not. 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 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 [16]. If this is the case, the modification is performed at all active WF servers of the WF instance (*PerformDynamic-Modification*). Afterwards, the locks are released and any blocked migrations or modification procedures may then be carried out.

3.3 Illustrative 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

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

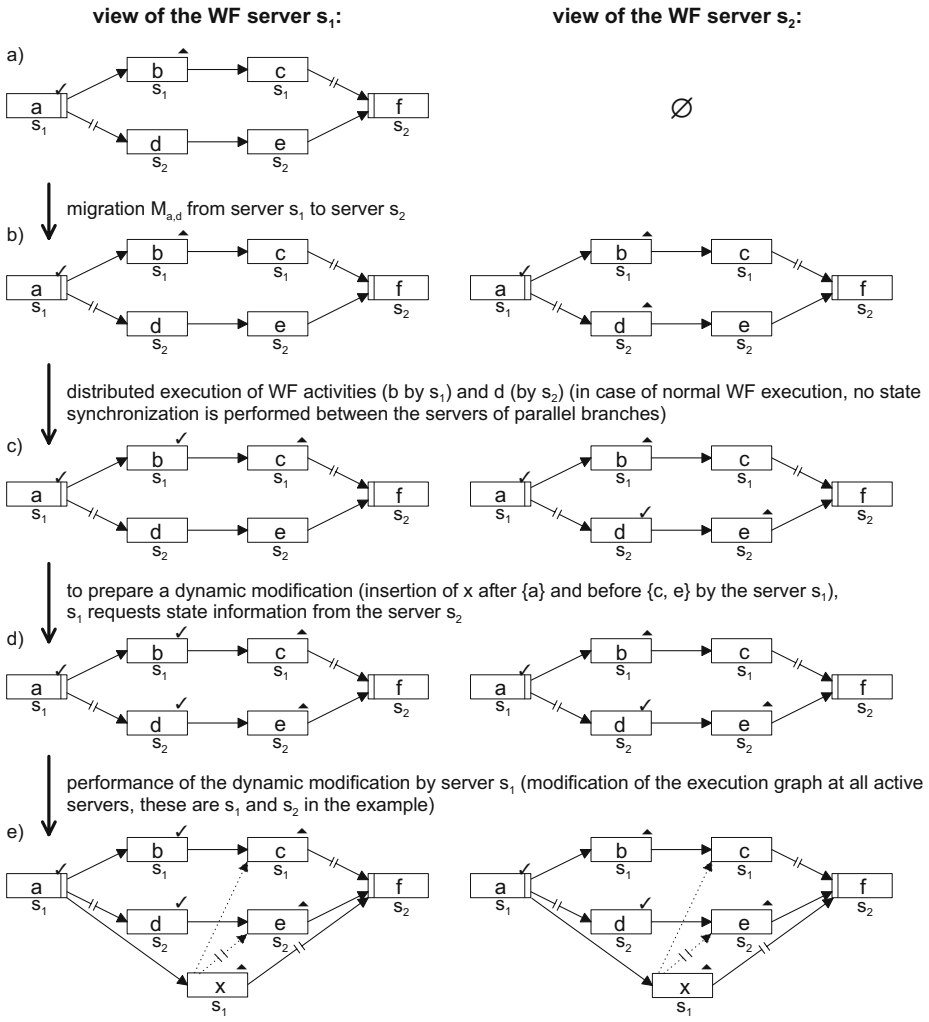

Fig. 4. Effects of migrations and ad-hoc modifications on the (distributed) execution schema of a WF instance (local view of the WF servers)

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)

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. 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 change history for this purpose. In ADEPT the migration target server needs this history anyway [16], so that its transmission does not lead to an additional effort. When the base operations recorded in the change 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 change 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

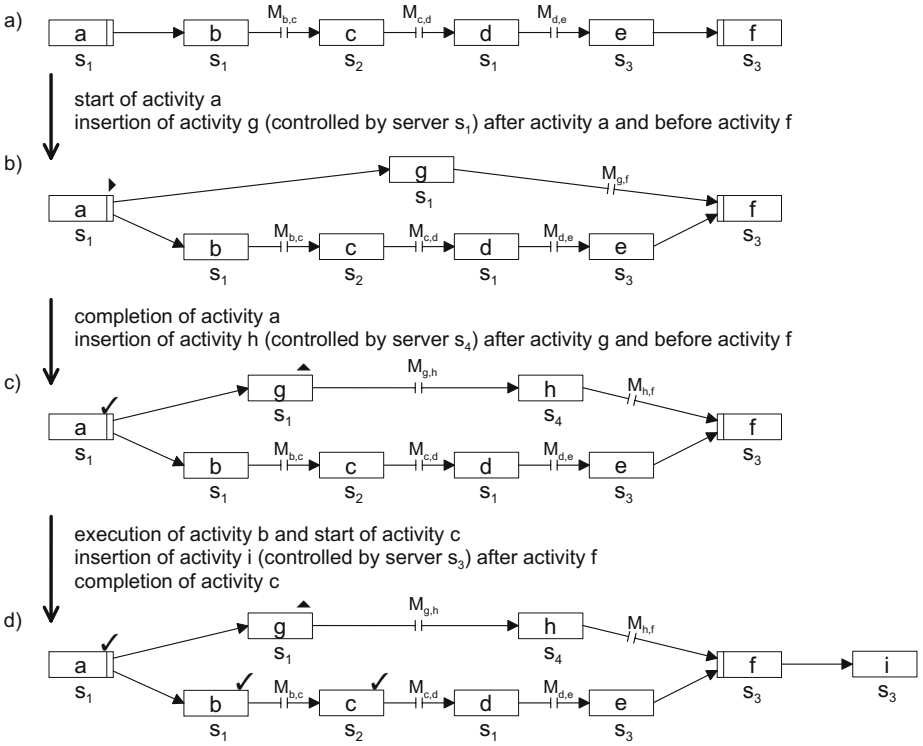

Fig. 5. a-d) WF instance and e) Execution history of WF server s_2 after completion of activity c. – In case of distributed WF control, with each entry the execution history records the WF server responsible for the control of the corresponding activity.

server for the last time. Hence the data related to this part of the change 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".

Transmitting Change History Entries. An obvious solution for avoiding redundant transfer of change 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, 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 change history entry corresponding to modification 3 needs to be transmitted. The transmitted part of the change 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 change 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 change history must be transmitted. Yet with the more advanced approach set out in the next section, we can avoid such redundant data transfer.

Requesting Change History Entries. To avoid redundant data transmissions as described in the previous section, we now sketch a more sophisticated method. With this method, the necessary change 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 change 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. Here, no additional effort is expended for communication, since both, the request for and the transmission of change history entries may be carried out within the same communication cycle.

With the described method, requesting the missing part of a change 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 change 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 change history entries. The source WF server then transmits all change history entries made after this point. Due to lack of space we omit further details.

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 Related Work

There are only few approaches which address both WF modification issues and distributed WF control [8,9,13,21,2]. WIDE [9] allows WF schema modifications

and their propagation to running WF instances (if compliant to the new schema). In addition, control of WF instances is distributed [9]. Thereby, the set of the potential actors of an activity determines the WF server which is to control this activity. In MOKASSIN [13] and WASA [20,21], 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 [2] 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 [10] 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 [1] and MOBILE [12]. (The latter approach was extended in [18] 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: 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.

6 Summary

Both distributed WF execution and ad-hoc modification are essential functions of any WfMS. Each of these aspects is closely linked with a number of requirements and objectives that are, to some extent, opposing. Reason for this is that the central control instance necessary for ad-hoc modifications typically impacts the efficiency of distributed WF execution. Therefore, we cannot afford to consider these two aspects separately. An investigation of exactly how these functions interact has been presented. And the results show that they are, in fact, compatible: We have realized ad-hoc modifications in a distributed WfMS.

Our approach also allows efficient distributed control of previously modified WF instances since only a part of the relatively small change history needs to be transmitted when transferring the modified execution schema. This is vital as migrations are frequent. 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, which constitutes the distributed variant of the ADEPT system (cf. Fig. 6). It shows that one can really build a WfMS which offers the described functionality within one system (for details see [7]). 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.

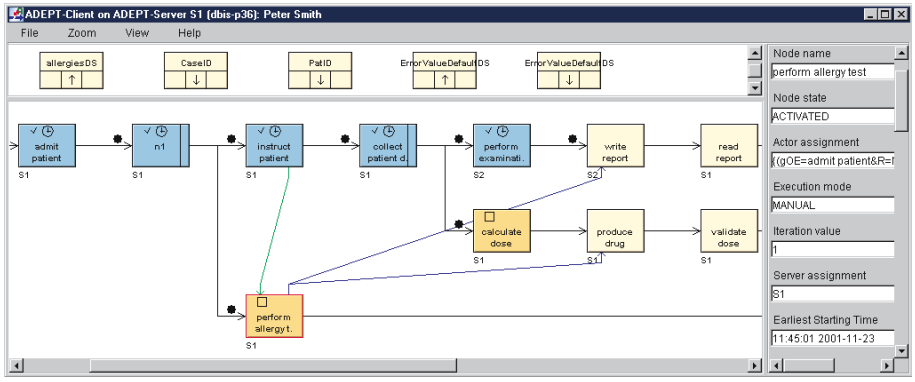

Fig. 6. ADEPT monitoring component showing a distributed workflow controlled by servers S1 and S2 after its runtime modification

References

1. Alonso, G., Kamath, M., Agrawal, D., El Abbadi, A., Günthör, R., Mohan, C.: Failure Handling in Large Scale Workflow Management Systems. Technical Report RJ9913, IBM Almaden Research Center (1994)
2. Barbará, D., Mehrotra, S., Rusinkiewicz, M.: INCAs: Managing Dynamic Workflows in Distributed Environments. J. of Database Management 7(1), 5–15 (1996)
3. Bauer, T., Dadam, P.: A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In: Proc. CoopIS 1997, Kiawah Island, SC, pp. 99–108 (1997)
4. Bauer, T., Dadam, P.: Distribution Models for Workflow Management Systems. Informatik Forschung und Entwicklung 14(4), 203–217 (1999) (in German)
5. Bauer, T., Dadam, P.: Efficient Distributed Workflow Management Based on Variable Server Assignments. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 94–109. Springer, Heidelberg (2000)
6. Bauer, T., Reichert, M., Dadam, P.: Intra-Subnet Load Balancing for Distributed Workflow Management Systems. Int. J. Coop. Inf. Sys. 12(3), 295–323 (2003)

7. Bauer, Th., Reichert, M.: An Approach for Supporting Ad-hoc Process Changes in Distributed Workflow Management Systems. Technical report, University of Twente, CTIT (September 2007)
8. Cao, J., Yang, J., Chan, W., Xu, C.: Exception handling in distributed workflow systems using mobile agents. In: Proc. ICEBE 2005, pp. 48–55 (2005)
9. Casati, F., Grefen, P., Pernici, B., Pozzi, G., Sánchez, G.: WIDE: Workflow Model and Architecture. CTIT Technical Report 96-19, University of Twente (1996)
10. Cichocki, A., Georgakopoulos, D., Rusinkiewicz, M.: Workflow Migration Supporting Virtual Enterprises. In: Proc. BIS 2000, Poznań, pp. 20–35 (2000)
11. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco (1993)
12. Jablonski, S.: Architecture of Workflow Management Systems. Informatik Forschung und Entwicklung 12(2), 72–81 (1997) (in German)
13. Joeris, G., Herzog, O.: Managing Evolving Workflow Specifications. In: Proc. CoopIS 1998, New York, pp. 310–321 (1998)
14. Lenz, R., Reichert, M.: IT Support for Healthcare Processes - Premises, Challenges, Perspectives. DKE 61, 82–111 (2007)
15. Muth, P., Wodtke, D., Weißenfels, J., Kotz-Dittrich, A., Weikum, G.: From Centralized Workflow Specification to Distributed Workflow Execution. JIIS 10(2), 159–184 (1998)
16. Reichert, M., Dadam, P.: ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control. JIIS 10(2), 93–129 (1998)
17. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distributed and Parallel Databases 16(1), 91–116 (2004)
18. Schuster, H., Neeb, J., Schamburger, R.: A Configuration Management Approach for Large Workflow Management Systems. In: Proc. Int. Conf. on Work Activities Coordination and Collaboration, San Francisco (1999)
19. Weber, B., Rinderle, S., Reichert, M.: Change patterns and change support features in process-aware information systems. In: CAiSE 2007. Proc. 19th Int'l Conf. on Advanced Information Systems Engineering, pp. 574–588 (2007)
20. Weske, M.: Flexible Modeling and Execution of Workflow Activities. In: Proc. 31st Hawaii Int. Conf. on Sys Sciences, Hawaii, pp. 713–722 (1998)
21. Weske, M.: Workflow Management Through Distributed and Persistent CORBA Workflow Objects. In: Jarke, M., Oberweis, A. (eds.) CAiSE 1999. LNCS, vol. 1626, pp. 446–450. Springer, Heidelberg (1999)