Heuer, A.; Leymann, F.; Priebe, D. (Hrsg.): Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, Oldenburg, März 2001, Springer Verlag, 2001, S. 47-66

Adaptives und verteiltes Workflow-Management

Thomas Bauer, Manfred Reichert, Peter Dadam

Universität Ulm, Abteilung Datenbanken und Informationssysteme
{bauer, reichert, dadam}@informatik.uni-ulm.de, http://www.informatik.uni-ulm.de/dbis

Zusammenfassung Die Unterstützung unternehmensweiter und -übergreifender Geschäftsprozesse stellt für ein Workflow-Management-System (WfMS) eine besondere Herausforderung dar. So sind Skalierbarkeit bei hoher Last und die Möglichkeit, zur Ausführungszeit eines Workflows (WF) dynamisch vom vormodellierten Ablauf abweichen zu können, unbedingt erforderlich, damit ein WfMS für ein breites Spektrum von Anwendungen eingesetzt werden kann. Allerdings wurden diese beiden Aspekte in der WF-Literatur bisher weitestgehend getrennt betrachtet. Dies ist äußerst problematisch, da mit ihnen entgegengesetzte Anforderungen verbunden sind. So wird zur Erzielung einer guten Skalierbarkeit angestrebt, dass eine WF-Instanz von mehreren WF-Servern – möglichst unabhängig voneinander – kontrolliert werden kann, wohingegen für dynamische WF-Änderungen eine (logisch) zentrale Kontrollinstanz benötigt wird, die den aktuellen und globalen Zustand der WF-Instanz kennt. In diesem Beitrag werden Verfahren vorgestellt, die es erlauben, dynamische Änderungen in einem verteilten WfMS durchzuführen. Besonders bemerkenswert ist dabei, dass es gelungen ist, die volle aus dem zentralen Fall bekannte Funktionalität zu realisieren, und trotzdem ein bezüglich der Kommunikationskosten äußerst günstiges Verhalten zu erreichen.

1 Einleitung

WfMS [JBS97, LR00, Obe96] ermöglichen die rechnerunterstützte Ausführung von Geschäftsprozessen in einer verteilten Systemumgebung. Der entscheidende Vorteil solcher Systeme ist, dass sie helfen, große vorgangsorientierte Anwendungssysteme überschaubarer zu gestalten. Dazu wird der applikationsspezifische Code der verwendeten Anwendungen von der Ablauflogik des Geschäftsprozesses getrennt. Anstelle eines großen monolithischen Programmpakets erhält man nun einzelne Aktivitäten, welche die Anwendungsprogramme repräsentieren. Ihre Ablauflogik wird in einer separaten Kontrollflussdefinition festgelegt, welche die Ausführungsreihenfolge (Sequenz, Verzweigung, Parallelität, Schleifen) der einzelnen Aktivitäten bestimmt. Das WfMS sorgt zur Ausführungszeit dafür, dass nur solche Aktivitäten einer WF-Instanz bearbeitet werden können, die der Ablauflogik zufolge zur Ausführung anstehen. Diese werden in die Arbeitslisten autorisierter Bearbeiter eingefügt. Welche Benutzer zur Bearbeitung einer bestimmten Aktivität autorisiert sind, wird meist durch eine Rolle festgelegt, die auch den entsprechenden Bearbeitern zugeordnet ist.

Die Funktionalität heutiger WfMS ist allerdings für viele der in der Praxis vorkommenden prozessorientierten Anwendungen nicht ausreichend. Ein Mangel ist insbesondere

die unzureichende Skalierbarkeit bei hoher Last. Der Forderung nach Skalierbarkeit kommen wir in ADEPT[1] dadurch nach, dass ein WfMS aus mehreren WF-Servern besteht. Um ein günstiges Kommunikationsverhalten zu erzielen, kann eine WF-Instanz abschnittsweise von verschiedenen WF-Servern kontrolliert werden [BD97, BD00]. Eine ebenfalls erkannte Schwachstelle existierender WfMS ist ihre unzureichende Adaptivität. In ADEPT ist es möglich, eine laufende WF-Instanz bei Bedarf (z.B. in Ausnahmesituationen) dynamisch zu verändern, so dass von dem vorgesehenen Ablauf abgewichen wird. Im Gegensatz zu vielen anderen Ansätzen wird dabei die Konsistenz der WF-Instanz auch nach der Änderung garantiert, d.h. Laufzeitfehler (z.B. Verklemmungen in Folge zyklischer Reihenfolgebeziehungen) sind ausgeschlossen [RD98, Rei00]. In den bisherigen Veröffentlichungen zu ADEPT haben wir, ebenso wie die anderen uns bekannten Gruppen, verteilte WF-Ausführung und Adaptivität nur getrennt betrachtet. Das Zusammenspiel dieser beiden für ein WfMS essentiellen Aspekte wurde dabei nicht systematisch untersucht. Dies ist problematisch, weil mit den beiden Aspekten entgegengesetzte Anforderungen verbunden sind: Zur Durchführung einer dynamischen Änderung wird eine zentrale Kontrollinstanz benötigt [RD98]. Die Existenz einer solchen konterkariert aber die durch verteilte WF-Ausführung erzielten Errungenschaften, da eine zentrale Komponente die Verfügbarkeit des WfMS verschlechtert und den Kommunikationsaufwand erhöht. Ein Grund dafür ist, dass eine solche Komponente über jede Änderung des Zustands jeder WF-Instanz informiert werden muss. Der Zustand der zu modifizierenden WF-Instanz wird nämlich benötigt, um entscheiden zu können, ob eine geplante Änderung überhaupt durchführbar ist [RD98].

Das Ziel des vorliegenden Beitrags ist es, dynamische Änderungen in einem verteilten WfMS zu ermöglichen. Dabei soll die Adaptivität gegenüber der zentralen WF-Ausführung nicht eingeschränkt sein, d.h. jede für den zentralen Fall erlaubte dynamische Änderung muss im verteilten Fall weiterhin zulässig sein. Die Unterstützung solcher dynamischer Abweichungen darf die Effizienz der verteilten WF-Steuerung aber keinesfalls beeinträchtigen. Insbesondere soll bei der „normalen“ WF-Ausführung kein großer zusätzlicher Kommunikationsaufwand notwendig werden. Außerdem sollen in dem angestrebten System (verteilte) dynamische Änderungen auf möglichst effiziente Art und Weise durchgeführt werden können.

Zur Behandlung dieser Anforderungen ist zu untersuchen (siehe auch [RBD99]), mit welchen Servern des WfMS eine dynamische Änderung synchronisiert werden muss. Vermutlich müssen dabei zumindest die aktuell an der WF-Instanz beteiligten Server einbezogen werden, da sie die resultierende Struktur und den Zustand der WF-Instanz (den sog. Ausführungsgraphen) benötigen, um diese korrekt steuern zu können. Deshalb wird ein Verfahren benötigt, mit dem die Menge dieser Server ohne großen Aufwand ermittelt werden kann. Außerdem ist zu klären, wie diesen und anderen Servern der durch die Änderung resultierende Ausführungsgraph der WF-Instanz übermittelt werden kann. Dabei ist essentiell, dass kein inakzeptabel großer Kommunikationsaufwand verursacht wird.

[1] ADEPT steht für Application Development Based on Encapsulated Pre-Modeled Process Templates.

Im nachfolgenden Abschnitt wird auf Grundlagen der verteilten WF-Ausführung und der dynamischen Änderungen in ADEPT eingegangen, die für das weitere Verständnis notwendig sind. Der Abschnitt 3 beschäftigt sich mit der Durchführung von dynamischen Änderungen in einem verteilten WfMS. In Abschnitt 4 wird dann erläutert, wie auch zuvor veränderte WF-Instanzen effizient gesteuert werden können. Eine Einordnung der beschriebenen Verfahren in die WF-Literatur findet sich in Abschnitt 5. Der Beitrag schließt mit einer Zusammenfassung und einem Ausblick auf zukünftige Arbeiten.

2 Grundlagen

Im ADEPT-Projekt [DKR+95, DRK00, RD98] betrachten wir Anforderungen, die sich bei der Unterstützung unternehmensweiter und -übergreifender WF-basierter Anwendungen ergeben. Im vorliegenden Abschnitt fassen wir die hieraus hervorgegangenen Konzepte zur verteilten WF-Steuerung und zu dynamischen WF- Änderungen kurz zusammen.

2.1 Verteilte Workflow-Ausführung in ADEPT

Aufgrund der großen Anzahl von Benutzern und gleichzeitig aktiven WF-Instanzen resultiert in unternehmensweiten Anwendungen eine sehr hohe Last.[2] Diese kann dazu führen, dass Komponenten des WfMS überlastet werden. Deshalb wird in ADEPT$_{distribution}$, der verteilten Variante von ADEPT, eine WF-Instanz nicht nur von einem einzigen WF-Server kontrolliert, sondern sie wird ggf. partitioniert und abschnittsweise von verschiedenen Servern gesteuert [BD97] (vgl. Abb. 1). Wird bei der Ausführung von Instanzen dieses WF-Typs das Ende einer Partition erreicht, so wird die Kontrolle über diesen WF an den nächsten WF-Server weitergereicht (*Migration*). Damit dies möglich ist, muss eine Beschreibung des Zustands der WF-Instanz zu diesem Server transportiert werden.[3] Um unnötigen Kommunikationsaufwand zwischen WF-Servern zu vermeiden, erfolgt in ADEPT die Steuerung von parallelen Zweigen unabhängig voneinander (wenn zur Zeit keine dynamische Änderung durchgeführt wird). Im Beispiel aus Abb. 1 weiß der WF-Server s_3, der gerade die Aktivität e kontrolliert, nicht, wie weit die Bearbeitung im oberen Zweig der Parallelität fortgeschritten ist. Dies hat den Vorteil, dass keine Synchronisation zwischen WF-Servern notwendig ist, die Aktivitäten paralleler Zweige steuern.

Die Partitionierung von WF-Graphen und die verteilte WF-Steuerung werden auch in anderen Ansätzen verwendet (z.B. [CGP+96, MWW+98]). Wir verfolgen in ADEPT zusätzlich das Ziel, die Kommunikationskosten zu minimieren. Konkrete Erfahrungen mit existierenden WfMS haben gezeigt, dass zwischen dem WF-Server und seinen

[2] In [KAGM96, SK97] werden Anwendungen beschrieben, bei denen die Zahl der Benutzer des WfMS bis auf einige zehntausend anwachsen kann oder mehrere zehntausend WF-Instanzen gleichzeitig im System sein können.

[3] Die WF-Vorlagen werden in ADEPT repliziert bei allen (relevanten) WF-Servern gespeichert.

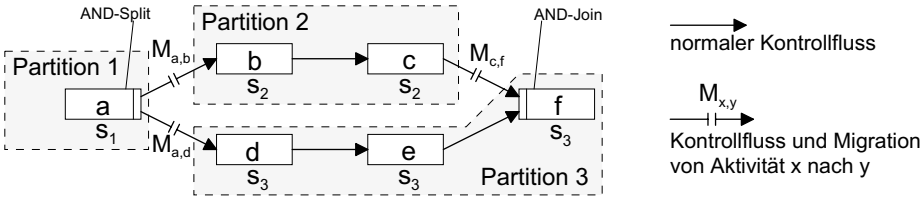


Abbildung 1. Partitionierung eines WF-Graphen und verteilte WF-Ausführung.

Clients sehr viel kommuniziert wird und zum Teil auch große Datenmengen ausgetauscht werden müssen. Dies kann dazu führen, dass das Kommunikationssystem überlastet wird. Deshalb werden in ADEPT die WF-Server der Aktivitäten so festgelegt, dass der Gesamtkommunikationsaufwand minimiert wird. Dazu wird der WF-Server für eine Aktivität in der Regel so gewählt, dass er im Teilnetz der Mehrzahl ihrer potentiellen Bearbeiter liegt. Dadurch wird in vielen Fällen eine teilnetzübergreifende Kommunikation zwischen dem WF-Server und seinen Clients vermieden. Außerdem werden die Antwortzeiten verbessert und die Verfügbarkeit erhöht, da bei der Ausführung von Aktivitäten kein Gateway oder WAN (Wide Area Network) zwischengeschaltet ist.

Bei diesem Ansatz wird also bereits zur Modellierungszeit eine (statische) Zuordnung von WF-Servern zu den Aktivitäten vorgenommen. Es gibt aber auch Fälle, in denen diese Vorgehensweise nicht ausreichend ist, um gute Resultate zu erzielen. Dies ist der Fall, wenn bei der WF-Definition *abhängige Bearbeiterzuordnungen* (z.B. "selber Bearbeiter wie Aktivität n") verwendet werden. Hier hängen die potentiellen Bearbeiter einer Aktivität vom Bearbeiter einer Vorgängeraktivität ab. Da sich die Menge der potentiellen Bearbeiter einer solchen Aktivität erst im Verlauf der WF-Ausführung ergibt, ist es vorteilhaft, ihren WF-Server ebenfalls erst zur Ausführungszeit festzulegen. Der Server kann dann in einem für die entsprechenden Bearbeiter günstigen Teilnetz gewählt werden. Zu diesem Zweck wurde für ADEPT$_{distribution}$ das Konzept der *variablen Serverzuordnungen* [BD99a, BD00] entwickelt. Hierbei handelt es sich um Ausdrücke (z.B. "Server im Teilnetz des Bearbeiters der Aktivität n"), die zur Ausführungszeit der WF-Instanz ausgewertet werden. Dadurch wird derjenige WF-Server ermittelt, der die zugehörige Aktivitäteninstanz kontrollieren soll.

2.2 Dynamische Änderungen

Um auf Ausnahmesituationen flexibel reagieren zu können, muss der Ausführungsgraph einer WF-Instanz zur Ausführungszeit modifiziert werden können. Das ADEPT$_{flex}$-Kalkül bietet z.B. die Möglichkeit, Aktivitäten dynamisch einzufügen oder zu löschen. Dabei sind auch sehr komplexe Operationen realisierbar, wie das Einfügen einer Aktivität, die erst nach der Beendigung einer beliebigen Menge von Aktivitäten ausführbar sein soll und vor dem Starten einer anderen Menge von Aktivitäten beendet sein muss. Auf die Verfahren zur Durchführung solcher dynamischen Änderungen wird hier nicht näher eingegangen, da dies für das weitere Verständnis dieses Beitrags nicht notwendig ist (für Details siehe [RD98, Rei00]).

Das WfMS bestimmt aus der Spezifikation einer Änderung automatisch eine Menge von *Basisoperationen* (z.B. Aktivität einfügen, Kante einfügen), die auf den Ausführungsgraphen der entsprechenden WF-Instanz angewandt werden. Wie in Abb. 2c dargestellt, werden diese Basisoperationen zusammen mit der Spezifikation der Änderung in einer *Änderungshistorie* vermerkt. Diese wird benötigt, um im Falle des partiellen Zurücksetzens einer WF-Instanz ggf. auch temporäre Änderungsoperationen rückgängig machen zu können (vgl. [DRK00]). Außerdem wird die dynamische Änderung in der *Ablaufhistorie* der WF-Instanz vermerkt (Eintrag *DynModif*(1) in Abb. 2b für die Änderung 1), in welcher ansonsten für die WF-Ausführung benötigte Informationen, wie die Start- und Endezeitpunkte und die Bearbeiter der Aktivitäten, abgelegt werden.

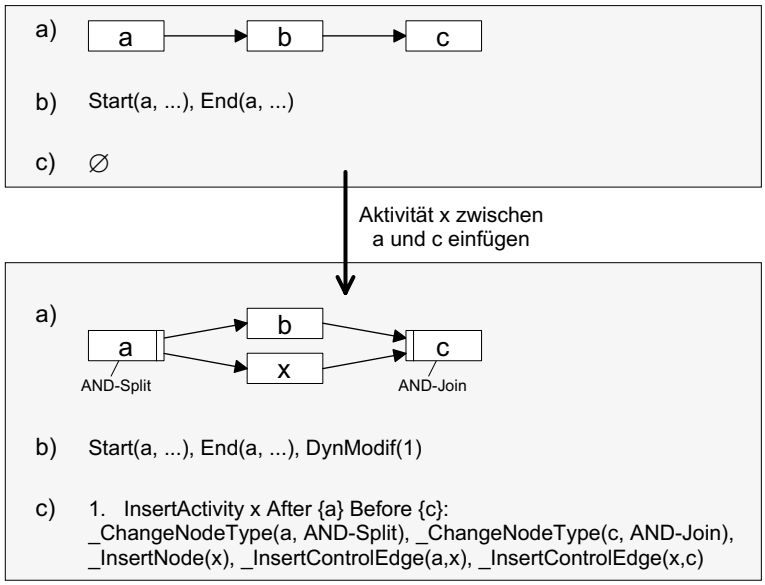


Abbildung 2. Beispiel für eine dynamische Änderung mit a) Ausführungsgraph, b) Ablaufhistorie und c) Änderungshistorie (vereinfacht dargestellt).

Um eine robuste WF-Ausführung zu ermöglichen, muss die *Konsistenz* eines WF-Ausführungsgraphen jederzeit garantiert werden können. Prinzipiell können aber durch eine dynamische Modifikation Inkonsistenzen entstehen. So können durch das Löschen einer Aktivität evtl. Eingabedaten nachfolgender Aktivitäten nicht mehr sicher versorgt sein, oder nach dem Einfügen einer Kante können Verklemmungen durch zyklische Kontrollflussreihenfolgen entstehen. Solche Fälle sind in ADEPT$_{flex}$ ausgeschlossen, weil das WfMS vor der Durchführung einer Änderung stets prüft, ob der resultierende Ausführungsgraph wieder eine fehlerfreie WF-Ausführung garantiert. Zu diesem Zweck wird analysiert, ob die Änderung aufgrund des aktuellen Zustands und der Struktur der WF-Instanz zulässig ist, d.h., ob die für die entsprechenden Basisoperatio-

nen (formal) definierten Vor- und Nachbedingungen erfüllt sind. Nur wenn dies gegeben ist, wird die Struktur und der Zustand des Ausführungsgraphen entsprechend verändert.

3 Dynamische Änderungen in verteilten WfMS

Prinzipiell laufen dynamische Änderungen in einem verteilten WfMS ebenso ab, wie in einem zentralen System: Anhand der Struktur und des Zustands der WF-Instanz wird überprüft, ob die gewünschte Änderung zulässig ist oder nicht. Falls dem so ist, werden die zugehörigen Basisoperationen ermittelt und der Ausführungsgraph der WF-Instanz wird entsprechend modifiziert. Für die Überprüfung der Zulässigkeit einer Änderung wird der globale, aktuelle Zustand der WF-Instanz benötigt. Um diesen Zustand zu ermitteln, muss im Allgemeinen Zustandsinformation von anderen Servern eingeholt werden (wie die Übertragung eines Zustandes effizient möglich ist, wird in [BRD00] beschrieben). Im vorliegenden Abschnitt wird ein Verfahren vorgestellt, mit dem die Menge der WF-Server bestimmt werden kann, von denen entsprechende Zustandsinformation benötigt wird. Im Gegensatz zum zentralen Fall reicht es in einem verteilten WfMS in der Regel nicht aus, den Ausführungsgraphen der WF-Instanz nur auf demjenigen Server zu modifizieren, der die Änderung steuert. Deshalb wird in diesem Abschnitt geklärt, bei welchen Servern eine Änderung eingebracht werden soll.

3.1 Synchronisation von Workflow-Servern bei dynamischen Änderungen

Eine dynamische Änderung kann von einem beliebigen Server, der die betroffene WF-Instanz gerade kontrolliert, initiiert werden. Dieser WF-Server kann die Änderung im Allgemeinen aber nicht alleine durchführen, sondern er muss für den Fall, dass aktuell mehrere Server an der WF-Kontrolle beteiligt sind, ggf. Zustandsinformation von diesen einholen. Des weiteren muss er veranlassen, dass die Änderung auch in die von anderen Servern verwalteten Ausführungsgraphen dieser WF-Instanz eingebracht wird. Als naive Lösung könnten alle Server des WfMS in eine Änderung mit einbezogen werden, indem entsprechende Aufrufe per Broadcast verbreitet werden. Dieser Ansatz scheidet aber aus, weil er in den meisten Fällen zu einem unnötig hohen Aufwand führen würde, und außerdem eine dynamische Änderung nur dann durchgeführt werden könnte, wenn alle Serverrechner des WfMS erreichbar sind. Deshalb werden im Folgenden alternative Vorgehensweisen untersucht.

Ansatz 1: Einbeziehung aller Server der betroffenen Workflow-Instanz
Bei diesem Ansatz werden nur diejenigen WF-Server bei der Änderung berücksichtigt, die bisher an der WF-Steuerung beteiligt waren bzw. es aktuell sind, oder die zukünftig in die Steuerung der WF-Instanz involviert sein werden. Dadurch wird der Kommunikationsaufwand gegenüber der oben erwähnten naiven Lösung zwar merklich reduziert, er ist aber immer noch unnötig groß. Diejenigen Server, welche die WF-Instanz ausschließlich in der Vergangenheit kontrolliert haben, müssen nämlich nicht in die Änderung einbezogen werden. Mit diesen Servern ist keine Synchronisation notwendig und die von ihnen verwaltete Zustandsinformation wurde schon durch Migrationen weitergegeben.

Ansatz 2: Einbeziehung der aktuellen und zukünftigen Server der Workflow-Instanz
Um eine WF-Instanz steuern zu können, müssen dem entsprechenden WF-Server alle bisher erfolgten dynamischen Änderungen bekannt sein. Deshalb ist eine Änderung für alle WF-Server relevant, welche die WF-Instanz aktuell kontrollieren oder zukünftig kontrollieren werden. Darum scheint es naheliegend zu sein, diese in die Änderungsoperation einzubeziehen. Sollen aber die zukünftigen Server berücksichtigt werden, so entsteht im Kontext von variablen Serverzuordnungen (vgl. Abschnitt 2.1) ein Problem. Es kann nämlich im Allgemeinen nicht ermittelt werden, welche Server die WF-Instanz zukünftig steuern werden, da die zur Auswertung der Serverzuordnungsausdrücke notwendigen Laufzeitdaten der WF-Instanz evtl. noch nicht existieren. So kann in Abb. 3 bei der Ausführung der Aktivität g der Server der Aktivität j nicht ermittelt werden, da der Bearbeiter der Aktivität i noch nicht feststeht. Eine Synchronisation mit zukünftigen Servern der WF-Instanz ist also nicht möglich. Diese Server müssen auch noch nicht über die Änderung informiert werden, weil sie die WF-Instanz noch nicht kontrollieren.

Ansatz 3: Einbeziehung nur der aktuellen Server der Workflow-Instanz
Eine Synchronisation mit allen aktuell an der WF-Instanz beteiligten Servern ist demnach die einzig akzeptable Lösung. Es ist aber keineswegs trivial, diese Server zu ermitteln, weil der Ausführungszustand von parallel ausgeführten Aktivitäten bei verteilter WF-Steuerung nicht bekannt sein muss. So weiß z.B. in Abb. 3 der Server s_4, der die Aktivität g kontrolliert, nicht, ob die Migration $M_{c,d}$ schon ausgeführt worden ist, und damit, ob der parallele Zweig vom Server s_2 oder s_3 kontrolliert wird. Außerdem ist es nicht ohne weiteres möglich, den für einen parallelen Zweig zuständigen Server zu ermitteln, wenn variable Serverzuordnungen verwendet werden. So referenziert in Abb. 3 die Serverzuordnung der Aktivität e den Bearbeiter der Aktivität c. Dieser ist aber dem Server s_4 nicht bekannt.

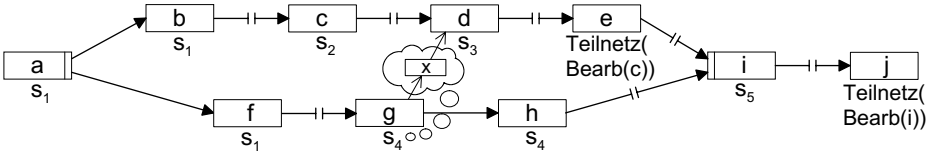


Abbildung 3. Einfügen der Aktivität x durch den Server s_4 zwischen den Aktivitäten g und d.

3.2 Bestimmung der aktuellen Server einer Workflow-Instanz

Wie soeben erläutert, ist es einem WF-Server nicht möglich, die anderen aktuell an einer WF-Instanz beteiligten Server aus der lokal vorhandenen Zustandsinformation zu ermitteln. Diese durch einen Broadcast-Aufruf zu „suchen“, verbietet sich, da dann dieselben Nachteile wie bei der eingangs des vorangehenden Abschnitts skizzierten naiven Lösung entstehen. Deshalb muss ein Verfahren entwickelt werden, bei dem die

Menge der an der Ausführung einer WF-Instanz aktuell beteiligten Server explizit verwaltet wird. Diese Verwaltung sollte aber nicht durch einen festen (und damit zentralen) Server erfolgen, da dieser die Verfügbarkeit des gesamten WfMS beeinträchtigen und einen potentiellen Flaschenhals darstellen würde. Deshalb wird diese Menge *ActiveServers* in ADEPT von einem WF-instanzspezifischen *ServerManager* verwaltet. Als *ServerManager* wird normalerweise jeweils derjenige Server verwendet, auf dem die entsprechende WF-Instanz gestartet wurde.[4] Dieser kann von jedem an der WF-Instanz beteiligten Server mit Hilfe der (lokal vorhandenen) Ablaufhistorie ermittelt werden und variiert für verschiedene WF-Instanzen (auch desselben Typs). Deshalb stellt er keinen Flaschenhals dar. Im nun folgenden Abschnitt wird beschrieben, wie die Menge der aktuell für eine WF-Instanz aktiven Server vom *ServerManager* verwaltet wird. In Abschnitt 3.2.2 wird dann erläutert, wie diese Menge bei dynamischen Änderungen ermittelt und verwendet wird.

3.2.1 Verwaltung der aktuell an einer Workflow-Instanz beteiligten Server

Um die Menge der an der Ausführung einer WF-Instanz aktuell beteiligten Server verwalten zu können, wird ein Verfahren benötigt, welches die aus einer Migration resultierende Veränderung der Menge *ActiveServers* an den jeweiligen *ServerManager* meldet (siehe Algorithmus 1). Dieser *ServerManager* verwendet wiederum ein Verfahren, mit dem diese Menge manipuliert wird (Algorithmus 2). Bei der Realisierung dieser beiden Verfahren muss beachtet werden, dass nicht mehrere Migrationen derselben WF-Instanz beliebig überlappend durchgeführt werden dürfen, da ansonsten Inkonsistenzen bei der Veränderung der Menge der aktuellen Server entstehen können. Außerdem darf sich die Menge *ActiveServers* während der Durchführung einer dynamischen Modifikation nicht durch Migrationen verändern, um in eine dynamische Änderung die richtigen Server einbeziehen zu können. Dies wird verhindert, indem von den beiden genannten Verfahren und von dem im Abschnitt 3.2.2 vorgestellten Verfahren zur Durchführung von dynamischen Änderungen verschiedene Sperren verwendet werden, deren Zusammenspiel im Folgenden erläutert wird.

Der Algorithmus 1 zeigt den Ablauf einer Migration. Zuerst fordert der Migrationsquellserver beim *ServerManager* eine nicht exklusive Sperre an.[5] Dann wird eine exklusive Kurzzeitsperre angefordert,[6] durch die sichergestellt wird, dass die Veränderung der Servermenge für eine gegebene WF-Instanz nicht gleichzeitig für mehrere Migrationen paralleler Zweige vorgenommen wird. Die bei einer Migration resultierende Änderung der Servermenge wird vom Quellserver an den *ServerManager* gemeldet. Dabei wird angegeben, ob der Quellserver der Migration weiterhin an der entsprechenden WF-

[4] Es kann Szenarien geben, bei denen sich durch diese Strategie stets derselbe Server ergibt, weil alle WF-Instanzen auf demselben WF-Server instanziiert werden (z.B. dem Server, der in einer Bank für die Terminals der Schalter zuständig ist). In einem solchen Fall sollte beim Erzeugen einer WF-Instanz z.B. ein zufällig ausgewählter Server als *ServerManager* festgelegt werden.

[5] Die Sperre verhindert nicht, dass mehrere Migrationen derselben WF-Instanz gleichzeitig durchgeführt werden. Ihr Zweck wird im Zusammenhang mit Algorithmus 3 klar.

[6] Die beiden Sperranforderungen können auch zu einem einzigen Aufruf zusammengefasst werden, um so einen Kommunikationszyklus einzusparen.

Instanz beteiligt ist (*Stay*) oder nicht (*LogOff*). Findet in dem Beispiel aus Abb. 3 die Migration $M_{b,c}$ vor $M_{f,g}$ statt, so wird bei $M_{b,c}$ die Option *Stay* verwendet, da der Server s_1 diese WF-Instanz weiterhin kontrolliert. Dementsprechend erfolgt die später stattfindende Migration $M_{f,g}$ mit der Option *LogOff*, weil hiermit der letzte vom Server s_1 kontrollierte Zweig beendet wird. Dass die beiden Migrationen gleichzeitig ausgeführt werden, ist wegen der zuvor gewährten (exklusiven) Kurzzeitsperre ausgeschlossen. Deshalb ist stets klar, ob ein WF-Server nach Beendigung einer Migration noch an der WF-Instanz beteiligt ist oder nicht. Als nächstes werden die WF-Instanzdaten zum Migrationszielserver übertragen. Da zu diesem Zeitpunkt die exklusive Kurzzeitsperre schon freigegeben wurde (durch *MigrateWorkflowInstance*), können weiterhin mehrere Migrationen derselben WF-Instanz gleichzeitig durchgeführt werden. Der Algorithmus endet damit, dass auch die nicht exklusive Sperre wieder freigegeben wird.

Algorithmus 1 (Durchführung einer Migration)
input
Inst: ID der zu migrierenden WF-Instanz
SourceServer: Quellserver der Migration (führt diesen Algorithmus aus)
TargetServer: Zielserver der Migration
begin
// Verwaltungsserver für die WF-Instanz aus Ablaufhistorie ermitteln
ServerManager = StartServer(Inst);
// nicht exklusive Sperre und exklusive Kurzzeitsperre beim *ServerManager* anfordern[7]
RequestSharedLock(Inst) $\rightarrow$ *ServerManager*;
RequestShortTermLock(Inst) $\rightarrow$ *ServerManager*;
// Menge der aktuellen Server ändern (*UpdateActiveServers*, siehe Algorithmus 2)
if *LastBranch(Inst)* **then**
// die Migration erfolgt im letzten beim *SourceServer* aktiven Ausführungszweig der
// WF-Instanz
UpdateActiveServers(Inst, SourceServer, LogOff, TargetServer) $\rightarrow$ *ServerManager*;
else // ein weiterer Ausführungszweig ist bei *SourceServer* aktiv
UpdateActiveServers(Inst, SourceServer, Stay, TargetServer) $\rightarrow$ *ServerManager*;
// eigentliche Migration durchführen und nicht exklusive Sperre freigeben
MigrateWorkflowInstance(Inst) $\rightarrow$ *TargetServer*;
ReleaseSharedLock(Inst) $\rightarrow$ *ServerManager*;
end.

Der Algorithmus 2 wird vom *ServerManager* verwendet, um die aktuell an einer bestimmten WF-Instanz beteiligten Server zu verwalten. Um diese Aufgabe erfüllen zu können, muss der *ServerManager* zusätzlich die erwähnten Sperren verwalten. Wird die Funktion *UpdateActiveServers* mit der Option *LogOff* aufgerufen, so wird der Migrationsquellserver aus der Menge *ActiveServers(Inst)* der aktuellen WF-Server entfernt, weil dieser Server dann nicht mehr an der WF-Instanz beteiligt ist. Der Migrationszielserver wird stets in diese Menge aufgenommen, unabhängig davon, ob er schon

[7] *p()* $\rightarrow$ s bedeutet, dass die Prozedur *p* aufgerufen wird, die dann vom Server s ausgeführt wird.

enthalten ist oder nicht, da die entsprechende Operation idempotent ist. Durch die vom Algorithmus 1 vor dem Aufruf von *UpdateActiveServers* angeforderte Kurzzeitsperre wird schließlich verhindert, dass der Algorithmus 2 für eine WF-Instanz mehrfach parallel durchlaufen wird, so dass Fehler durch das überlappende Verändern der Menge *ActiveServers(Inst)* ausgeschlossen sind. Nach der Anpassung dieser Menge wird die erwähnte Kurzzeitsperre wieder freigegeben.

Algorithmus 2 (*UpdateActiveServers*: Verwaltung der aktiven WF-Server)
input
Inst: ID der betroffenen WF-Instanz
SourceServer: Quellserver der Migration
Option: gibt an, ob der Quellserver weiterhin in die WF-Instanz involviert ist (*Stay*) oder nicht (*LogOff*)
TargetServer: Zielserver der Migration
begin
// Menge der für die WF-Instanz *Inst* aktiven WF-Server aktualisieren
if *Option = LogOff* **then**
ActiveServers(Inst) = ActiveServers(Inst) $-$ {*SourceServer*};
ActiveServers(Inst) = ActiveServers(Inst) $\cup$ {*TargetServer*};
// die Kurzzeitsperre wieder freigeben
ReleaseShortTermLock(Inst);
end.

3.2.2 Durchführung dynamischer Änderungen

Im vorherigen Abschnitt wurde beschrieben, wie der *ServerManager* die Menge der aktuell an einer WF-Instanz beteiligten Server verwaltet. Im Folgenden wird erläutert, wie diese Menge bei der Durchführung einer dynamischen Änderung verwendet wird. Zu diesem Zweck fordert sie derjenige WF-Server, der eine Änderung durchführen möchte, beim *ServerManager* an.

Wichtig ist, dass sich während der Durchführung einer dynamischen Änderung die Menge der an der betroffenen WF-Instanz beteiligten Server nicht durch Migrationen verändert, weil sonst evtl. die falschen Server in die Änderung einbezogen werden würden. Außerdem darf der Ausführungsgraph einer WF-Instanz nicht gleichzeitig durch mehrere Änderungen umstrukturiert werden, da dies zu einem unzulässigen Graphen führen könnte. Um beides zu verhindern, fordert der Algorithmus 3 beim *ServerManager* eine exklusive Sperre an. Diese entspricht einer Schreibsperre [GR93, HR99] in einem Datenbanksystem und ist mit Lesesperren (*RequestSharedLock* aus Algorithmus 1) und weiteren Schreibsperren derselben WF-Instanz unverträglich. Dadurch wird verhindert, dass für eine zu ändernde WF-Instanz zeitgleich Migrationen stattfinden. Nach Gewährung der Sperre wird die Menge der für diese WF-Instanz aktiven Server erfragt.[8] Bei allen Servern der Menge *ActiveServers* wird nun eine Sperre angefordert,

[8] Dies kann auch mit dem Anfordern der Sperre zu einem einzigen Aufruf zusammengefasst werden, um so einen Kommunikationszyklus einzusparen.

die lokale Veränderungen des Zustands der WF-Instanz verhindert. Bereits gestartete Aktivitäten können aber weiterhin beendet werden, da die entsprechenden Zustände im ADEPT$_{flex}$-Modell äquivalent sind. Dann wird die (gesperrte) Zustandsinformation bei allen an der WF-Instanz beteiligten Servern eingeholt. Der hierdurch resultierende globale und aktuelle Zustand einer WF-Instanz wird benötigt, um prüfen zu können, ob eine bestimmte Änderung zulässig ist oder nicht. So ist in dem Beispiel aus Abb. 3 dem Server s_4, der gerade die Aktivität g kontrolliert und eine Aktivität x nach der Aktivität g und vor d einfügen will, der aktuelle Zustand der Aktivität d des parallelen Zweiges nicht bekannt. Die dynamische Änderung ist aber nur dann zulässig, wenn die Aktivität d zum Zeitpunkt der Änderung noch nicht gestartet wurde [RD98]. Ist dies der Fall, so wird sie bei allen an der WF-Instanz beteiligten Servern durchgeführt (*Perform-DynamicModification*). Anschließend werden die Sperren wieder freigegeben, woraufhin blockierte Migrationen und Änderungsoperationen durchgeführt werden können.

Algorithmus 3 (Durchführung einer dynamischen Änderung)
input
Inst: ID der zu ändernden WF-Instanz
Modification: Spezifikation der dynamischen Änderung
begin
// Verwaltungsserver für die WF-Instanz ermitteln
ServerManager = StartServer(Inst);
// Exklusive Sperre beim *ServerManager* anfordern und aktuelle Servermenge ermitteln
RequestExclusivLock(Inst) $\rightarrow$ *ServerManager*;
ActiveServers = GetActiveServers(Inst) $\rightarrow$ *ServerManager*;
// bei allen Servern Sperre anfordern, aktuellen Zustand ermitteln und ggf. Änderung
// durchführen
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) $\rightarrow s$;
// alle Sperren wieder freigeben
for each Server $s \in$ *ActiveServers* **do**
ReleaseStateLock(Inst) $\rightarrow s$;
ReleaseExclusivLock(Inst) $\rightarrow$ *ServerManager*;
end.

4 Verteilte Ausführung veränderter Workflow-Instanzen

Bei einer Migration muss der aktuelle Zustand der betroffenen WF-Instanz übermittelt werden. Dies geschieht in ADEPT$_{distribution}$, indem die Ablaufhistorie (partiell) übertragen wird [BRD00]. Außerdem werden die Werte von WF-relevanten Daten transferiert. Im Falle von dynamischen Änderungen muss der Migrationszielserver zusätzlich den veränderten Graphen der WF-Instanz kennen, damit er diese korrekt steuern kann. Bei dem im vorherigen Abschnitt vorgestellten Verfahren werden nur die aktuell an der zu verändernden WF-Instanz beteiligten Server in die Änderung einbezogen. Die Server nachfolgender Aktivitäten müssen dagegen ggf. noch über die erfolgten Änderungen informiert werden. Die hierzu erforderliche Informationsübermittlung findet jeweils bei den Migrationen der WF-Instanz zu diesen Servern statt. Da Migrationen recht häufige Operationen sind, muss die Übertragung der entsprechenden Information auf eine effiziente Art und Weise erfolgen. Im Abschnitt 4.1 wird ein Verfahren vorgestellt, das diese Bedingung schon recht gut erfüllt. Dieses wird in Abschnitt 4.2 noch optimiert, so dass redundante Datenübertragungen völlig ausgeschlossen sind.

4.1 Effiziente Übermittlung von dynamischen Änderungen

Im Folgenden wird untersucht, wie ein veränderter WF-Ausführungsgraph dem Zielserver einer Migration bekannt gemacht werden kann. Dabei wird angestrebt, ein bezüglich der Kommunikationskosten möglichst effizientes Verfahren zu erhalten.

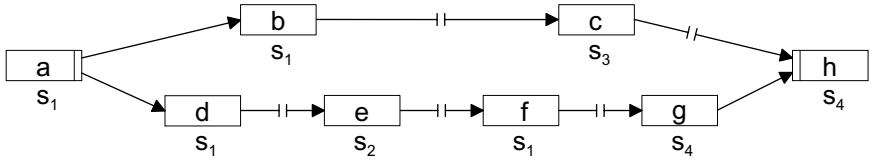
Die einfachste Möglichkeit, um dem Migrationszielserver den aktuellen Ausführungsgraphen bekannt zu machen, ist natürlich, diesen vollständig zu übertragen. Allerdings resultiert aus dieser Vorgehensweise eine unnötig hohe Belastung für das Kommunikationssystem, da diese Graphen viele Knoten und Kanten umfassen können, weshalb ihre Beschreibung sehr groß sein kann. Aus diesem Grund scheidet dieser ineffiziente Ansatz aus.

Es ist auch nicht notwendig, den gesamten Ausführungsgraphen an den Zielserver einer Migration zu übertragen, da dieser Server die zugehörige WF-Vorlage schon kennt. Diese stimmt weitestgehend mit dem aktuellen Ausführungsgraphen der WF-Instanz überein, so dass es wesentlich effizienter ist, lediglich die verhältnismäßig kleine Beschreibung der angewandten Änderungsoperation(en) zu übertragen. Die Verwendung der Änderungshistorie bietet eine gute Möglichkeit, um diese Idee umzusetzen. Diese Historie wird im ADEPT$_{flex}$-Modell vom Migrationszielserver ohnehin benötigt [RD98], so dass aus ihrer Übertragung kein zusätzlicher Aufwand resultiert. Werden die in der Änderungshistorie vermerkten Basisoperationen auf die ursprüngliche WF-Vorlage angewandt, so ergibt sich der benötigte aktuelle Graph der WF-Instanz. Wir erhalten somit ein einfach zu realisierendes Verfahren zur Übermittlung eines Ausführungsgraphen, durch das der Kommunikationsaufwand drastisch reduziert wird. Da auf eine einzelne WF-Instanz in der Regel nur sehr wenige Änderungen angewandt werden, ist der zum Berechnen des aktuellen Ausführungsgraphen notwendige Aufwand gering.

4.2 Optimierung des Verfahrens zur Übermittlung von Änderungshistorien

Im Allgemeinen kann derselbe WF-Server mehrere Male an der Steuerung einer WF-Instanz beteiligt sein. So gibt der Server s_1 in dem Beispiel aus Abb. 4 die Kontrolle nach Beendigung der Aktivität d ab, erhält sie später aber zur Steuerung der Aktivität f wieder zurück. Ein solcher Server kennt deshalb bereits die Änderungshistorieneinträge zu den von ihm selbst durchgeführten Änderungen (da er die Änderungshistorie solange aufbewahrt, bis er über die Beendigung der entsprechenden WF-Instanz informiert wird). Zusätzlich kennt er alle Änderungen, die erfolgt sind, bevor er letztmals die WF-Instanz kontrolliert hat. Der zu solchen Änderungen gehörende Teil der Änderungshistorie muss also nicht mehr zu diesem Server übertragen werden, wodurch das zur Migration des „aktuellen Ausführungsgraphen“ notwendige Datenvolumen weiter reduziert werden kann.

a)



b) Start(a, s_1, ...), DynModif(1), End(a, s_1, ...), DynModif(2), Start(d, s_1, ...), End(d, s_1, ...), Start(e, s_2, ...), DynModif(3), End(e, s_2, ...)

Abbildung 4. a) WF-Instanz und b) Ablaufhistorie[10] des Servers s_2 nach Beendigung der Aktivität e.

4.2.1 Versenden von Änderungshistorieneinträgen

Die naheliegendeste Idee, um redundante Übertragungen von Änderungshistorieneinträgen zu vermeiden, besteht darin, dass der Migrationsquellserver aus der ihm bekannten Ablaufhistorie ermittelt, welche Änderungen dem Zielserver schon bekannt sein müssen. Die entsprechenden Einträge werden dann nicht mehr übertragen. So kann in dem in Abb. 4 dargestellten Beispiel der Server s_2 nach Beendigung der Aktivität e ermitteln, dass der Migrationszielserver s_1 die Änderungen 1 und 2 schon kennen muss. Der Grund dafür ist, dass sich in der Ablaufhistorie die Verweise (*DynModif*) auf diese Änderungen vor dem Eintrag für die Beendigung der vom Server s_1 kontrollierten Aktivität d befinden. Deshalb muss bei der Migration $M_{e,f}$ ausschließlich der Änderungshistorieneintrag zur Änderung 3 übertragen werden.

Allerdings kann mit dem skizzierten Verfahren eine redundante Übertragung von Änderungshistorieneinträgen nicht immer verhindert werden: Bei den Migrationen $M_{f,g}$ und

[10] Bei verteilter WF-Steuerung wird in einer Ablaufhistorie, zusätzlich zu den in Abschnitt 2.2 beschriebenen Werten, in jedem Eintrag vermerkt, welcher Server die entsprechende Aktivität kontrolliert hat.

$M_{c,h}$ zum Server s_4 müssen prinzipiell die Einträge zu den Änderungen 1, 2 und 3 übertragen werden, da der Server s_4 bisher nicht an der Ausführung der WF-Instanz beteiligt war. Der Migrationsquellserver s_1 bzw. s_3 kann aber aus der bei ihm lokal vorhandenen Information nicht ableiten, ob die jeweils andere Migration schon erfolgt ist. Deshalb muss bei beiden Migrationen die gesamte Änderungshistorie übertragen werden. Durch das im nächsten Abschnitt vorgestellte Verfahren wird eine solch redundante Datenübertragung vermieden.

4.2.2 Anfordern von Änderungshistorieneinträgen

Um das im vorherigen Abschnitt beschriebene Problem zu lösen, wird nun ein Verfahren vorgestellt, bei dem die noch benötigten Änderungshistorieneinträge durch den Migrationszielserver explizit angefordert werden. Dieser teilt dem Quellserver bei einer Migration mit, welche Einträge ihm schon bekannt sind, so dass die fehlenden Änderungshistorieneinträge gezielt übertragen werden können. In ADEPT $_{distribution}$ wird für die Übertragung von Ablaufhistorien ein ähnliches Verfahren verwendet, das ebenfalls auf dem Anfordern von noch benötigter Information basiert (siehe [BRD00]). Das Anfordern und Übertragen von Änderungshistorieneinträgen kann im selben Kommunikationszyklus erfolgen, so dass hierfür keine zusätzliche Kommunikation notwendig wird.

Das Anfordern des noch fehlenden Teils der Änderungshistorie ist bei dem hier beschriebenen Verfahren relativ einfach und effizient implementierbar, da stets jeder an einer WF-Instanz beteiligte Server alle bis zum aktuellen Zeitpunkt erfolgten Änderungen kennt. Deshalb ist einem Migrationszielserver der „Anfang“ der Änderungshistorie bekannt, d.h. er verfügt bis zu einer bestimmten Stelle über alle Einträge und ab dieser Stelle kennt er keine weiteren Einträge. Zum Anfordern der fehlenden Teile der Änderungshistorie genügt es also, die ID des letzten bekannten Eintrags an den Quellserver der Migration zu übertragen, woraufhin dieser alle auf diesen Eintrag folgenden Einträge übermittelt.

Das soeben angedeutete Verfahren wird durch den Algorithmus 4 realisiert. Dieser wird von dem Migrationsquellserver als Teil der Prozedur *MigrateWorkflowInstance* des Algorithmus 1 ausgeführt, von der außerdem die Ablaufhistorie und die WF-relevanten Daten übertragen werden (siehe [BRD00]). Der Algorithmus 4 stößt die Übertragung der Änderungshistorie an, indem die ID des neuesten beim Migrationszielserver bekannten Änderungshistorieneintrags erfragt wird. Ist diesem Server noch keine Änderungshistorie zu dieser WF-Instanz bekannt, so gibt er *NULL* zurück. In diesem Fall ist für ihn die gesamte am Quellserver bekannte Änderungshistorie relevant. Andernfalls ist für den Zielserver die Änderungshistorie erst ab dem Historieneintrag relevant, der auf den zurückgemeldeten Eintrag folgt. Der jeweils relevante Teil der Änderungshistorie wird in die Historie *RelevantChangeHistory* kopiert und zum Zielserver übertragen. Dies kann in einer einzigen Übertragung gemeinsam mit den anderen oben erwähnten WF-Daten geschehen.

Die Funktionsweise des Algorithmus 4 soll noch an dem in Abb. 4 dargestellten Beispiel verdeutlicht werden: Bei der Migration $M_{e,f}$ kennt der Zielserver s_1 die dynami-

Algorithmus 4 (Übermittlung einer dynamischen Änderung)
input
Inst: ID der zu ändernden WF-Instanz
TargetServer: Server, an den die Änderungshistorie übertragen wird
begin
// Übertragung der Änderungshistorie anstoßen, indem ID des letzten bekannten Eintrags
// erfragt wird
LastEntry = GetLastEntry(Inst) $\rightarrow$ *TargetServer*;
// für Übertragung relevanten Teil der Änderungshistorie ermitteln
if *LastEntry = NULL* **then**
// am Zielserver ist Änderungshistorie überhaupt noch nicht bekannt
Relevant = True;
else // alle Einträge bis einschließlich *LastEntry* sind am Zielserver schon bekannt
Relevant = False;
// Positionszähler für originale und zu erzeugende Änderungshistorie initialisieren
i = 1; *j* = 1;
// gesamte Änderungshistorie der WF-Instanz *Inst* durchlaufen
while *ChangeHistory(Inst)[i]* $\neq$ *EOF* **do**
if *Relevant = True* **then** // Eintrag ggf. in Ergebnis übernehmen
RelevantChangeHistory[j] = ChangeHistory(Inst)[i];
j = *j* + 1;
// Prüfen, ob das Ende der am Zielserver bekannten Historie erreicht wurde
if *EntryID(ChangeHistory(Inst)[i]) = LastEntry* **then** *Relevant = True*;
i = *i* + 1;
// Übertragung der Änderungshistorie durchführen
TransmitChange(Inst, RelevantChangeHistory) $\rightarrow$ *TargetServer*;
end.

schen Änderungen 1 und 2. Deshalb gibt er *LastEntry* = 2 zurück, woraufhin der Migrationsquellserver die Änderungshistorieneinträge 1 und 2 ignoriert und nur den Eintrag 3 überträgt. Damit ergibt sich dasselbe Ergebnis, wie bei dem im Abschnitt 4.2.1 vorgestellten Ansatz. Für die Migrationen $M_{c,h}$ und $M_{f,g}$ wird o.B.d.A. angenommen, dass $M_{c,h}$ zuerst durchgeführt wird.[11] Da der Server s_4 noch über keine Änderungshistorie zu dieser WF-Instanz verfügt, ergibt sich bei dieser Migration *LastEntry = NULL*, weshalb die gesamte Historie übertragen wird. Bei der anschließend stattfindenden Migration $M_{f,g}$ sind dem Zielserver s_4 die Historieneinträge 1 bis 3 bekannt, weshalb *LastEntry* den Wert 3 erhält. Beim Durchlaufen der While-Schleife von Algorithmus 4 wird die Variable *Relevant* erst dann auf *True* gesetzt, nachdem die Einträge 1 bis 3 bearbeitet wurden. Da keine weiteren Einträge in der Änderungshistorie folgen, bleibt *RelevantChangeHistory* leer, so dass keine Änderungshistorieneinträge übermittelt wer-

[11] Dass die Migrationen gleichzeitig durchgeführt werden, wird durch eine vom Migrationszielserver gehaltene Sperre verhindert. Diese sorgt dafür, dass die zur selben WF-Instanz eingehenden Migrationen serialisiert werden, d.h. die Sperre wird ab dem Anstoßen der Migration, während dem Ermitteln und Übertragen der Änderungshistorieneinträge (und der anderen WF-Daten [BRD00]), bis zur Integration der Einträge in die Änderungshistorie gehalten. Diese Sperre verhindert, dass Einträge redundant angefordert werden, weil von veralteter lokaler Information ausgegangen wird.

den. Das aus Abschnitt 4.2.1 bekannte Problem der redundanten Datenübertragung wird hier also vermieden.

Es ist somit nicht nur möglich, dynamische Änderungen in einem verteilten WfMS effizient durchzuführen (siehe Abschnitt 3), veränderte WF-Instanzen können außerdem mit sehr geringen Übertragungskosten migriert werden.

5 Diskussion

In der WF-Literatur finden sich zahlreiche Arbeiten, die sich mit Skalierbarkeitsfragestellungen und verteilter WF-Ausführung beschäftigen (z.B. [AKA+94, AMG+95, BMR96, CGP+96, GJS+99, GT98, Jab97, MWW+98, SK97, SM96, Wes99]), einen umfassenden Überblick bietet [BD99b]. Ebenso gibt es viele Veröffentlichungen zu dem Thema dynamische WF-Änderungen (z.B. [BPS97, CCPP98, DMP97, EM97, HK96, HSS96, JH98, KG99, KRW90, LP98, MR00, Sie98, Wes98]), die in [Rei00] diskutiert werden. Jedoch gibt es kaum Projekte, die beide Aspekte gemeinsam betrachten – insbesondere wird deren Zusammenspiel nicht hinreichend gewürdigt. Es ist nicht das Ziel dieser Arbeiten, ein bezüglich der Kommunikationskosten effizientes WfMS zu entwickeln, das skalierbar und flexibel ist. Dieser Aspekt wird in der vorliegenden Arbeit erstmalig systematisch untersucht.

WIDE erlaubt dynamische Änderungen einer WF-Vorlage und deren Propagierung auf laufende WF-Instanzen [CCPP98]. Außerdem werden WF-Instanzen verteilt gesteuert [CGP+96], wobei die Bearbeiter einer Aktivität den WF-Server determinieren, der diese Aktivität kontrolliert. Bei MOKASSIN [GJS+99, JH98] und WASA [Wes98, Wes99] wird die verteilte WF-Ausführung durch eine zugrunde liegende CORBA-Infrastruktur realisiert. Außerdem sind Änderungen auf Schema- und auf Instanzebene möglich, wobei auch Konsistenzfragestellungen betrachtet werden. INCAs [BMR96] verwirklicht die Steuerung der WF-Instanzen durch Regeln, die modifiziert werden können, um dynamische Änderungen durchzuführen. Die WF-Steuerung findet verteilt statt, wobei eine WF-Instanz stets von dem Rechner desjenigen Benutzers kontrolliert wird, der die aktuelle Aktivität bearbeitet. Bei all diesen Ansätzen wird aber nicht explizit auf das Zusammenspiel der dynamischen Änderungen und der verteilten WF-Ausführung eingegangen.

In der Literatur finden sich auch einige Ansätze für verteiltes WF-Management, bei denen eine WF-Instanz während ihrer gesamten Lebenszeit von nur einem einzigen WF-Server kontrolliert wird (z.B. Exotica [AKA+94], MOBILE [Jab97] - wurde aber in [SNS99] erweitert). Es finden also keine Migrationen statt, unterschiedliche WF-Instanzen können aber von verschiedenen Servern kontrolliert werden. Da für jede WF-Instanz eine zentrale Kontrollinstanz existiert, könnten dynamische Änderungen bei diesen Ansätzen also genauso wie in einem zentralen WfMS durchgeführt werden. Allerdings ist es bei diesem Verteilungsmodell nicht möglich, für jede einzelne Aktivität einen bezüglich der Kommunikationskosten günstigen WF-Server auszuwählen. Da deshalb bei der „normalen WF-Ausführung“ höhere Mehrkosten entstehen, als die bei den (verhältnismäßig seltenen) dynamischen Änderungen erzielten Einsparungen, wurde für ADEPT kein solcher Ansatz gewählt.

6 Zusammenfassung und Ausblick

Verteilte WF-Ausführung und dynamische Änderungen sind essentiell, um WfMS auch zur Unterstützung von anspruchsvollen vorgangsorientierten Anwendungssystemen einsetzen zu können. Allerdings sind mit diesen beiden Aspekten teilweise entgegengesetzte Anforderungen und Ziele verbunden, da die für dynamische Änderungen notwendige zentrale Kontrollinstanz die Effizienz der verteilten WF-Ausführung beeinträchtigt. Deshalb können die beiden Themen nicht getrennt voneinander betrachtet werden. Ihre Wechselwirkungen werden in dieser Arbeit erstmalig untersucht, mit dem Ergebnis, dass die beiden Aspekte durchaus vereinbar sind. Es ist gelungen, dynamische Änderungen in einem verteilten WfMS auf effiziente Art und Weise zu realisieren. Auch die verteilte Steuerung einer zuvor veränderten WF-Instanz ist äußerst effizient möglich, da zur Übermittlung des modifizierten Ausführungsgraphen lediglich ein Teil der ohnehin relativ kleinen Änderungshistorie übertragen werden muss. Dies ist besonders wichtig, da Migrationen häufige Operationen sind. Zusammenfassend lässt sich feststellen, dass es gelungen ist, verteilte WF-Ausführung und dynamische Änderungen nahtlos in ein System zu integrieren.

Alle in diesem Beitrag vorgestellten Verfahren wurden in dem WfMS-Prototypen ADEPT$_{workflow}$ implementiert [Zei99]. Dieser Prototyp wurde auf der CeBIT'2000, der EDBT'2000 [HRB+00] und der BIS'2000 [DRK00] einem breiten Publikum präsentiert. Durch die Implementierung konnte die Umsetzbarkeit der Verfahren gezeigt werden. Außerdem ist bei der Durchführung von Migrationen zu erkennen, dass nur moderate Datenmengen übertragen werden.

Optimierungspotential besteht noch hinsichtlich der Wahl der WF-Server, die in eine dynamische Änderung einbezogen werden müssen: Betrifft eine Änderung nur einen Ausschnitt des WF-Graphen, so könnte eine Änderung auch nur von einem Teil der aktuell in die WF-Instanz involvierten WF-Server durchgeführt werden, wodurch der Synchronisations- und Kommunikationsaufwand reduziert wird. Im Extremfall wird nur ein einziger Zweig einer Parallelität verändert, weshalb eigentlich auch nur ein Server für die Durchführung der Änderung benötigt wird. Allerdings können Aktivitäten paralleler Ausführungszweige (z.B. durch Abhängigkeiten im Datenfluss oder durch entsprechend festgelegte temporale Bedingungen) von dieser Änderung betroffen sein, so dass die zugehörigen Server in diesem Fällen doch einbezogen werden müssen. Unsere bisherigen Untersuchungen ergaben, dass eine solche Optimierung deshalb eher selten eingesetzt werden kann, so dass eine signifikante Verbesserung des Systemverhaltens nicht zu erwarten ist. Dennoch bietet dieser Aspekt Ansatzpunkte für weitergehende Forschung.

Danksagung: Wir danken Clemens Hensinger, Thomas Strzeletz und Jochen Zeitler für die anregenden Diskussionen.

Literatur

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör und C. Mohan: *Failure Handling in Large Scale Workflow Management Systems.* Technischer Bericht

RJ9913, IBM Almaden Research Center, November 1994.

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

[BD97] T. Bauer und P. Dadam: *A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration*. In: *Proc. 2nd IFCIS Conf. on Cooperative Information Systems*, S. 99–108, Kiawah Island, SC, Juni 1997.

[BD99a] T. Bauer und 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*, S. 25–32, Paderborn, Oktober 1999.

[BD99b] T. Bauer und P. Dadam: *Verteilungsmodelle für Workflow-Management-Systeme – Klassifikation und Simulation*. Informatik Forschung und Entwicklung, 14(4):203–217, Dezember 1999.

[BD00] T. Bauer und P. Dadam: *Efficient Distributed Workflow Management Based on Variable Server Assignments*. In: *Proc. 12th Conf. on Advanced Information Systems Engineering*, S. 94–109, Stockholm, Juni 2000.

[BMR96] D. Barbará, S. Mehrotra und M. Rusinkiewicz: *INCAs: Managing Dynamic Workflows in Distributed Environments*. Journal of Database Management, Special Issue on Multidatabases, 7(1):5–15, 1996.

[BPS97] P. Bichler, G. Preuner und M. Schrefl: *Workflow Transparency*. In: *Proc. 9th Int. Conf. on Advanced Information Systems Engineering*, S. 423–436, Barcelona, 1997.

[BRD00] T. Bauer, M. Reichert und P. Dadam: *Effiziente Durchführung von Prozessmigrationen in verteilten Workflow-Management-Systemen*. Ulmer Informatik-Berichte 2000-08, Universität Ulm, Fakultät für Informatik, Juni 2000.

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

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

[DKR+95] P. Dadam, K. Kuhn, M. Reichert, T. Beuter und M. Nathe: *ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen*. In: *Proc. GI/SI-Jahrestagung*, S. 677–686, Zürich, September 1995.

[DMP97] B. Dellen, F. Maurer und G. Pews: *Knowledge Based Techniques to Increase the Flexibility of Workflow Management*. Data & Knowledge Engineering, 1997.

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

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

[GJS+99] B. Gronemann, G. Joeris, S. Scheil, M. Steinfort und 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, September 1999.

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

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

[HK96] M. Hsu und C. Kleissner: *ObjectFlow: Towards a Process Management Infrastructure*. Distributed & Parallel Databases, 4:169–194, 1996.

[HR99] T. Härder und E. Rahm: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer-Verlag, 1999.

[HRB+00] C. Hensinger, M. Reichert, T. Bauer, T. Strzeletz und P. Dadam: *ADEPT$_{workflow}$ - Advanced Workflow Technology for the Efficient Support of Adaptive, Enterprise-wide Processes*. In: *7th Int. Conf. on Extending Database Technology, Software Demonstrations Track*, S. 29–30, Konstanz, März 2000.

[HSS96] P. Heinl, H. Schuster und H. Stein: *Behandlung von Ad-hoc-Workflows im MOBILE Workflow-Modell*. In: *Proc. Softwaretechnik in Automation und Kommunikation – Rechnergestützte Teamarbeit*, S. 229–242, München, März 1996.

[Jab97] S. Jablonski: *Architektur von Workflow-Mangement-Systemen*. Informatik Forschung und Entwicklung, Themenheft Workflow-Management, 12(2):72–81, 1997.

[JBS97] S. Jablonski, M. Böhm und W. Schulze: *Workflow-Management: Entwicklung von Anwendungen und Systemen; Facetten einer neuen Technologie*. dpunkt-Verlag, 1997.

[JH98] G. Joeris und O. Herzog: *Managing Evolving Workflow Specifications*. In: *Proc. 3rd IFCIS Conf. on Cooperative Information Systems*, New York, August 1998.

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

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

[KRW90] B. Karbe, N. Ramsperger und P. Weiss: *Support of Cooperative Work by Electronic Circulation Folders*. In: *Proc. Conf. on Office Information Systems*, S. 109–117, Cambridge, MA, 1990.

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

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

[MR00] R. Müller und E. Rahm: *Dealing with Logical Failures for Collaborating Workflows*. In: *Proc. 5th Int. Conf. on Cooperative Information Systems*, S. 210–223, Eilat, September 2000.

[MWW+98] P. Muth, D. Wodtke, J. Weißenfels, A. Kotz-Dittrich und G. Weikum: *From Centralized Workflow Specification to Distributed Workflow Execution*. Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems, 10(2):159–184, März/April 1998.

[Obe96] A. Oberweis: *Modellierung und Ausführung von Workflows mit Petri-Netzen*. Teubner-Verlag, 1996.

[RBD99] M. Reichert, T. Bauer und P. Dadam: *Enterprise-Wide and Cross-Enterprise Workflow-Management: Challenges and Research Issues for Adaptive Workflows*. In: *Proc. Workshop Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications, 29. Jahrestagung der GI*, S. 56–64, Paderborn, Oktober 1999.

[RD98] M. Reichert und P. Dadam: *ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control*. Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems, 10(2):93–129, März/April 1998.

[Rei00] M. Reichert: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. Dissertation, Universität Ulm, Fakultät für Informatik, Juli 2000.

[Sie98] R. Siebert: *An Open Architecture for Adaptive Workflow Management Systems*. In: *Proc. 3rd Biennial World Conf. on Integrated Design and Process Technology, Vol. 2 - Issues and Applications of Database Technology*, S. 79–85, Berlin, Juli 1998.

[SK97] A. Sheth und 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*, S. 12–21, Istanbul, August 1997.

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

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

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

[Wes99] M. Weske: *Workflow Management Through Distributed and Persistent CORBA Workflow Objects*. In: *Proc. 11th Int. Conf. on Advanced Information Systems Engineering*, Heidelberg, 1999.

[Zei99] J. Zeitler: *Integration von Verteilungskonzepten in ein adaptives Workflow-Management-System*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, 1999.