Konzeption eines flexiblen Workflow-Management-Systems für Corba-Architekturen

Mathias Weske, Jens Hündling, Dominik Kuropka, Hilmar Schuschel
Lehrstuhl für Informatik, Universität Münster
Steinfurter Straße 107, D-48149 Münster
{weske,hundlin,kuropka,schusch}@helios.uni-muenster.de

Zusammenfassung

Dieser Beitrag enthält wesentliche konzeptionelle Überlegungen bei der Entwicklung eines flexiblen Workflow-Management-Systems auf der Basis einer Corba-Architektur. Ausgehend von einem Objektmodell, das die relevanten Entitäten klassifiziert und ihre Beziehungen abbildet, wird eine Einordnung der Workflow-Objekte und -Dienste in das Umfeld spezifizierter Corba-Dienste geliefert. Schließlich werden Zustände und Zustandsübergänge von Workflow-Instanzen beschrieben und Realisierungsalternativen für verteilte Workflow-Ausführungen skizziert.

1 Einführung

Workflow-Management-Systeme sind komplexe Software-Systeme, die zur Modellierung und kontrollierten Ausführung von Anwendungsprozessen in unterschiedlichen Umgebungen eingesetzt werden [6, 15, 11, 23]. Standen in der ersten Phase des Einsatzes von Workflow-Technologie kommerzielle Büroanwendungen mit einer fest vordefinierten Struktur im Vordergrund [12, 8], die auf einer homogenen Software-Plattform ausgeführt wurden, so finden Workflow-Techniken in zunehmendem Maß in naturwissenschaftlichen [21, 27] und technischen [13] Umgebungen Verwendung, wo die Prozesse meist weniger stark strukturiert sind [29] und wo es sich typischerweise um heterogene Software-Umgebungen handelt.

Zur Integration verteilter Anwendungen in heterogenen Umgebungen wurde die Corba-Architektur entwickelt [25]. Derzeit sind eine Reihe von Corba-Diensten (Common Object Services, COS) bereits vollständig spezifiziert [17], und zentrale Dienste sind in kommerziell verfügbaren Produkten [10, 26] implementiert. Bei der Verwendung von Corba-Technologie werden Dienste, die von unterschiedlichen Software-Systemen eines verteilten Systems erbracht werden können, durch eine einheitliche Schnittstellenbeschreibung und eine Software-Komponente (den Object-Request-Broker, ORB), die die Vermittlung der Dienste übernimmt, integriert. Da verteilte Corba-Umgebungen eine integrierte und plattformunabhängige Architektur komponentenbasierter verteilter Systeme darstellen und Workflow-Management auf die Integration verteilter Anwendungen abzielt, ist die

Verwendung von Corba-Technologie zur Entwicklung von Workflow-Management-Systemen ein angemessener Ansatz.

Die Entwicklung von Workflow-Management-Systemen auf der Basis der Object Management Architecture ist derzeit Gegenstand von Standardisierungsbemühungen der Object Management Group, wobei bestehende Corba-Dienste so weit als möglich verwendet werden sollen [17]. Der Annahmeprozeß einer *Workflow Facility* befindet sich derzeit (November 97) im Zustand *initial submissions received* [18]. Erste Begutachtungen der Einreichungen haben eine Reihe von Defiziten herausgearbeitet [22], von denen die wichtigsten weiter unten skizziert werden. In diesem Zusammenhang sind auch Aktivitäten der OMG zu nennen, die auf eine Definition einer *Business Object Facility* abzielen [2]. Business-Objekte stellen spezielle Strukturen und Funktionen in gekapselter Form zur Verfügung, die über Corba-Schnittstellen abgerufen werden können, die in der Business Object Facility definiert werden. Da der Integrationsaspekt bei Workflow-Anwendungen eine wichtige Rolle spielt und da Business-Objekte im Rahmen von Workflow-Anwendungen wichtige Funktionen übernehmen können, ist eine Corba-Basierung für Workflow-Management-Systeme auch aus diesem Grund angemessen. So kann ein Workflow-Management-System über Corba-Schnittstellen auf die Funktionalität von Business-Objekten zugreifen; darüber hinaus können Business-Objekte im Rahmen von Datenfluß zwischen Workflow-Aktivitäten transferiert werden.

Die in diesem Beitrag vorgestellten Ansätze und Konzepte wurden im Rahmen des WASA-Projekts an der Universität Münster erarbeitet; dieses Projekt beschäftigt sich mit der Unterstützung flexibler Workflows [27, 29] in unterschiedlichen Anwendungsgebieten. Die bei der Entwicklung einer ersten Version eines WASA-Prototypen [28] gesammelten Erfahrungen haben die hier dargestellten Überlegungen nachhaltig beeinflußt. In diesem Bericht beschreiben wir die konzeptionellen Überlegungen der Entwicklung eines flexiblen Workflow-Management-Systems auf der Basis einer Corba-Architektur. Dabei wird ausgehend von einem Analyse-Objektmodell, welches die Objekte und deren Beziehungen aus logischer Sicht modelliert, ein (implementierungsnäheres) Design-Objektmodell entwickelt, das insbesondere die Beziehungen des zu entwicklenden Software-Systems zu bereits bestehenden bzw. spezifizierten Corba-Diensten beschreibt. Basierend auf diesem Objektmodell beschreiben wir Zustände und Zustandsübergänge von Workflow-Instanzen, die unter anderem festlegen, in welchem Zustand eine Methode aufgerufen werden kann und welchen Zustandsübergang durch ihre Ausführung bewirkt wird. Da das WASA-Projekt auf die flexible Unterstützung von Workflows abzielt, werden Methoden untersucht, die dynamische Modifikationen und Benutzer-Interventionen in systemkontrollierte Workflow-Ausführungen unterstützen [29]. Eine Diskussion verwandter Arbeiten, insbesondere eine Würdigung der Einreichungen zur OMG Workflow Facility, und zusammenfassende Bemerkungen schließen diesen Beitrag.

2 Workflow-Grundlagen

Um das Ziel der kontrollierten Ausführung von Anwendungsprozessen zu erreichen, benötigen Workflow-Management-Systeme eine von ihnen zu verarbeitende Spezifikation dieser Prozesse, die als Workflow-Modell bezeichnet wird. Zur Spezifikation von Workflow-Modellen werden Workflow-Sprachen verwendet [30]. Wir verwenden im weiteren Verlauf dieses Beitrags eine graph-basierte Workflow-Sprache [29], bei der Workflow-Modelle als geschachtelte gerichtete Graphen repräsentiert

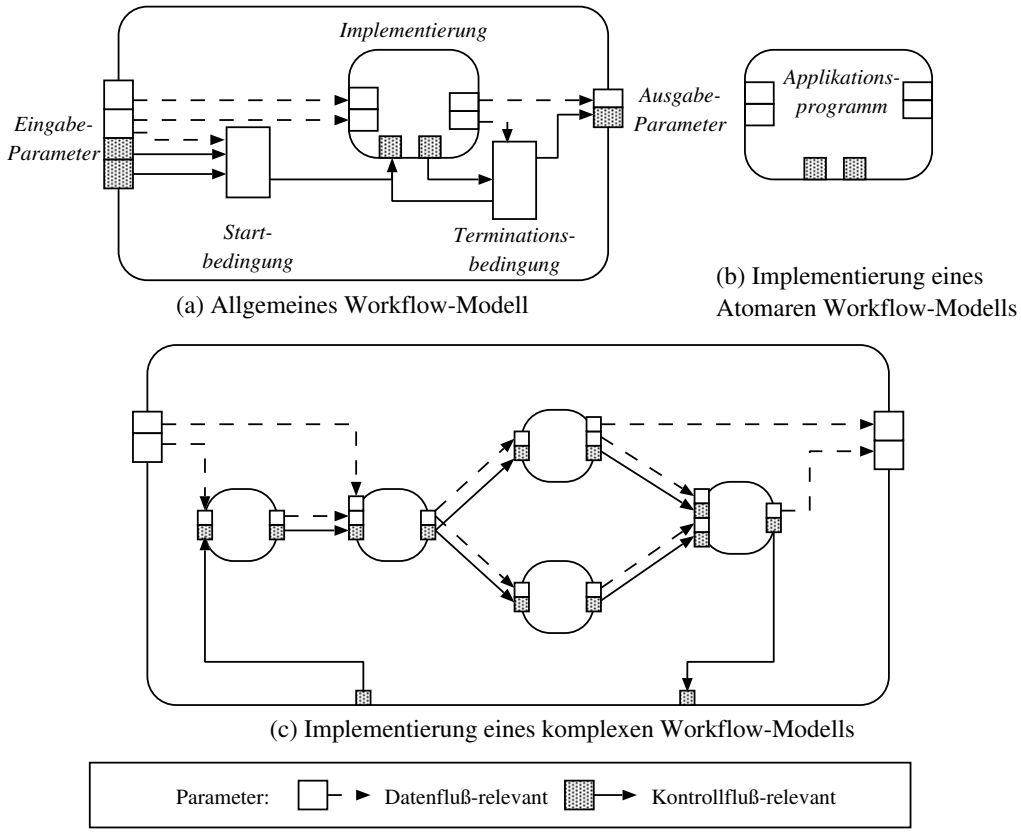

Abbildung 1: Struktur von Workflow-Modellen

werden. Die Schachtelung wird zur Modularisierung von Workflow-Modellen verwendet, mit den Zielen der Wiederverwendbarkeit von Workflow-Modellen und der Komplexitätsreduktion. Die Hierarchisierung von Workflow-Modellen wird dadurch realisiert, daß komplexe und atomare Workflow-Modelle zur Verfügung stehen. Während komplexe Workflow-Modelle aus einer Menge (komplexer oder atomarer) Workflow-Modelle und deren Beziehungen bestehen, sind atomare Workflow-Modelle nicht weiter verfeinert; sie werden meist durch Applikationsprogramme realisiert, deren Ausführung durch das Workflow-Management-System angestoßen wird. Findet keine Benutzerinteraktion bei der Ausführung eines solchen Applikationsprogramms statt, so handelt es sich um einen automatischen Workflow, sonst um einen manuellen Workflow.

Ein durch ein komplexes Workflow-Modell charakterisierter Anwendungsprozeß wird aufgrund seiner relativen Position zu den anderen Workflows auch als Toplevel-Workflow bezeichnet; er besteht aus einer Menge von Sub-Workflows. Diese Sub-Workflows können eine Reihe von Beziehungen zueinander besitzen, etwa Datenfluß- und Kontrollflußbeziehungen. Um Datenfluß geeignet modellieren zu können, besitzt jeder Workflow eine Menge von Eingabeparametern und eine Menge von Ausgabeparametern. Durch Verbindung eines Ausgabeparameters eines Workflows i mit einem Eingabeparameters eines Workflows j wird ein Datenfluß von i nach j modelliert.

Die Abbildung 1 zeigt die Struktur eines Workflow-Modells, wobei in Teil (a) der Abbildung neben Ein- und Ausgabeparametern auch eine Startbedingung und eine Terminationsbedingung

skizziert sind. Im allgemeinen wird ein Workflow nur dann gestartet, wenn seine Startbedingung zu wahr ausgewertet wird; ein Workflow terminiert, wenn seine Terminationsbedingung wahr ist. Die möglichen Verfeinerungen (bzw. Implementierungen) eines Workflow-Modells unter Verwendung eines Applikationsprogramms (falls es sich um das Workflow-Modell eines atomaren Workflows handelt) bzw. eines komplexen Workflow-Modells sind in den Teilen (b) bzw. (c) von Abbildung 1 dargestellt. Eine detailliertere Beschreibung der Zusammenhänge erfolgt bei der Vorstellung der Objektmodelle im folgenden Abschnitt.

3 Objektorientierte Modellierung von Workflow-Systemen

Ein Objektmodell beschreibt die statische Struktur von Objekten in einem System und ihre Beziehungen untereinander. Wir unterscheiden dabei zwischen Analyse-Objektmodell und Design-Objektmodell. Bei der Entwicklung eines Systems wird zunächst das Analyse-Objektmodell erstellt. Ziel hierbei ist es, die Struktur des Systems formal zu beschreiben, ohne bereits festzulegen, wie die Implementierung des Systems erfolgen soll. Durch Ergänzung des Modells um Implementierungsdetails erhalten wir das Design-Objektmodell. Dient das Analyse-Objektmodell dem Verständnis der Anwendung, so liegt der Schwerpunkt der Betrachtungen beim Übergang zum Design-Objektmodell auf einer angemessenen und effizienten Implementierung des Systems.

In diesem Abschnitt erfolgt zunächst die Beschreibung des Analyse-Objektmodells, gefolgt von der Darstellung des Design-Objektmodells. Zur Darstellung von Objektmodellen verwenden wir die Unified Modeling Language (UML) [19]. Abschließend werden in diesem Abschnitt Ausführungsalternativen für verteilte Workflow-Management-Systeme vorgestellt und dabei untersucht, welche der vorgestellten Alternativen unsere Anforderungen erfüllt.

3.1 Analyse-Objektmodell

Das zentrale Element des Analyse-Objektmodells ist die Klasse `Workflow` (siehe Abbildung 2). Auf Klassenebene wird nicht zwischen Workflow-Modell und Workflow-Instanz unterschieden. Stattdessen modellieren wir Workflow-Objekte und unterscheiden lediglich zwischen den Zuständen *Modell* und *Instanz*, die ein Workflow-Objekt annehmen kann. (Ein "Workflow-Objekt im Zustand Instanz" wird auch als *Workflow-Instanz* bezeichnet; analog wird ein "Workflow-Objekt im Zustand Modell" auch als Workflow-Modell bezeichnet.)

Die in Abschnitt 2 vorgenommene Unterscheidung zwischen atomaren und komplexen Workflows findet sich auch in dem Analyse-Objektmodell wieder. So sind die Klassen `Atomic` und `Complex` Sub-Klassen der Klasse `Workflow`, d.h., es besteht eine Generalisierungsbeziehung zwischen diesen Klassen, wobei man beim Übergang zur Klasse `Workflow` von der Komplexität eines Workflows abstrahiert. Wie bei der Vorstellung der Workflow-Grundlagen diskutiert wurde, können atomare Workflows automatisch oder manuell zu bearbeiten sein; daher sind `Automatic` und `Manual` Sub-Klassen von `Atomic`.

Die Beziehung eines Sub-Workflows zu seinem Super-Workflow wird durch die `WF-SubWF Relationship` modelliert. Während ein Workflow-Modell im allgemeinen mehreren komplexen Workflows zugeordnet sein kann, ist eine Workflow-Instanz maximal einer komplexen Workflow-Instanz zugeordnet. Bezüglich dieser Beziehung gilt die folgende Einschränkung (die in Abbildung 2 durch

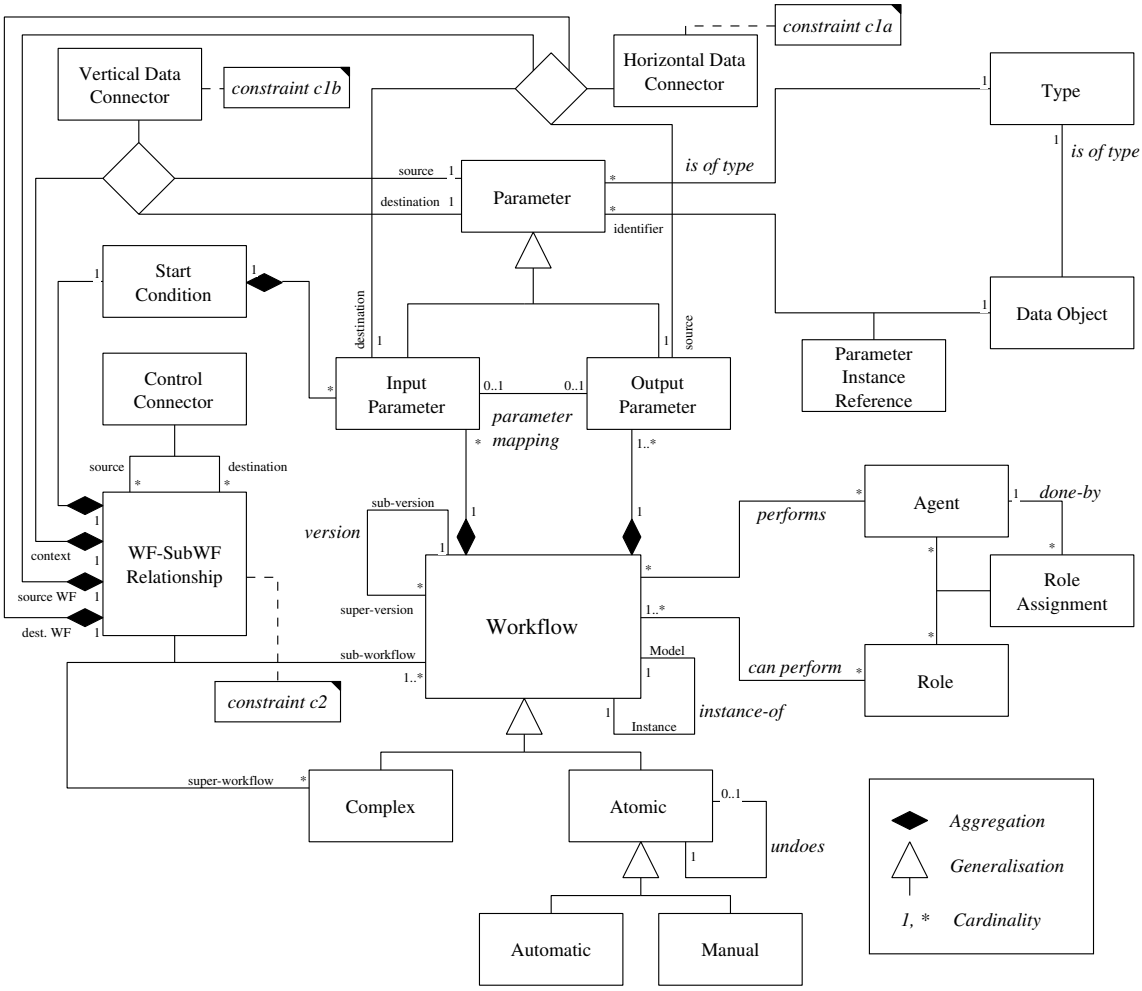

Abbildung 2: Analyse-Objektmodell

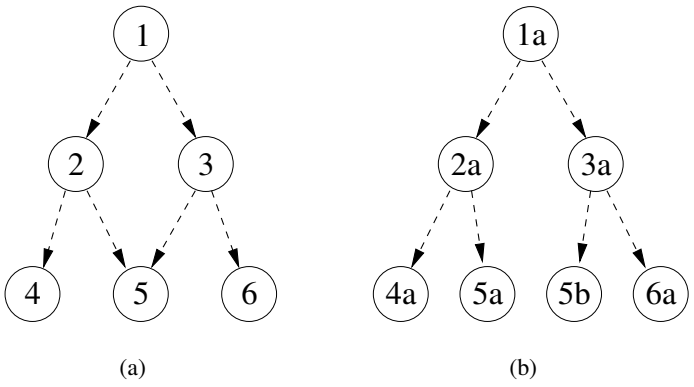

Abbildung 3: Workflow-Modell (a) versus Workflow-Instanz (b)

constraint c2 bezeichnet ist): Die Workflow-Sub-Workflow-Relationship muß generell zyklenfrei sein; solche Beziehungen können ausschließlich zwischen zwei Workflow-Modellen oder zwischen zwei Workflow-Instanzen auftreten. Eine Menge von Workflow-Instanzen, die miteinander in Beziehung stehen, bilden darüber hinaus eine Baumstruktur aus, d.h., jede Workflow-Instanz hat höchstens einen Vorgänger. In Abbildung 3 ist ein Workflow-Modell und eine davon erzeugte Workflow-Instanz zu sehen. Da das Workflow-Modell 5 in den Modellen 2 und 3 als Sub-Workflow-Modell erscheint, werden bei der Erzeugung der Workflow-Instanz 1 zwei Workflow-Instanzen unter Verwendung des Workflow-Modells 5 erzeugt, die in Abbildung 3 mit 5a bzw. 5b bezeichnet sind.

Eine wichtige Eigenschaft von Workflow-Modellen ist die Abbildung von Datenfluß zwischen Workflows; hierzu werden Datenkonnektoren verwendet. Wir unterscheiden horizontalen und vertikalen Datenfluß, wobei ein horizontaler Datenfluß zwei Sub-Workflows eines gemeinsamen komplexen Workflows miteinander verbindet, während ein vertikaler Datenfluß zwischen einem Super-Workflow und seinen Sub-Workflows bestehen kann. Dabei ist ein Datenfluß von dem Super-Workflow zu seinen Sub-Workflows sowie ein Datenfluß in entgegengesetzter Richtung möglich. Bei dem horizontalen Datenfluß zwischen Workflow-Modellen i und j eines Super-Workflows k muß beachtet werden, daß in beiden Workflow-Sub-Workflow-Relationships k als Super-Workflow erscheint (*constraint c1a*). Beim vertikalen Datenfluß ist zu beachten, daß entweder ein Eingabeparameter eines Super-Workflows mit einem Eingabeparameter eines Sub-Workflows oder ein Ausgabeparameter eines Sub-Workflows mit einem Ausgabeparameter des Super-Workflows verbunden ist (*constraint c1b*). Durch diese Modellierung ist es möglich, Workflow-Modelle in verschiedenen komplexen Workflow-Modellen wieder zu verwenden, wobei die jeweils modellierten Datenflüsse sich ausschließlich auf die jeweilige Verwendung des Workflow-Modells im Rahmen eines komplexen Workflow-Modells beziehen.

Neben Datenfluß spielt die Modellierung von Kontrollfluß eine wichtige Rolle bei der Workflow-Modellierung. Ein Kontrollkonnektor stellt eine binäre Beziehung zwischen zwei Workflow-Sub-Workflow-Beziehungen dar und legt mögliche Ausführungsreihenfolgen der Sub-Workflows fest. Startbedingungen werden bei der Workflow-Modellierung benötigt, um explizit anzugeben, unter welchen Bedingungen ein Workflow ausgeführt werden kann. In unserem Modell kann einem Workflow im Kontext eines komplexen Workflows eine Startbedingung zugeordnet werden (siehe

Abb. 1). Dies wird analog zum Kontrollkonnektor durch die Beziehung zwischen der Klasse `Start Condition` und der Klasse `WF-SubWF Relationship` zum Ausdruck gebracht.

Bei Parametern wird durch eine Generalisierungsbeziehung zwischen Input- und Output-Parametern unterschieden. Die Klasse `Input Parameter` besitzt im Gegensatz zur Klasse `Output Parameter` eine Aggregationsbeziehung zur Klasse `Start Condition`. Damit wird es bei der Implementierung der Klasse `Start Condition` möglich, die Eingabe-Parameter des Workflows zu verwenden, um zur Laufzeit anhand übergebener Werte die Startbedingung auszuwerten.

Um die Beziehung zwischen Workflow-Modellen und Workflow-Instanzen zu verwalten, besitzt jede Workflow-Instanz eine `instance of`-Beziehung zu einem Workflow-Modell, nämlich zu dem Modell, von dem sie instantiiert wurde. Diese Beziehung ist notwendig, um Veränderungen eines Modells, falls gewünscht, auf die davon abgeleiteten Workflow-Instanzen zu übertragen. Zur Laufzeit von Workflows werden im Rahmen der Rollenauflösung Agenten zur Ausführung konkreter Aktivitäten ausgewählt. Dabei sind jedem Workflow eine oder mehrere Rollen zugeordnet. Es kann sich dabei um eine in bestimmter Weise qualifizierte Person handeln oder auch um ein Anwendungsprogramm. Zwischen Rollen und Agenten besteht eine *n:m*-Beziehung, die festlegt, welche Rollen von welchen Agenten übernommen werden können.

Das Rollenkonzept wird durch die Klassen `Role` und `Agent` realisiert. Rollen sind Workflow-Modellen zugeordnet und abstrahieren von konkreten Agenten. Beispielsweise werden dem Workflow *Kunde_Benachrichtigen* nicht direkt die Mitarbeitern *Müller* und *Schulz* zugeordnet, sondern die Rolle *Sachbearbeiter*. Dies hat den Vorteil, daß Workflow-Modelle unabhängig von den momentan zur Verfügung stehenden Mitarbeitern und Mitarbeiterinnen formuliert werden können. Während bei atomaren Workflows eine zugeordnete Rolle zum Ausdruck bringt, welcher Agent den Workflow bearbeiten kann, drückt die Rolle eines komplexen Workflows die Verantwortlichkeit für diesen Workflow aus. Die Beziehung zwischen den Klassen `Agent` und `Workflow` existiert nur für Workflow-Instanzen und beschreibt, welcher Agent eine konkrete Instanz bearbeitet.

Zu Abschluß wird nun eine mögliche Modellierungsalternative dargestellt, die eine Unterscheidung zwischen Modell und Instanz auf Klassenebene vornimmt. So könnte man eine Workflow-Instanz als Spezialisierung von einem Workflow-Modell ansehen oder sowohl Modell als auch Instanz als Spezialisierung einer abstrakten, dritten Klasse, welche die Gemeinsamkeiten zwischen Modell und Instanz zusammenfaßt. Beide Arten der Modellierung erweisen sich bei näherer Betrachtung aber als problematisch, da sich die Beziehungen nicht auf die Workflow-Klassen beschränken, sondern auch zwischen vielen zu den Workflow-Klassen in Beziehung stehenden Klassen auftreten. Angenommen man modelliert eine Workflow-Instanz als Spezialisierung von einem Workflow-Modell. Die Klasse `Workflow-Modell` besitzt eine Input- und eine Output-Aggregationsbeziehung zur Klasse `Parameter`. Diese Beziehungen zur Klasse `Parameter` würden nun an die Klasse `Workflow-Instanz` vererbt. Allerdings können Parameter, die mit Workflow-Instanzen in Beziehung stehen, im Gegensatz zu Parametern, die mit Workflow-Modellen in Beziehung stehen auch Beziehungen zu Daten-Objekten eingehen. Somit sind diese Parameter nicht als Objekte einer Klasse modellierbar, sondern es gilt zwischen Modell- und Instanzparametern zu unterscheiden. Hierbei stellt die Klasse `Instanzparameter` eine Spezialisierung der Klasse `Modellparameter` dar.

Die Vielzahl der Spezialisierungsbeziehungen und Mehrfachvererbungen, die bei dieser Art der Modellierung entstehen würden, machen das alternative Objektmodell unübersichtlich und tragen

nur wenig zum Verständnis der Struktur des Problems bei. Aus diesen Gründen haben wir uns dafür entschieden, auf eine Unterscheidung zwischen Workflow-Modell und Workflow-Instanz auf Klassenebene zu verzichten und sie statt dessen als Objekte der Klasse `Workflow` zu modellieren, welche sich lediglich in ihrem Zustand unterscheiden. Hieraus ergibt sich, daß das Objektmodell so allgemein formuliert werden muß, daß es sowohl Workflow-Modelle als auch Workflow-Instanzen beschreiben kann. Ein ähnlicher Ansatz findet sich in [9]. In Abhängigkeit davon, ob ein Workflow im Zustand *Modell* oder *Instanz* ist, gelten nun unterschiedliche Konsistenzbedingungen. Beispielsweise gibt es im Objektmodell nur eine Klasse `Parameter`, welche in einer *1:n*-Beziehung mit der Klasse `Data Object` steht. Die Tatsache, daß nur ein Parameter einer Workflow-Instanz eine solche Beziehung eingehen kann, muß durch eine entsprechende Bedingung zum Ausdruck gebracht werden.

3.2 Design-Objektmodell

Aufgabe des Design-Objektmodells ist es, eine implementierungsnähere Darstellung des Systems zu geben. Daher wird hier insbesondere auf Attribute, Methoden und die Beziehungen der verschiedenen Klassen aus implementierungstechnischer Sicht eingegangen. Das Design-Objektmodell ist in Abbildung 4 dargestellt.

Beim Übergang von der problemorientierten Modellierung (Analyse-Objektmodell) zur implementierungsorientierten Modellierung (Design-Objektmodell) kann es neben Verfeinerungen auch zu einer Umstrukturierung des Modells kommen. Dies tritt immer dann ein, wenn es aus der Sicht der Anwendung notwendig ist, bestimmte Entitäten explizit zu modellieren, während dies aus implementierungtechnischer Sicht nicht notwendig oder sinnvoll ist. Eine solche Situation tritt bei unserer Anwendung u.a. bei der Modellierung von Kontrollkonnektoren auf. Zwar ist deren explizite Modellierung aus Anwendungssicht sinnvoll, intern können sie aber leicht durch Datenkonnektoren realisiert werden, wie nun erklärt wird.

Ein Kontrollkonnektor kann zu jedem Zeitpunkt einen der Zustände "wahr-signalisiert", "falsch-signalisiert" und "nicht-signalisiert" einnehmen. Diese Zustände lassen sich durch eine Variable des Datentyps `Boolean` im Workflow-System (mit einem um den Wert `NULL` erweiterten Wertebereich, s.u.) modellieren. Wird ein Kontrollkonnektor von einem Workflow-Objekt i nach einem Workflow-Objekt j erstellt, so übergibt i eine als Kontrollkonnektor gekennzeichnete Variable an das Workflow-Objekt j. Wird der Wert "wahr-signalisiert" übergeben, so kann u.U. die Startbedingung ausgewertet und ggf. die Ausführung des Workflows an dieser Stelle fortgesetzt werden.

Der Vorteil eines solchen Umgangs mit Kontrollkonnektoren liegt nicht nur in der Einsparung einer (redundanten) Klasse, sondern auch in der Vereinfachung der Modellierung der Startbedingung ohne Einschränkung der Ausdrucksfähigkeit. Es muß nun nicht mehr zwischen Startbedingungen auf Variablen und Startbedingungen auf Kontrollkonnektoren unterschieden werden, und Startbedingungen können unter Verwendung von traditionellen im Rahmen des Datenflusses modellierten Parametern sowie von Informationen über den Kontrollfluß des Workflows spezifiziert und ausgewertet werden. In Abbildung 1 sind Kontrollfluß-relevante Parameter im Unterschied zu ausschließlich Datenfluß-relevanten grau hinterlegt.

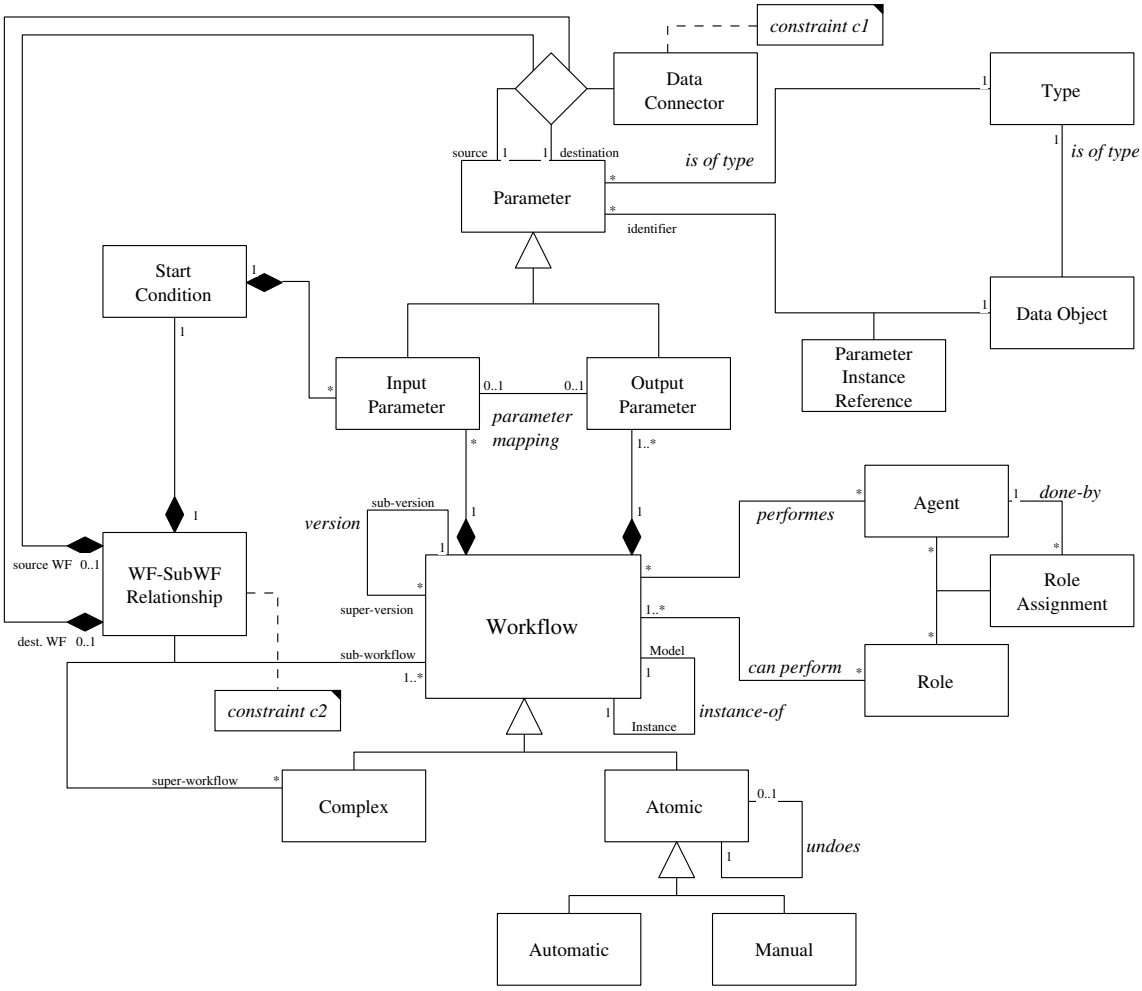

Abbildung 4: Design-Objektmodell

3.2.1 Die Klassen des Modells

Die Klasse `Workflow` ist eine abstrakte Klasse, die – wie oben beschrieben – sowohl Workflow-Modelle als auch Workflow-Instanzen darstellen kann. Zum Erzeugen eines Objekts stehen zwei Konstruktoren zur Verfügung: `Workflow()` erstellt ein einzelnes Workflow-Objekt; `Workflow(originalmodel:Workflow)` erstellt eine Workflow-Instanz, deren Struktur von `originalmodel` definiert wird. Dabei wird der Toplevel-Workflow des Originals kopiert, und die Kopie wird in die `instance of`-Beziehung zwischen Workflow-Objekten aufgenommen.

Bei Workflow-Modellen, die direkt von einem Benutzer aus gestartet werden können, ist das `Executable`-Flag auf wahr gesetzt. Die anderen Workflows können nur von ihren jeweiligen Super-Workflows aus gestartet werden. Der Zustand eines Workflows wird im Attribut `status` verwaltet. Durch die Methoden `start()`, `reset()`, `undo()` und `skip()` kann die Ausführung von Workflow-Instanzen durch autorisierte Benutzer im Rahmen von Benutzerinteraktionen [29] gesteuert werden. Workflow-Modelle stehen in unmittelbarer Beziehung zu den Klassen `Role` und `Agent`. Während die Beziehung `can perform` mögliche Zuordnungen zwischen Rollen und Workflow-Objekten ausdrückt, gibt die Beziehung `performs` an, welcher Agent einen Workflow tatsächlich bearbeitet hat. Während Rolleninformationen insbesondere für Workflow-Modelle von Bedeutung sind, werden Informationen über konkrete Ausführungen von Workflow-Instanzen durch die `performs`-Beziehung zwischen Workflow-Objekten und Agenten modelliert.

Um die Beziehungen unterschiedlicher Workflows verwalten zu können, stehen Workflows über `version` zueinander in Beziehung. Eine solche Beziehung wird instantiiert, wenn ein Workflow durch Änderungen aus einem zweiten Workflow hervorgegangen ist. Auf diese Weise wird eine Versionierung von Workflows ermöglicht, die insbesondere zur Flexibilisierung von Workflow-Management-Systemen von großer Bedeutung ist [27].

Wie oben beschrieben, werden Kontrollflußbeziehungen von Workflows durch Datenfluß implementiert. Daher haben Workflows stets einen Ausgabeparameter `succ_termination` vom Typ `Boolean`, der angibt, ob ein Workflow erfolgreich abgeschlossen wurde. Der Wertebereich dieses Datentyps wird um "NULL" erweitert, um die möglichen Werte *true signaled* (wahr-signalisiert), *false signaled* (falsch-signalisiert) sowie *not signaled* (nicht-signalisiert) abbilden zu können. Dieser Ausgabeparameter kann dann von den folgenden Workflows in deren Startbedingungen verwendet werden.

Ein Objekt der Klasse `WF-SubWF Relationship`, welches in Beziehung zu jeweils einem Objekt der Klassen `Complex` und `Workflow` steht, drückt die Beziehung eines Workflows zu einem komplexen Workflow im Workflow-Modell aus. Alle Parameter einer Workflow-Instanz werden durch die abstrakte Klasse `Parameter` abgebildet. Parameter haben einen eindeutigen Typ und stehen in Beziehung zu Datenobjekten, die Parameterinhalte darstellen. Es ist nur eine Beziehung zu Datenobjekten des geeigneten Datentyps erlaubt. Dies bedeutet, daß beide Parameter die gleiche Referenz auf ein Objekt der Klasse `Type` besitzen. Jeder Parameter wird bei seiner Instantiierung mit `NULL` belegt.

Weil Datenkonnektoren in unserem Modell zusätzlich die Funktion von Kontrollkonnektoren übernehmen, wird das Boolesche Attribut `is_controlconnector` für Datenkonnektoren eingeführt. Von der Klasse `Parameter` leiten sich die Klassen `Input Parameter` und `Output Parameter` ab, die Beziehungen zu dem Workflow-Modell haben. Zur eindeutigen Identifizierung von Eingabe- und

Ausgabeparametern im Kontext eines komplexen Workflow-Modells haben diese das Attribut `name`. Die Klasse `Input Parameter` hat zusätzlich zu der Beziehung zur Klasse `Workflow` auch eine Beziehung zur Klasse `Start Condition`. Über diese Beziehung wird festgelegt, welche Kontrollparameter (d.h., Kontrollkonnektoren) die Startbedingung zusätzlich zu den Input-Parametern des Workflows auswerten muß. Diese Beziehung wird beim Anlegen eines Kontrollkonnektors automatisch erzeugt.

Um auszudrücken, daß ein Datenobjekt eine Workflow-Instanz durchläuft, d.h., von einer Workflow-Instanz im Rahmen von Datenfluß erzeugt oder verändert wird und anschließend an Folge-Aktivitäten weitergeleitet wird, verwendet man die Beziehung `Parameter Mapping`. Ist ein Eingabeparameter mit einem Ausgabeparameter über diese Relation verbunden, so wird ein Datenobjekt von einer Aktivität sowohl als Eingabe- als auch als Ausgabe-Objekt verwendet.

Wir weisen darauf hin, daß im Design-Objektmodell horizontale und vertikale Datenkonnektoren durch eine Beziehung repräsentiert sind, an der zwei Parameter-Objekte und ein bzw. zwei WF-Sub-Workflow-Relationship-Objekte beteiligt sind. Handelt es sich um einen vertikalen Datenfluß, so ist genau eines der zuletzt genannten Objekte beteiligt, während beim horizontalen Datenfluß zwei Objekte der Klasse WF-Sub-Workflow Relationship in Beziehung stehen. Der in der Abbildung 4 dargestellte *constraint c1* ergibt sich aus den oben diskutierten *constraints c1a* und *c1b*. Die `Start Condition` entscheidet nach der Prüfung der in den Workflow eingehenden Parameter und der speziellen, von der Startbedingung verwendeten Kontrollparameter über den Start des Workflows. Die Auswertung der Parameter erfolgt durch die Methode `evaluate`. Spezielle Startbedingungen können durch Spezialisierungen der Klasse `Start Condition` erzeugt werden. In diesen Klassen müssen die `evaluate`-Methoden geeignet überschrieben werden. Für allgemein formulierte Bedingungen ist eine generische Sub-Klasse `Generic Start Condition` zu definieren, die Ausdrücke, wie etwa '`(NOT a) AND (x > 5)`' geeignet auswertet.

Rollen werden durch die Klasse `Role` abgebildet und stehen in Beziehung zu den Klassen `Workflow` und `Agent`. Rollen sind eindeutig benannt durch das Attribut `name`. Ein Agent hat die Attribute `name`, `password` und `logged_in:Boolean`. Hinzu kommen die Methoden `login()` und `logout()` zum ein- bzw. ausloggen und die Methode `get_worklist()`, mit der eine Arbeitsliste für einen Agenten aufgestellt werden kann. Zu beachten ist hierbei, daß diese Arbeitsliste nicht über die Beziehung `performs` hergestellt wird, sondern über ein Traversieren der Kanten über die Klassen `Role` und `Workflow`. Die Beziehung `performs` dient nur der Verwaltung von Workflow-Ausführungsdaten. Diese Beziehung bleibt auch nach Beendigung der Workflow-Instanz erhalten und erlaubt so nachfolgende Anfragen an bereits vollendete Workflow-Instanzen.

Durch die Klasse `Role Assignment` werden Agenten und Rollen einander zugeordnet. Für die Historie von Bedeutung sind die Attribute `valid_since:Date` und `valid_until:Date`. Über die Beziehung `done-by` wird festgehalten, welcher Agent (mit administrativen Rechten) einen anderen Agenten in eine Rolle aufgenommen hat. Jedes Objekt der Klasse `Type` stellt einen Datentypen innerhalb des Workflow-Systems dar. Datentypen sind eindeutig über das Attribut `name` benannt.

Anwendungsdaten innerhalb des Workflow-Systems werden in Objekten der Klasse `Data Object` gespeichert und über eine lineare Versionierung verwaltet. Die Datenwerte werden in dem Attribut `data:List(Any)` abgelegt. Die aktuelle Versionsnummer liegt im Attribut `current_version`. Über die Methoden `get_data()` und `put_data(data:Any)` kann die aktuelle Version gelesen bzw. geändert werden. Zur Erstellung einer neuen Version verwendet man die Methode `create_new_version()`,

die die aktuelle Versionsnummer erhöht und die Daten der neuen Version mit den Daten der zuletzt gültigen instantiiert. Durch diese Versionierung von Daten kann ein Workflow im Nachhinein nachvollzogen werden, und es ist möglich, Daten vorhergehender Workflow-Ausführungen zur Wiederholung von Workflows zu verwenden. Die Klasse `Parameter Instance Reference` schafft die Verbindung zwischen den Parametern eines Workflow-Modells und den dazugehörigen Daten. Das Attribut `version` ist dabei ein Verweis auf die von der Instanz verwendeten oder manipulierten Versionen der Daten. Damit kann für jede Workflow-Instanz auch im Nachhinein bestimmt werden, welche Daten als Eingabe- und welche als Ausgabeparameter verwendet wurden.

3.2.2 Beziehung zu spezifizierten Corba-Diensten

In diesem Abschnitt wollen wir kurz diskutieren, welche spezifizierten Corba-Dienste [17] zur Implementierung des zu entwickelnden Systems sinnvoll eingesetzt werden können. Um die Präsentation des Objektmodells übersichtlich zu gestalten, wurde auf eine graphische Darstellung bestehender und zu verwendender Corba-Dienste in dem Objektmodell verzichtet. Stattdessen wollen wir nun kurz die wesentlichen Corba-Services und deren mögliche Verwendung im Rahmen unseres Systems skizzieren.

Der Corba *Lifecycle Service* stellt Methoden und Klassen zur Verfügung, um Objekte dynamisch zu erzeugen, zu löschen, zu kopieren und zu verschieben. Die Implementierung der oben beschriebenen Klassen kann durch Verwendung dieses Dienstes erleichtert werden, da unsere Anwendung höchst verteilt ist und zudem dynamische Veränderungen von Strukturinformationen erlauben soll. Von besonderer Bedeutung ist in diesem Zusammenhang eine effiziente Implementierung einer Methode zum Verschieben von Objekten zwischen Rechnersystemen, um eine ausfallsichere und performante verteilte Ausführung von Workflows zu gewährleisten.

Die Aufgabe des *Persistency Service* ist die dauerhafte und transparente Speicherung von Corba-Objekten. Da Workflow-Anwendungen nach dem Verlust des Hauptspeicherinhaltes und dem Wiederanlaufen des Systems eine Situation vorfinden sollen, in der sie sich unmittelbar vor dem Ausfall befanden, ist Persistenz eine wichtige Eigenschaft von Workflow-Systemen. Leider ist dieser Dienst bisher in keinem ORB vollständig implementiert. Darüber hinaus befindet sich die Spezifikation dieses Dienstes derzeit in einer Überarbeitungsphase, so daß auch in näherer Zukunft nicht mit seiner Verfügbarkeit gerechnet werden kann und daher entsprechende Eigenentwicklungen notwendig werden.

Der *Relationship Service* stellt Klassen zur Verfügung, mit denen Objekte zueinander in Beziehung gesetzt werden können, ähnlich wie bei der objektorientierten Modellierung. Dieser Dienst kann verwendet werden, um die im Objektmodell spezifizierten Beziehungen zur Laufzeit des Systems verwalten zu lassen. In Anbetracht der oben beschriebenen komplexen Beziehungsstrukturen kommt diesem Dienst eine wichtige Bedeutung bei der Implementierung unseres Systems zu.

Der *Naming Service* und der *Event Service* stellen keine zentrale Komponente des Workflow-Systems dar; sie können aber zur Vereinfachung spezieller Aufgaben herangezogen werden. Der Event Service dient einem klaren und effizienten Nachrichtenaustausch zwischen Corba-Objekten und kann für eine effizientes Monitoring des Systems sowie für die Kommunikation von Workflow-Objekten mit Benutzern des Systems eingesetzt werden. Der Naming Service ermöglicht ein einfaches Auffinden von Objekten anhand eines strukturierten Namens. Der *Trading Object Service*

ermöglicht eine späte Bindung von Klienten an Server. Er zeichnet sich insbesondere dadurch aus, daß ein Klient einem *Trading Object Server* zur Laufzeit Kriterien angeben kann, die der gesuchte Server erfüllen muß. Dies ist besonders in verteilten Umgebungen von Vorteil. In unserem Fall erscheint es sinnvoll, die Factory-Objekte des Lifecycle Service auf diese Weise zu ermitteln. In diesem Zusammenhang können auch Anteile der Rollenauflösung fallen, z.B. das Ermitteln eines Rechnersystems zur Ausführung automatischer Workflows.

3.2.3 Ausführungsalternativen

Im folgenden werden kurz verteilte Laufzeitarchitekturen für Workflow-Management-Systeme vorgestellt, die sich in ihrem Zentralisierungsgrad unterscheiden. Es werden zentrale, hierarchische und verteilte Ausführungen betrachtet.

Werden alle Workflows eines Systems von einem zentralen Knoten aus kontrolliert, so spricht man von einer zentralen Ausführung. Diesen Ansatz verfolgen die meisten derzeit kommerziell verfügbaren Workflow-Management-Systeme, etwa [8]. Gegen eine solche Laufzeitarchitektur spricht zunächst die inhärent dezentrale Struktur ausgeführter Workflows. So werden zwei durch Daten- oder Kontrollflußbeziehungen miteinander verbundene Workflows oft durch unterschiedliche Personen oder unterschiedliche Rechner eines verteilten Rechnersystems ausgeführt. Hinzu kommen Gründe der Ausfallsicherheit: Wird ein Workflow von einer zentralen Workflow-Engine gesteuert, führt ein Ausfall dieses Knotens zu einer Blockierung aller Workflows des Systems. Wie häufig, wenn zentrale Lösungen für verteilte Probleme herangezogen werden, kann der zentrale Knoten auch zum Engpaß des Systems werden. Allerdings läßt sich der Ablauf der Workflows im Rahmen des Workflow-Monitoring von einem zentralen Knoten aus gut überwachen, da alle relevanten Informationen dort vorliegen. Für diese Aufgabe wird bei einer dezentralen Architektur zusätzliche Kommunikation nötig.

Durch eine dezentrale und an die Struktur von Workflows angepaßte Laufzeitarchitektur können die oben genannten Probleme teilweise umgangen werden. Einen ersten Ansatz stellt eine hierarchische Laufzeitarchitektur dar. Hierbei übernimmt jeder komplexe Workflow die Steuerung des Kontroll- und Datenflusses seiner Sub-Workflows. Die traditionell von Workflow-Management-Systemen erbrachte Funktionalität wird also von Workflow-Instanzen erbracht. Da sich Workflows im allgemeinen über mehrere Ebenen erstrecken, erfolgt die Kontrolle der Workflows hierarchisch. Ein Engpaß wie bei einer zentralen Architektur ist bei dieser Architektur eher unwahrscheinlich, da jeder Workflow nur mit seinen direkten Sub-Workflows unmittelbar kommuniziert. Kommt es zu einem Ausfall auf der Ebene eines komplexen Workflows, sind über- und nebengeordnete Workflows von diesem Ausfall nicht betroffen und können die Ausführung fortsetzen.

In der nächsten Verteilungsstufe erzeugt und startet ein Workflow lediglich diejenigen Sub-Workflows, die keine eingehenden Kontrollkonnektoren besitzen. Alle weiteren Sub-Workflows werden von den aktivierten Sub-Workflows instantiiert und gestartet. Damit erfolgt nicht nur die Workflow-Ausführung verteilt, sondern auch die Kontrolle der Ausführung. Insbesondere wird der Kontroll- und Datenfluß nun von den Sub-Workflows selbst gesteuert. Bei dieser Architektur entsprechen die Kommunikationswege unmittelbar den spezifizierten Kontroll- und Datenflüssen. Es gibt daher keine Umwege über zentrale Knoten und somit auch keine Engpässe. Ebenso ist bei diesem Ansatz die Ausfallsicherheit hoch, da selbst auf Ebene eines komplexen Workflows einzelne Teile unabhängig

voneinander ablaufen können. Zum Zwecke des Monitoring müssen bei der verteilten wie auch bei der hierarchischen Ausführung alle Instanzen mit einem zentralen Monitoring-Objekt kommunizieren.

4 Zustände und Zustandsübergänge

Um die dynamischen Aspekte eines Systems möglichst vollständig zu beschreiben, ist eine explizite Modellierung von Zuständen und Zustandsübergängen angemessen; dies bezieht sich in unserer Anwendung im wesentlichen auf Workflow-Instanzen. Im allgemeinen besitzen Objekte einen Zustand und kommunizieren über Nachrichten miteinander, und ein Objekt reagiert auf Ereignisse in Abhängigkeit von seinem Zustand. In diesem Abschnitt beschreiben wir die Zustände und Zustandsübergänge, die ein Workflow-Instanz-Objekt innerhalb seines Lebenszyklus besitzt.

Der Zustand einer Workflow-Instanz kann im Rahmen der kontrollierten Ausführung von dem Workflow-Management-System sowie durch aufgerufene Applikationsprogramme oder durch Benutzerinteraktionen geändert werden. Workflow-Instanzen wurden in Abschnitt 3.1 beschrieben; sie existieren in verschiedenen Typen, die sich auch in ihrem Verhalten, d.h. in den Zuständen und Zustandsübergängen, unterscheiden. Zur Darstellung geben wir Zustandsübergangsdiagramme an, in denen Zustände durch beschriftete Rechtecke und Zustandsübergänge durch markierte Pfeile dargestellt werden (siehe Abbildung 5). Zustände können geschachtelt sein; sie können dann in der Punkt-Schreibweise eindeutig angegeben werden. So kann z.B. der Sub-Zustand *running* im Zustand *open* durch *open.running* eindeutig spezifiziert werden. Wenn die Bezeichnung der Unterzustände eindeutig ist, geben wir nur den einfachen Namen des Zustands an.

Der allgemeine Ablauf von Workflow-Instanzen wird in Abbildung 5 dargestellt; dieses generische Zustandsübergangsdiagramm beschreibt insbesondere das Verhalten von komplexen und atomaren manuellen Workflow-Instanzen. Abbildung 6 zeigt ein entsprechend verfeinertes Zustandsübergangsdiagramm für atomare automatische Workflow-Instanzen.

4.1 Generisches Zustandsübergangsdiagramm

Das in Abbildung 5 dargestellte generische Zustandsübergangsdiagramm wird als Grundlage für die weitere Diskussion verwendet. Wird eine Workflow-Instanz kreiert, so nimmt sie den Zustand *open* an, den sie erst bei ihrer Beendigung verläßt. Einer der Sub-Zustände ist *notRunning*, in dem sich eine Workflow-Instanz befindet, wenn sie nicht an der Ausführung eines Workflows beteiligt ist. Das bedeutet, daß sie entweder noch nicht gestartet worden ist (Zustand *notStarted*) oder derzeit unterbrochen ist (*suspended*).

Der Zustand *init* stellt den Startzustand von Workflow-Instanzen dar. Dies wird in einem Zustandsübergangsdiagramm durch einen Pfeil mit einem Kreis am Ausgangspunkt dargestellt. In dem *init*-Zustand werden die Vorbereitungen für die Ausführung der Workflow-Instanz getroffen. Sind die Vorbereitungen abgeschlossen, so kann durch den *getReady*-Übergang zum Zustand *ready* gewechselt werden. Durch die *disable* bzw. *enable*-Übergänge können Workflow-Instanzen temporär deaktiviert werden. Von *getReady* aus kann die Workflow-Instanz aktiviert werden, wobei der *activate*-Übergang zum Zustand *running* vollzogen wird. Der Zustand *suspended* kann vom *running*-Zustand aus betreten werden. Er drückt aus, daß die betreffende Workflow-Instanz unterbrochen worden ist. Der Zustand kann durch den *resume*-Übergang zum Zustand *open.running* wieder verlassen werden. In

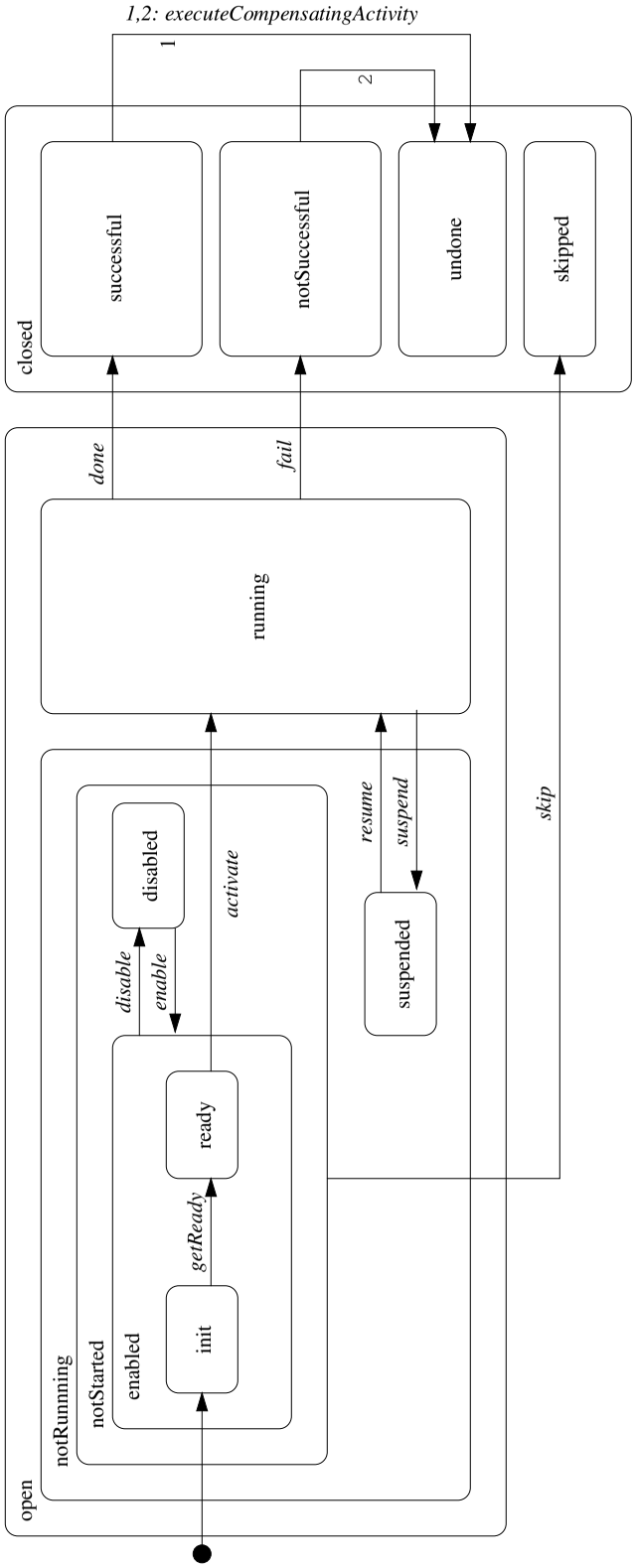

Abbildung 5: Generisches Zustandsübergangsdiagramm

dem Zustand *open.running* wird die eigentliche Arbeit der Workflow-Instanz verrichtet. Von diesem Zustand kann zu *closed* gewechselt werden. Letzteres kann durch die Zustandsübergänge *done* bzw. *fail* erfolgen, was der erfolgreichen bzw. nicht erfolgreichen Ausführung der Workflow-Instanz entspricht.

Beendete Workflow-Instanzen befinden sich im *closed*-Zustand. Befindet sich eine Workflow-Instanz einmal in diesem Zustand, so verläßt sie ihn nicht wieder. Dieser Zustand beinhaltet mehrere Sub-Zustände, die wir jetzt näher beschreiben wollen. Bei einer erfolgreichen Beendigung der Aktivität wird der Zustand *successful* betreten. Scheitert die Ausführung der Aktivität, so wird der Zustand *notSuccessful* betreten. Diese beiden Zustände können durch den *executeCompensating-Activity*-Übergang zum Zustand *undone* wieder verlassen werden. Wir weisen darauf hin, daß die Möglichkeit der Kompensation von Workflows nicht generell gegeben ist, sondern von der Verfügbarkeit entsprechender kompensierender Workflows abhängig ist, die den bereits ausgeführten Workflow semantisch ungeschehen machen (semantic undo). Workflows können auch übersprungen werden, was durch den Zustand *skipped* ausgedrückt wird. Eine solche Situation fällt in den Bereich der dynamischen Modifikation [29]. Eine Workflow-Instanz kann nur dann übersprungen werden, wenn sie noch nicht gestartet wurde, d.h., solange sie sich in einem *notStarted*-Zustand befindet.

4.2 Zustände von komplexen Workflow-Instanzen

Das Zustandsübergangsdiagramm einer komplexen Workflow-Instanz unterscheidet sich zunächst nicht von dem generischen Diagramm aus Abbildung 5. Allerdings gibt es einige semantische Unterschiede, die im folgenden erklärt werden.

Im Zustand *init* wird die Initialisierung der Eingabeparameter des komplexen Workflows sowie die Rollenauflösung durchgeführt. Hierbei ist zu beachten, daß einem komplexen Workflow ein *verantwortlicher* Agent zugeordnet wird. Diese Rollenauflösung hat somit eine andere als die oben beschriebene traditionelle Bedeutung. Außerdem werden i.a. weitere Vorbereitungen getroffen, die zur Ausführung der Sub-Workflow-Instanzen notwendig sind, etwa die Ermittlung der Workflows ohne eingehende Kontrollkonnektoren. Sind die Vorbereitungen abgeschlossen, so wird der Zustand *ready* angenommen. Nun kann der komplexe Workflow aktiviert werden, d.h., der Übergang nach *running* kann stattfinden.

Nun werden die zu startenden Sub-Workflow-Instanzen erzeugt und initialisiert. Welche Aktionen dabei im einzelnen durchgeführt werden, hängt von der gewählten Ausführungsalternative (zentral, hierarchisch, verteilt) ab. Bei verteilten Ausführungen werden die Sub-Workflows ohne eingehende Kontrollkonnektoren erzeugt. Wechselt ein komplexer Workflow von *running* in den Zustand *suspended*, so kann dies für die Sub-Workflows je nach Implementierung unterschiedliche Auswirkungen haben:

- Alle Sub-Workflows gehen in den Zustand *suspended* über, werden also ebenfalls unterbrochen. Dabei ist zu beachten, daß der Workflow komplett angehalten wird, was u.U. mit einem hohe Aufwand verbunden ist, da das Stoppen durch alle Ebenen der Workflow-Hierarchie "weitergereicht" werden muß.

- Die Sub-Workflows laufen zunächst unverändert weiter, allerdings werden nach deren Beendigung keine weiteren Sub-Workflows gestartet. Dies ist insbesonders dann einfach zu reali-

sieren, wenn Super-Workflows ihre eigenen Sub-Workflows selbst erzeugen und den Kontroll- und Datenfluß steuern, also bei der hierarchischen Ausführungsalternative.

- Die aktiven Sub-Workflows laufen weiter. Sind alle Sub-Workflows beendet, bleibt der Super-Workflow so lange unvollendet, wie er sich im *suspended*-Zustand befindet. In diesem Fall wird lediglich die Beendigung des komplexen Workflows ausgesetzt, die Sub-Workflows können ihre Tätigkeit ungehindert fortsetzen und beenden.

4.3 Zustände von manuellen Workflow-Instanzen

Bei manuellen Workflow-Instanzen handelt es sich um Aktivitäten, die typischerweise auf Arbeitslisten von Agenten erscheinen und von diesen ausgeführt werden können. Das zugehörige Zustandsübergangsdiagramm unterscheidet sich nicht von der in Abbildung 5 gezeigten generischen Variante; die Bedeutung des Diagramms bei manuellen Workflow-Instanzen wird im folgenden beschrieben.

Die vom allgemeinen Modell abweichende Bedeutung des Zustands *enabled* wird im Zusammenhang seiner Sub-Zustände *init* und *ready* deutlich. Der Zustand *init* stellt den Startzustand einer Workflow-Instanz dar. In dem *init*-Zustand werden die Vorbereitungen für die Ausführung der Workflow-Instanz getroffen. Dies ist bei manuellen Workflows die Rollenauflösung und das Einlesen der Input-Parameter. Ist die Initialisierung der Workflow-Instanz abgeschlossen, so wird vom *init*-Zustand in den *ready*-Zustand übergegangen (*getReady*). Dies entspricht der Situation, in der die Aktivität bei den entsprechenden Agenten auf der Arbeitsliste erscheint. Der *ready*-Zustand kann durch den *activate*-Übergang von *ready* nach *running* verlassen werden. Dieser Zustandsübergang wird dann ausgelöst, wenn der Agent die Aktivität aus seiner Arbeitsliste auswählt, um sie zu bearbeiten.

4.4 Zustände von automatischen Workflow-Instanzen

Bei einer automatischen Workflow-Instanz handelt es sich um eine atomare Aktivität, die im allgemeinen von einem Anwendungsprogramm ausgeführt wird. Abbildung 6 zeigt das zugehörige Zustandsübergangsdiagramm. Diejenigen Zustände und Übergänge, die sich von dem allgemeinen Modell aus Abschnitt 4.1 unterscheiden, werden im folgenden beschrieben.

Der Zustand *init* stellt den Startzustand einer automatischen Workflow-Instanz dar. In diesem Zustand werden die Vorbereitungen für die Ausführung der Workflow-Instanz getroffen. Dies beschränkt sich auf das Einlesen der Input-Daten. Der Zustand *suspended* kann von allen Sub-Zuständen von *running* aus betreten werden und drückt aus, daß die betreffende Workflow-Instanz unterbrochen worden ist; bei einer automatischen Workflow-Instanz wird dann das laufende Programm unterbrochen. Der Zustand kann wieder (nach *running*) verlassen werden, wobei das Programm fortgesetzt wird. Im Zustand *running* wird die eigentliche Aktivität der Workflow-Instanz ausgeführt. Hierbei muß noch zwischen den unterschiedlichen Arten von automatischen Workflow-Instanzen unterschieden werden. Im allgemeinen wird unterschieden zwischen transaktionalen und nicht-transaktionalen automatischen Workflow-Instanzen. Falls das ausführende Software-System transaktionale Zusicherungen über die Auführung liefern kann, so handelt es sich um transaktionale automatische Workflows, sonst um nicht-transaktionale. Beispiele für transaktionale Applikationen

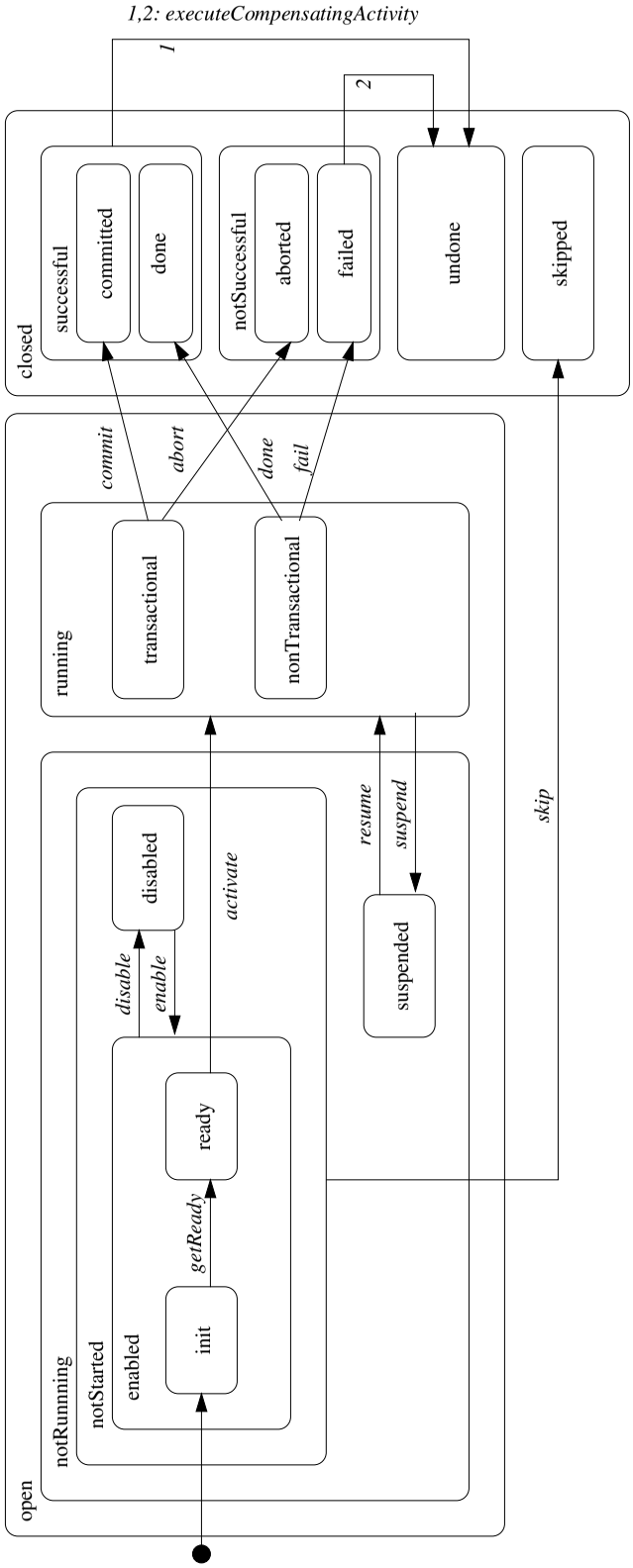

Abbildung 6: Zustandsübergangsdiagramm von atomaren, automatischen Workflow-Instanzen

sind etwa Datenbank-Programme; bei einem Zugriff auf einen Web-Server handelt es sich demgegenüber um eine Applikation, die keine Zusicherungen über ihre Ausführung machen kann; der entsprechende automatische Workflow ist daher nicht-transaktional. Diese Eigenschaften von Applikationsprogrammen werden durch die Zustände *transactional* bzw. *nonTransactional* repräsentiert.

Der Zustand *transactional* kann auf zwei Arten verlassen werden: entweder mit *commit* zum Zustand *committed* oder mit *abort* zum Zustand *aborted*, was einer erfolgreichen bzw. nicht erfolgreichen Ausführung eines transaktionalen Workflows entspricht. Hierbei wird durch das ausführende Software-System (z.B. ein Datenbanksystem) zugesichert, daß es keine Auswirkungen auf den Datenbestand gibt. Daher ist hier auch kein "undo" im Sinne einer kompensierenden Aktivität nötig. Vom Zustand *nonTransactional* steht der *done*-Übergang zum Zustand *done* für die erfolgreiche Ausführung der Aktivität und der *fail*-Übergang zum Zustand *failed* bei einer gescheiterten Ausführung der Aktivität bereit. Die Zustände *committed* und *done* werden zu *successful*, die Zustände *aborted* und *failed* werden zu *notSuccessful* zusammengefaßt.

4.5 Weitere Überlegungen zu den Benutzerinteraktionen

Bei der Bearbeitung einer atomaren Workflow-Instanz hat der die Aktivität ausführende Agent die Möglichkeit, in den Ablauf dynamisch einzugreifen. Hierbei können die Operationen *skip*, *stop* und *repeat* verwendet werden. Durch die Benutzung eines Monitoring-Tools zur Überwachung des Fortschritts von Workflow-Instanzen kann neben weiteren dynamischen Eingriffen in den Ablauf von atomaren Workflow-Instanzen auch der Ablauf von komplexen Workflows beeinflußt werden.

- *Skip:* Eine Aktivität kann nur dann übersprungen werden, wenn dies bei der Modellierung angegeben wurde (Aktivitäten mit dieser Eigenschaft werden als *skippable* bezeichnet). Will ein Agent eine Aktivität überspringen, so ist dies solange möglich, wie sich diese Aktivität im Zustand *notStarted* befindet.

- *Undo:* Eine Aktivität kann nur dann zurückgenommen werden, wenn dies bei der Modellierung angegeben wurde (*undoable*) und wenn ein entsprechender Workflow zur Kompensation bereitsteht. Diese Aktion kann nur dann durchgeführt werden, wenn sich die Instanz im *closed*-Zustand befindet. Wenn eine Workflow-Instanz zurückgenommen werden soll, so wird eine kompensierende Aktivität gestartet, die einer eigenständigen Workflow-Instanz entspricht.

- *Repeat:* Eine Aktivität kann nur wiederholt werden, wenn sie *undoable* ist. Wenn eine Aktivität z.B. den Datenbestand geändert hat, so muß dieser wieder in den originalen Zustand versetzt werden. Dies geschieht, wie bereits oben erwähnt, durch eine kompensierende Aktivität. Hat eine Aktivität keine Daten verändert, sondern z.B. nur eine Berechnung oder Anzeige durchgeführt, so entspricht ihre kompensierende Aktivität einer *leeren Operation*. Somit können auch letztgenannte Aktivitäten wiederholt werden. Eine Wiederholung ist eine zusammengesetzte Aktivität aus einem *undo* und einem anschließenden Starten einer neuen Workflow-Instanz auf der Grundlage der vormals benutzten Daten.

- *Stop:* Beim Stoppen einer Aktivität handelt es sich um vorzeitiges Beenden einer Aktivität, ohne auf deren Beendigung zu warten (bei automatischen Aktivitäten) bzw. ohne Beendigung der Ausführung (bei manuellen Aktivitäten). Somit ist die Aktivität auf jeden Fall als

gescheitert zu betrachten, d.h., der Endzustand ist *notSuccessful*. Die Stop-Aktion entspricht dem *fail-* bzw. *abort-* Zustandsübergang.

- *Disable:* Jede Aktivität kann deaktiviert werden, wenn sich diese Aktivität im Zustand *enabled* befindet. Dieser Zustand beinhaltet, daß die Aktivität noch nicht gestartet worden ist. Bei komplexen und automatischen Aktivitäten wird die Initialisierungsphase unterbrochen, soweit dies möglich ist. Anderenfalls wird zumindest eine Aktivierung (*activate*) verhindert. Bei einer manuellen Workflow-Instanz, die sich bereits im Zustand *ready* befindet, d.h., sie ist auf den Arbeitslisten der Agenten erschienen, gibt es mehrere Alternativen:
  - Die Aktivität wird von der Arbeitsliste entfernt. Hierbei könnte für die Agenten allerdings der Eindruck entstehen, daß die Aktivität durch einen anderen Agenten bearbeitet wird.
  - Die Aktivität wird optisch als *disabled* gekennzeichnet. Nachteilig ist hierbei vielleicht, daß die Arbeitsliste u.U. unübersichtlich wird, wenn zuviele Instanzen *disabled* wurden.
  - Soll eine Aktivität deaktiviert werden, so kann bzw. muß zusätzlich angegeben werden, ob sie auf den Arbeitslisten bleiben soll oder nicht.

4.6 Implementierung von Zuständen und Zustandsübergängen

Der Zustand einer Workflow-Instanz wird durch den Wert eines Attribut der Klasse Workflow beschrieben (siehe Abschnitt 3.1). Der Zustand einer Workflow-Instanz kann durch bestimmte Operationen abgefragt und gesetzt werden.

Für Zustandsübergänge gibt es mehrere Implementierungsalternativen. Sie können beispielsweise als Operationen implementiert werden, die durch Ereignisse getriggert werden. Eine solche Operation überprüft den Zustand der Workflow-Instanz und entscheidet, ob ein Zustandsübergang erlaubt ist oder nicht. Hier sei zunächst darauf hingewiesen, daß einem Zustandsübergang nicht notwendigerweise genau ein Ereignis entspricht. Beispiele hierfür werden später genannt.

Eine elegantere Möglichkeit wäre die Implementierung einer Operation `transition`, die als Eingabeparameter ein Ereignis oder einen Zustandsübergang erhält und dann anhand von Regeln überprüft, ob der Zustand geändert werden kann oder nicht. Anschließend wird eine Boolesche Variable zurückgegeben, welche ausdrückt, ob der Übergang durchgeführt werden konnte. In diesem Punkt wird die Motivation für die Benutzung von Zuständen deutlich. Es wird hierdurch nämlich ermöglicht, nur in bestimmten Zuständen der Workflow-Instanz bestimmte Operationen zuzulassen. Insbesondere im Hinblick auf Benutzereingriffe in den Ablauf eines Workflows bieten Zustände besondere Vorteile, da bestimmte Eingriffe (z.B. *skip*) nur in bestimmten Zuständen (hier *notStarted*) zugelassen werden. Auch im Hinblick auf dynamische Veränderungen spielen Zustände eine wichtige Rolle. So kann beispielweise eine Veränderung des Kontrollflusses *nach* einer bestimmten Aktivität nur solange erfolgen, wie sich die entsprechende Workflow-Instanz in einem *open*-Zustand befindet. Befindet sie sich z.B. im *running*-Zustand, so muß sie zunächst ausgesetzt (*suspended*) werden. Weiterhin sind Veränderungen *vor* einer Aktivität nur dann möglich, wenn diese noch nicht gestartet worden ist. Bevor eine Veränderung jetzt durchgeführt wird, muß die Instanz deaktiviert (*disabled*) werden.

Wie bereits oben erwähnt, muß zwischen den Zustandsübergängen und den zugrundeliegenden Ereignissen unterschieden werden. Oftmals ist einem Ereignis genau ein Übergang zugeordnet, so

daß man diese Trennung nicht für nötig halten könnte (z.B. entspricht das Ereignis *activate* dem gleichnamigen Zustandsübergang); beispielhaft seien einige Ausnahmen erwähnt.

- Dem Zustandsübergang *fail* entsprechen zwei Ereignisse. Das erste Ereignis tritt beim Scheitern einer Aktivität ein. Das andere Ereignis entspricht dem Abbruch der Aktivität durch den Benutzer (*stop*, siehe Abschnitt 4.5).
- Der Zustandsübergang *skip* beinhaltet zum einen die vom Benutzer ausgeführte Skip-Operation und zum anderen die Eliminierung der Workflow-Instanz, wenn diese sich auf einem "toten" Pfad befindet (Dead-Path-Elimination, [12]).
- Wenn ein komplexer Workflow, der bereits aktiviert worden ist, gestoppt werden soll, so müssen die Sub-Workflows ebenfalls gestoppt werden. Dieses Ereignis führt bei Sub-Workflows, die sich im Zustand *running* befinden, zu dem Zustandsübergang *fail*. Bei einer entsprechenden Ausführungsalternative (vergleiche hierzu Abschnitt 3.2.3), müssen bereits instantiierte Workflow-Instanzen, die später im Kontrollfluß liegen, vom Zustand *open.init* in einen *closed*-Zustand übergehen.

5 Verwandte Arbeiten

In der Forschung und Entwicklung im Bereich Workflow-Management gibt es derzeit eine Reihe von Schwerpunkten, von denen Flexibilisierung und dynamische Veränderungen [7, 20, 28, 29, 5, 6] sowie verteilte Ausführungen [24, 3, 31, 13] und die OMG-Aktivitäten bei der Definition einer Workflow-Facility [18] für unsere Arbeiten besonders relevant sind.

Die Object Management Group möchte in ihren bestehenden Corba-Standard [17] eine Workflow-Management-Facility aufnehmen. Sie startete dazu einen "Request for Proposals" [18], auf den es eine Reihe von Einreichungen namhafter Firmen aus dem Workflow-Bereich gibt [16, 1, 4, 9]. Diese Vorschläge sind in der Regel aus bereits bestehenden Workflow-Management-Systemen entstanden, die meist nicht unter Verwendung von objektorientierten Techniken modelliert oder entwickelt wurden. Aus diesem Grund ist es auch nicht verwunderlich, daß diese Einreichungen eine Reihe von Mängeln aufweisen, die u.a. in [22] aufgezeigt werden. Besonders auffällig ist, daß kaum "objektorientiert" vorgegangen wird, sondern oft bereits bestehende relationale Strukturen in Objekte umgewandelt werden. Allein der Vorschlag der Northern Telecom [16] läßt eine objektorientierte Denkweise erkennen. Der zweite Hauptkritikpunkt besteht in der mangelhaften Integration bestehender Corba-Services und -Facilities, obwohl eine angemessene Einbettung von Workflow-Diensten in den Gesamtkontext der Corba Common Object Services eine vereinfachte Implementierung und Wartung von Workflow-Systemen sowie eine erhöhte Interoperabilität verschiedener Workflow-Anwendungen ermöglichen könnte.

Aspekte der verteilte Ausführung von Workflows werden in den Vorschlägen nur teilweise besprochen. Gerade im Rahmen einer Corba-Umgebung sollten hier jedoch die Vorteile einer verteilten Ausführung genutzt werden. Es ist insbesondere zu berücksichtigen, daß nicht nur die Daten-Objekte verteilt werden können, sondern auch die Workflow-Objekte; wir sind darauf in Abschnitt 3.2.3 kurz eingegangen. Eine ausführliche Diskussion und eine Kritik der Einreichungen bezüglich der Ausschreibung der Workflow-Facility ist bei [22] nachzulesen.

6 Diskussion

Zum Design und zur Implementierung komplexer Software-Systeme ist eine vollständige und konsistente Modellierung des Systems und seiner angestrebten Funktionalität unerläßlich. In diesem Beitrag haben wir ausgehend von Anforderungen an ein flexibles Workflow-Management-System ein Objektmodell vorgestellt, das die wesentlichen Entitäten klassifiziert und ihre Beziehungen zu anderen Entitäten darstellt. Neben einem Analyse-Objektmodell wurde ein Design-Objektmodell entwickelt, das das Analyse-Objektmodell um implementierungstechnische Details erweitert und eine Einbettung der Workflow-Funktionalität in den Kontext der Corba Common Services liefert. Workflow-Objekte nehmen während ihrer Laufzeit eine Reihe von Zuständen an, die sie von ihrer Modellierung, über die Instantiierung als Workflow-Instanzen bis hin zu ihrer Termination und nachfolgenden Anfragen an bereits abgeschlossene Workflow-Instanzen führt. Wir haben Zustände und Zustandsübergänge von Workflow-Objekten unter Verwendung von geschachtelten Zustandsübergangsdiagrammen spezifiziert, wobei insbesondere spezifiziert wurde, welche Operationen in welchen Zuständen zur Verfügung stehen und welche Zustandsübergänge sie bewirken. Ausgehend von der in diesem Beitrag beschriebenen Konzepte implementieren wir derzeit eine zweite Version des Workflow-Management-Systems WASA, dessen generische Architektur in [14] beschrieben wurde. Da wir uns insbesondere für Fragestellungen der Flexibilisierung von Workflow-Systemen interessieren, werden Erfahrungen und bereits entwicklte Konzepte Eingang in den Prototypen finden.

Literatur

[1] CoCreate Software, et al. *Submission to Request for Proposals OMG Workflow Facility (jFlow).* OMG Document bom/97-08-05, 1997. (available from www.omg.org)

[2] Data Access Technologies, Inc., et al. *Combined Business Object Facility Proposal.* OMG Business Object Domain Task Force BODTF-RFP 1 Submission. OMG Document bom/97-11-09, 1997.

[3] A. Dogac, E. Gokkoca. S. Arpinar, P. Koksal, I. Cingil, B. Arpinar, N. Tatbul, P. Karagoz, U. Halici, M. Altinel. *Design and Implementation of a Distributed Workflow Management System: METUFlow.* NATO ASI Workshop, Istanbul, August 12–21, 1997. To appear in Springer ASI NATO Series.

[4] Electronic Data Systems Corporationate Software. *Submission to Request for Proposals OMG Workflow Facility.* OMG Document bom/97-08-06, 1997. (available from www.omg.org)

[5] C. Ellis, K. Keddara, G. Rozenberg. *Dynamic Change Within Workflow Systems.* In Proc. Conference on Organizational Computing Systems (COOCS), Milpitas, CA 1995, 10–22.

[6] D. Georgakopoulos, M. Hornick, A. Sheth. *An Overview of Workflow Mangement: From Process Modeling to Workflow Automation Infrastructure.* Distributed and Parallel Databases, 3:119–153, 1995.

[7] J. Hagemeyer, T. Herrmann, K. Just-Hahn, R. Striemer. *Flexibility in Workflow Systems (in German).* Software-Ergonomie '97, 179–190, Dresden, March 3–6 1997.

[8] IBM. *IBM FlowMark: Modeling Workflow, Version 2 Release 2.* Publ. No SH-19-8241-01, 1996.

[9] Intelligent Systems Technology Inc. *Submission to Request for Proposals OMG Workflow Facility.* OMG Document bom/97-08-07, 1997. (available from www.omg.org)

[10] Iona. *Programming Guide Orbix 2.* Iona Technologies PLC, March 1997

[11] S. Jablonski, C. Bußler. *Workflow-Management: Modeling Concepts, Architecture and Implementation* International Thomson Computer Press, 1996.

[12] F. Leymann, W. Alterhuber. *Managing Business Processes as an Information Resource.* IBM Systems Journal 33, 1994, 326–347.

[13] R. McClatchey, J.-M. LeGoff, N. Baker, W. Harris and Z. Kovacs. *A Distributed Workflow and Product Data Management Application for the Construction of Large Scale Scientific Apparatus.* NATO ASI Workshop, Istanbul, August 12–21, 1997. To appear in Springer ASI NATO Series.

[14] C. Bauzer Medeiros, G. Vossen, M. Weske. *WASA: A workflow-based architecture to support scientific database applications (Extended Abstract).* In Proc. 6th DEXA Conference, London, Spinger LNCS 978, 574–583, 1995.

[15] C. Mohan. *State of the Art in Workflow Management System Research and Products.* Tutorial notes, 5th International Conference on Extending Database Technology, Avignon, France 1996.

[16] Northern Telecom. *Submission to Request for Proposals OMG Workflow Facility.* OMG Document bom/97-08-04, 1997. (available from www.omg.org)

[17] OMG. *CorbaServices: Common Object Services Specification.* (available from www.omg.org)

[18] OMG. *Workflow Management Facility: Request for Proposals.* OMG Document cf/97-05-06, 1997. (available from www.omg.org)

[19] Rational Software et al. *Unified Modeling Language – UML Notation Guide. Version 1.1*, September 1997. (available from www.rational.com/uml)

[20] M. Reichert, P. Dadam. *A Framework for Dynamic Changes in Workflow Management Systems.* Proc. 8th International Workshop on Database and Expert Systems Applications 1997, Toulouse, France. IEEE Computer Society Press, 42–48.

[21] T. Reuß, G. Vossen, M. Weske. *Modeling Samples Processing in Laboratory Environments as Scientific Workflows.* Proc. 8th International Workshop on Database and Expert Systems Applications 1997, Toulouse, France. IEEE Computer Society Press, 49–55.

[22] W. Schulze. *Evaluation of the Submissions to the Workflow Management Facility RFP.* OMG Document bom/97-09-02, 1997.

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

[24] A. Sheth, K.J. Kochut. *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems.* NATO ASI Workshop, Istanbul, August 12–21, 1997. To appear in Springer ASI NATO Series.

[25] J. Siegel. *Corba – Fundamentals and Programming.* John Wiley, 1996.

[26] Sun Microsystems. *Neo Programming Guide.* Sun Microsystems Inc., Mountain View, CA, 1995.

[27] G. Vossen, M. Weske. *The WASA Approach to Workflow Management for Scientific Applications.* NATO ASI Workshop, Istanbul, August 12–21, 1997. To appear in Springer ASI NATO Series.

[28] G. Vossen, M. Weske, G. Wittkowski. *Dynamic Workflow Management on the Web.* Technical Report Angewandte Mathematik und Informatik 24/96-I, Universität Münster, 1996.

[29] M. Weske. *Flexible Modeling and Execution of Workflow Activities.* accepted for 31st Hawaii Conference on System Sciences. IEEE Computer Science Press, 1998.

[30] M. Weske, G. Vossen *Workflow Languages.* To appear in: P. Bernus, K. Mertins, G. Schmidt (Editors): Handbook on Architectures of Information Systems, Springer, 1998.

[31] D. Wodtke, J. Weissenfels, G. Weikum, A. Kotz Dittrich. *The Mentor Project: Steps Towards Enterprise-Wide Workflow Management.* In Proc. 12th IEEE International Conference on Data Engineering (1996), 556–565